(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 601 30 420 T2** 2008.02.28

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 1 186 999 B1**
(21) Deutsches Aktenzeichen: **601 30 420.9**
(96) Europäisches Aktenzeichen: **01 000 209.5**
(96) Europäischer Anmeldetag: **12.06.2001**
(97) Erstveröffentlichung durch das EPA: **13.03.2002**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **12.09.2007**
(47) Veröffentlichungstag im Patentblatt: **28.02.2008**

(51) Int Cl.[8]: ***G06F 9/44*** (2006.01)
G06F 5/06 (2006.01)
G06F 17/30 (2006.01)
G06F 9/46 (2006.01)

(30) Unionspriorität:
594379 **15.06.2000** **US**

(84) Benannte Vertragsstaaten:
AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LI, LU, MC, NL, PT, SE, TR

(73) Patentinhaber:
International Business Machines Corp., Armonk, N.Y., US

(74) Vertreter:
Duscher, R., Dipl.-Phys. Dr.rer.nat., Pat.-Ass., 70176 Stuttgart

(72) Erfinder:
Kirkman, Richard Karl c/o IBM United Kingdom, Winchester, Hampshire SO21 2JN, GB

(54) Bezeichnung: **Vorrichtung und Verfahren zur Aufrechterhaltung einer verknüpften Liste**

Beschreibung

[0001] Die vorliegende Erfindung bezieht sich auf digitale Datenverarbeitungssysteme und insbesondere auf Softwaredatenstrukturen, die im Allgemeinen als verknüpfte Listen bekannt sind.

[0002] Ein modernes Computersystem umfasst üblicherweise eine Zentraleinheit (Central Processing Unit, CPU) und unterstützende Hardware, die für das Speichern, Abrufen und Übertragen von Daten erforderlich ist, wie beispielsweise Datenübertragungsbusse und Speicher. Es beinhaltet außerdem Hardware, die für die Kommunikation mit der Außenwelt notwendig ist, z.B. Ein-/Ausgabe-Steuereinheiten oder Speicher-Steuereinheiten sowie damit verbundene Einheiten wie beispielsweise Tastaturen, Bildschirme, Bandlaufwerke, Festplattenlaufwerke und Datenübertragungsleitungen, die mit einem Netzwerk usw. verbunden sind. Die CPU ist das Herzstück des Systems. Sie führt die Befehle aus, die ein Computerprogramm umfassen, und steuert die Funktionsweise der übrigen Systemkomponenten.

[0003] Aus Sicht der Computerhardware arbeiten die meisten Systeme im Wesentlichen auf die gleiche Art und Weise. Prozessoren können einen begrenzten Satz sehr einfacher Operationen ausführen, z.B. arithmetische, logische Vergleiche und die Verschiebung von Daten von einer Speicherstelle zu einer anderen. Jede Operation wird jedoch sehr schnell ausgeführt. Programme, die einen Computer so steuern, dass er eine enorme Anzahl dieser einfachen Operationen ausführt, erwecken den Eindruck, als führe der Computer eine komplizierte Aufgabe durch.

[0004] Die Tatsache, dass Computersysteme für eine große Vielzahl von Aufgaben eingesetzt werden, lässt sich größtenteils auf die Fülle und Vielfalt von Software zurückführen, die den Ausführung von Computerprozessoren steuert. Im Laufe der Jahre wurde die Software ebenso wie die Hardware stetig weiterentwickelt und dabei immer komplexer. Obwohl eine bestimmte Softwareanwendung äußerst spezialisiert sein kann, gibt es bestimmte gemeinsame Methoden und Strukturen, die von den Fachleuten immer wieder verwendet werden. Eine Art und Weise, wie Software und damit auch die Computersysteme, die zu einem wesentlichen Teil aus Software bestehen, verbessert werden können, besteht darin, Verbesserungen an diesen häufig verwendeten Softwaremethoden und -strukturen vorzunehmen.

[0005] Softwareentwickler wissen, wie wichtig die Strukturierung von Daten ist, d.h. die logische Anordnung von Daten, so dass diese ohne zu großen Aufwand abgerufen und bearbeitet werden können. Die von einer Softwareanwendung zu verwendende Datenstruktur ist eines der ersten Dinge, das bei der Planung und Entwicklung der Anwendung festgelegt werden muss. Obwohl nahezu jede maßgeschneiderte Datenstruktur erzeugt werden kann, verwenden Softwareentwickler immer wieder eine Reihe von gängigen Datenstrukturarten. Eine der am weitesten verbreiteten Strukturen ist die verknüpfte Liste, die einfach oder doppelt verknüpft sein kann.

[0006] Eine verknüpfte Liste ist eine Sammlung von Datenelementen, bei der jedes Element in der Sammlung einen Verweis auf das nächste Element in der Sammlung enthält. Dabei kann ein einzelnes Element zusätzlich zu dem Verweis nur eine einzige variable beinhalten, oder es kann sich um eine komplexe Datenstruktur handeln, die viele unterschiedliche Datenarten wie beispielsweise Textfolgen, numerische Daten, Zeiger usw. enthält, oder es kann sich um ein Leerliste handeln, die außer dem Verweis nichts weiter enthält. Der Verweis ist üblicherweise ein Zeiger auf die Speicheradresse des nächsten Listenelements, kann jedoch auch die Form eines Anordnungsindex oder einer anderweitigen Größe annehmen, aus dem/der eine Adresse oder eine anderweitige Speicherstelle des nächsten Elements abgeleitet werden kann. Der Einfachheit halber wird der Verweis innerhalb eines verknüpften Listenelements im Folgenden als „Zeiger" bezeichnet, wobei dies unabhängig davon ist, ob er direkt auf eine Adresse zeigt oder indirekt etwas indiziert, aus dem wie oben beschrieben eine Adresse erhalten wird. Die Adressen oder sonstigen Speicherverweise verknüpfter Listenelemente müssen nicht zwingend eine bestimmte Reihenfolge einhalten, so dass es möglich ist, ein Element in eine verknüpfte Liste einzufügen bzw. daraus zu entfernen, ohne dass alle oder eine größere Zahl von Elementen in der Liste neu zugeordnet werden müssen. Es ist lediglich erforderlich, die Zeiger auf benachbarte Elemente zu ändern. Eine verknüpfte Liste kann einfach verknüpft sein und über Vorwärtszeiger verfügen, die nur in eine einzige Richtung zeigen, oder sie kann doppelt verknüpft sein und sowohl Vorwärts- als auch Rückwärtszeiger aufweisen.

[0007] Besondere Sorgfalt ist bei der Aktualisierung einer verknüpften Liste in einer Mehrprogrammumgebung geboten. Wenn gleichzeitig mehrere Programmsegmente versuchen, ein und dieselbe verknüpfte Liste zu ändern, oder wenn ein einziges Programmsegment versucht, die Liste zu ändern, während ein anderes Programmsegment sie durchläuft, kann es zu unvorhersagbaren Ergebnissen kommen, und Daten werden mög-

licherweise beschädigt. Dieses Problem ist besonders schwerwiegend, wenn eine einzige CPU den Hardware-Mehrpfadbetrieb unterstützt oder wenn mehrere CPUs gleichzeitige mehrere Programmsegmente ausführen, die alle auf einen gemeinsamen Speicher zugreifen.

[0008] Bei einer einfach verknüpften Liste kann in manchen Umgebungen eine Liste aktualisiert werden, indem bestimmte Arten von atomaren Operationen verwendet werden. Allerdings eignet sich dieser Ansatz nicht für alle Umgebung und lässt sich im Allgemeinen nicht auf doppelt verknüpfte Listen anwenden, bei denen Zeiger in zwei verschiedenen Listenelementen aktualisiert werden müssen. Eine allgemeiner angelegte Lösung besteht in einem System von Sperren. Dabei erhält eine Aufgabe, die eine verknüpfte Liste aktualisieren möchte, üblicherweise eine exklusive Sperre für die Liste, was bedeutet, dass andere Aufgaben erst dann wieder auf die Liste zugreifen können, nachdem die Aktualisierung abgeschlossen wurde. Eine Aufgabe, die eine Liste lediglich durchlaufen möchte, ohne sie zu ändern, kann eine gemeinsam genutzte Sperre erhalten, mit der andere Aufgaben an der Änderung der Liste gehindert werden, ohne dass ihnen notwendigerweise auch das Durchlaufen der Liste untersagt wird.

[0009] Die Verwendung von Sperren ist einfach und wirksam, kann sich jedoch nachteilig auf die Leistung auswirken. Es kann einige Zeit in Anspruch nehmen, die Liste zu durchlaufen und dabei nach einem bestimmten Element zu suchen. Während dieser Zeit werden alle anderen Prozesse an der Änderung der Liste bzw. – je nach Art der Sperre – unter Umständen auch am Durchlaufen der Liste gehindert. Bei kleinen Listen und solchen in isolierten Anwendungen dürfte dies kein Problem darstellen. Mit einiger Wahrscheinlichkeit treten auch Listen mit großem Umfang und häufigem Zugriff auf, z.B. eine Liste, mit welcher der Status von Aufgaben des Betriebssystems verwaltet wird. Wenn derartige Listen exklusiv gesperrt werden, kann es zu Leistungsengpässen kommen, wobei derartige Engpässe mit höherer Wahrscheinlichkeit bei einem System mit mehreren CPUs auftreten.

[0010] Es besteht ein Bedarf nach einem Verfahren für die Aktualisierung verknüpfter Listen ohne die Nachteile der Verfahren nach dem Stand der Technik und insbesondere nach einem Verfahren für die Aktualisierung verknüpfter Listen einschließlich doppelt verknüpfter Listen in einer Mehrprozessorumgebung, das sich weniger störend auf Aufgaben auswirkt, die von anderen Prozessoren ausgeführt werden.

[0011] Die vorliegende Erfindung stellt gemäß einem ersten Aspekt ein Verfahren nach Anspruch 1 bereit.

[0012] Vorzugsweise umfasst die Hilfsdatenstruktur eine Liste von Umgehungselementen, wobei jedes Umgehungselement einen Verweis auf ein Listenelement unmittelbar nach einem gesperrten Teil der Liste enthält, wobei der Schritt des Aufzeichnens von Sperrdaten das Ändern eines Verweises auf ein erstes Listenelement des ersten Teils der verknüpften Liste umfasst, so dass dieser stattdessen auf ein Umgehungselement verweist, wobei eine Aufgabe, welche die verknüpfte Liste durchläuft, ohne sie jedoch zu ändern, die verknüpfte Liste über das Umgehungselement und den ersten Teil der verknüpften Liste durchläuft.

[0013] Vorzugsweise handelt es sich bei der verknüpften Liste um eine doppelt verknüpfte Liste.

[0014] Vorzugsweise besteht der erste Teil der verknüpften Liste, der durch den Schritt des Aufzeichnens von Sperrdaten gesperrt ist, aus höchstens drei Verknüpfungen.

[0015] Vorzugsweise umfasst das Verfahren des ersten Aspekts weiterhin die folgenden Schritte: Veranlassen einer zweiten Aktualisierungsaufgabe für das Aktualisieren der verknüpften Liste über die Hilfsdatenstruktur; Aufzeichnen von Sperrdaten innerhalb der Hilfsdatenstruktur, um zu verhindern, dass Aufgaben mit Ausnahme der zweiten Aktualisierungsaufgabe, die über die Hilfsdatenstruktur auf die verknüpfte Liste zugreifen, einen zweiten Teil der verknüpften Liste ändern, wobei der zweite Teil nicht alle Element der verknüpften Liste enthält; und Aktualisieren der verknüpften Liste innerhalb des zweiten Teils der verknüpften Liste, indem eine Aktion aus der Gruppe von Aktionen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen eines Elements innerhalb des zweiten Teils der verknüpften Liste aus der verknüpften Liste und (b) Einfügen eines Elements in die verknüpfte Liste innerhalb des zweiten Teils der verknüpften Liste, wobei der Schritt des Aktualisierens der verknüpften Liste innerhalb des zweiten Teils gleichzeitig mit dem Schritt des Aktualisierens der verknüpften Liste innerhalb des ersten Teils ausgeführt wird.

[0016] Vorzugsweise umfasst das Verfahren des ersten Aspekts weiterhin den folgenden Schritt: Weitergeben eines Zeigers auf eine frei wählbare Verknüpfung innerhalb der verknüpften Liste, um einen Punkt für das Ändern der verknüpften Liste zu finden, wobei der erste Teil, der durch den Schritt des Aufzeichnens von Sperrdaten gesperrt wird, zumindest einen Teil der frei wählbaren Verknüpfung enthält.

[0017] Vorzugsweise ist die Hilfsdatenstruktur ein Objekt in einer Klassenhierarchie einer objektorientierten Programmierstruktur.

[0018] Vorzugsweise beinhaltet die Klassenhierarchie eine erste Klasse für das Durchlaufen der verknüpften Liste, ohne sie zu ändern, und eine zweite Klasse für Objekte, welche die verknüpfte Liste durchlaufen und sie ändern.

[0019] Vorzugsweise beinhaltet die Klassenhierarchie weiterhin eine dritte Klasse für Objekte, welche die verknüpfte Liste durchlaufen, ohne sie zu ändern, und über einen Sperrzugriff auf Listenebene verfügen, eine vierte Klasse für Objekte, welche die verknüpfte Liste durchlaufen, ohne sie zu ändern, und über einen Sperrzugriff auf Verknüpfungsebene verfügen, eine fünfte Klasse für Objekte, welche die verknüpfte Liste durchlaufen und sie ändern und über einen Sperrzugriff auf Listenebene verfügen, sowie eine sechste Klasse für Objekte, welche die verknüpfte Liste durchlaufen und sie ändern und über einen Sperrzugriff auf Verknüpfungsebene verfügen, wobei die dritte und die vierte Klasse von der ersten Klasse erben und die fünfte und die sechsten Klasse von der zweiten Klasse erben.

[0020] Vorzugsweise ist für die verknüpfte Liste eine Ordnungsbeziehung definiert.

[0021] Vorzugsweise umfasst das Verfahren des ersten Aspekts weiterhin die folgenden Schritte: Kenntlichmachen einer ersten Gruppe von aufeinanderfolgenden Verknüpfungen in einem relevanten Teil der doppelt verknüpften Liste, wobei die erste Gruppe von aufeinanderfolgenden Verknüpfungen nicht alle Verknüpfungen der doppelt verknüpften Liste umfasst; Verhindern, dass Aufgaben mit Ausnahme der ersten Aufgabe zumindest einen Teil von Verknüpfungen der ersten Gruppe von aufeinanderfolgenden Verknüpfungen ändern, und gleichzeitig Zulassen, dass Aufgaben mit Ausnahme der ersten Aufgabe Verknüpfungen der doppelt verknüpften Liste, die nicht in der ersten Gruppe enthalten sind, ändern; Ändern der doppelt verknüpften Liste innerhalb der ersten Gruppe, indem eine Operation aus der Gruppe von Operationen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen einer Verknüpfung der ersten Gruppe aus der doppelt verknüpften Liste und (b) Einfügen einer Verknüpfung in die doppelt verknüpfte Liste zwischen zwei Verknüpfungen der ersten Gruppe, wobei die Operation von der ersten Aufgabe ausgeführt wird; und Aufheben der Sperre, die Aufgaben mit Ausnahme der ersten Aufgabe daran hindert, zumindest einen Teil der Verknüpfungen der ersten Gruppe zu ändern, nachdem der Änderungsschritt ausgeführt wurde.

[0022] Vorzugsweise besteht die erste Gruppe von aufeinanderfolgenden Verknüpfungen der verknüpften Liste, die durch den Schritt des Sperrens von Aufgaben gesperrt wird, aus höchstens drei Verknüpfungen.

[0023] Vorzugsweise umfasst der Schritt des Kenntlichmachens einer ersten Gruppe von aufeinanderfolgenden Verknüpfen das Empfangen eines Zeigers auf eine frei wählbare Verknüpfung innerhalb der verknüpften Liste, um so einen Punkt für das Ändern der verknüpften Liste ausfindig zu machen, wobei sich die frei wählbare Verknüpfung innerhalb der ersten Gruppe befindet.

[0024] Vorzugsweise umfasst das Verfahren des ersten Aspekts weiter die folgenden Schritte: Kenntlichmachen einer zweiten Gruppe von aufeinanderfolgenden Verknüpfungen in einem relevanten Teil der doppelt verknüpften Liste, wobei die zweite Gruppe von aufeinanderfolgenden Verknüpfungen nicht alle Verknüpfungen der doppelt verknüpften Liste umfasst; Verhindern, dass Aufgaben mit Ausnahme einer zweiten Aufgabe mindestens einen Teil von Verknüpfungen der zweiten Gruppe von aufeinanderfolgenden Verknüpfungen ändern, und gleichzeitig Zulassen, dass Aufgaben mit Ausnahme der zweiten Aufgabe Verknüpfungen der doppelt verknüpften Liste, die nicht in der zweiten Gruppe enthalten sind, ändern; Ändern der doppelt verknüpften Liste innerhalb der zweiten Gruppe, indem eine Operation aus der Gruppe von Operationen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen einer Verknüpfung der zweiten Gruppe aus der doppelt verknüpften Liste und (b) Einfügen einer Verknüpfung in die doppelt verknüpfte Liste zwischen zwei Verknüpfungen der zweiten Gruppe, wobei die Operation von der zweiten Aufgabe ausgeführt wird; und Aufheben der Sperre, die Aufgaben mit Ausnahme der zweiten Aufgabe daran hindert, zumindest einen Teil der Verknüpfungen der zweiten Gruppe zu ändern, nachdem der Schritt des Änderns der doppelt verknüpften Liste innerhalb der zweiten Gruppe ausgeführt wurde; wobei der Schritt des Änderns der doppelt verknüpften Liste innerhalb der zweiten Gruppe nach dem Schritt, mit dem Aufgaben mit Ausnahme der ersten Aufgabe daran gehindert werden, Verknüpfungen der ersten Gruppe zu ändern, und vor dem Schritt ausgeführt wird, mit dem die Sperre, die Aufgaben mit Ausnahme der ersten Aufgabe daran gehindert hat, Verknüpfungen der ersten Gruppe zu ändern, aufgehoben wird.

[0025] Gemäß einem zweiten Aspekt stellt die vorliegende Erfindung ein Computerprogramm bereit, das,

wenn es in einen Computer geladen und ausgeführt wird, die Schritte eines Verfahrens gemäß dem ersten Aspekt ausführt.

[0026] Gemäß einer bevorzugten Ausführungsform der vorliegenden Erfindung verfügt eine verknüpfte Liste über eine ihr zugehörige Hilfsdatenstruktur, die externe Verweise auf die verknüpfte Liste enthält, welche von Aktualisierungsaufgaben verwendet werden. Mit der Hilfsdatenstruktur wird verhindert, dass Teilgruppen der verknüpften Liste von anderen Aufgaben geändert werden, wodurch gleichzeitige Aktualisierungen verschiedener Teilgruppen ermöglicht werden.

[0027] Bei der bevorzugten Ausführungsform erfolgt die Verwaltung der verknüpften Liste unter Verwendung von objektorientierten (OO) Programmiermethoden. Dabei ist die Liste vorzugsweise eine doppelt verknüpfte Liste. Die Hilfsdatenstruktur ist ein Objekt mit der Bezeichnung SharedList, das Iterator-Objekte einschließlich Inspektor-Objekten und Mutator-Objekten, Sperrobjekten und Anzeigeobjekten umfasst. Iterator-Objekte werden von Clients für den externen Zugriff auf die verknüpfte Liste verwendet. Ein Inspektor-Objekt ist eine Art von Iterator-Objekt, das die Liste durchläuft, ohne sie zu ändern; ein Mutator-Objekt ist eine Art von Iterator-Objekt, das die Liste ändern kann und sie unter Umständen ebenfalls durchläuft. Sowohl Inspektoren als auch Mutatoren verfügen über ihnen zugehörige Anzeigeobjekte, welche die Zeiger auf die eigentlichen Listenelemente enthalten. Mutatoren (jedoch nicht Inspektoren) verfügen darüber hinaus über ihnen zugehörige Sperrobjekte, die den Zugriff auf ausgewählte Listenelemente sperren. Während des Betriebs sperrt ein Mutator-Objekt einen Teil der verknüpften Liste, indem es Zeiger so ändert, dass sie auf das Sperrobjekt zeigen, wobei das Sperrobjekt zusätzliche Zeiger enthält, die den gesperrten Listenteil umgehen. Danach können andere Inspektoren die Liste über den umgegangenen Teil durchlaufen, wohingegen Mutatoren, welche die Liste durchlaufen, an dem gesperrten Teil aufgehalten werden.

[0028] Indem nur ein verhältnismäßig kleiner Teil einer umfangreichen verknüpften Liste gesperrt wird, können mehrere Aufgaben gleichzeitig auf verschiedene Listenelemente zugreifen und getrennte Listenaktualisierungen vornehmen, wodurch – insbesondere bei Verwendung mehrerer Prozessoren – die Leistung verbessert wird. Indem die Sperre auf der Ebene der einzelnen Listenelemente und nicht der gesamten Liste erfolgt, können bei der bevorzugten Ausführungsform genauer gesagt mehrere Mutatoren gleichzeitig verschiedene Teile einer verknüpften Liste durchlaufen und ändern, ohne miteinander in Konflikt zu geraten, während Inspektoren gleichzeitig eine Liste durchlaufen können, die von einem oder mehreren Mutatoren geändert wird.

[0029] Im Folgenden wird eine bevorzugte Ausführungsform der vorliegenden Erfindung beispielhaft und mit Bezug auf die beigefügten Zeichnungen beschrieben, wobei:

[0030] Fig. 1 ein Übersichts-Blockschaltbild der wichtigsten Hardwarekomponenten eines Computersystems für die Verwendung einer verknüpften Listendatenstruktur gemäß der bevorzugten Ausführungsform der vorliegenden Erfindung ist.

[0031] Fig. 2 eine Übersichts-Entwurfsdarstellung der wichtigsten Softwarekomponenten des Speichers **102** des Computersystems gemäß der bevorzugten Ausführungsform ist.

[0032] Fig. 3 eine Übersichtsdarstellung der verknüpften Listendatenstruktur gemäß der bevorzugten Ausführungsform ist.

[0033] **Fig.** 4 die von außen betrachtete Hierarchie der Objektklassenvererbung innerhalb der verknüpften Listendatenstruktur gemäß der bevorzugten Ausführungsform zeigt.

[0034] **Fig.** 5 die Beziehungen zwischen den verschiedenen externen und internen Klassen und ihren Vererbungshierarchien gemäß der bevorzugten Ausführungsform zeigt.

[0035] Fig. 6 eine vereinfachte Ansicht eines beispielhaften SharedChain-Objekts mit drei Verknüpfungen gemäß der bevorzugten Ausführungsform zeigt.

[0036] Fig. 7 zeigt, wie gemäß der bevorzugten Ausführungsform mit Anzeigeobjekten auf Elemente in einer verknüpften Liste verwiesen werden kann.

[0037] Fig. 8 die Beziehungen zwischen SharedChain-Objekten, Sperrobjekten und Verknüpfungen zeigt, indem ein Beispiel für ein Sperrobjekt gemäß der bevorzugten Ausführungsform gezeigt wird, das eine Verknüpfung gesperrt hat.

[0038] Fig. 9 ein Beispiel für ein Sperrobjekt gemäß der bevorzugten Ausführungsform für einen Mutator zeigt, der die Verknüpfungszeiger, die benötigt werden, um eine Verknüpfung in die Kette zwischen zwei anderen Verknüpfungen einzufügen, gesperrt hat.

[0039] Fig. 10 ein Beispiel für ein Sperrobjekt gemäß der bevorzugten Ausführungsform für einen Mutator zeigt, der die Verknüpfungszeiger, die benötigt werden, um eine Verknüpfung aus der Kette zu entfernen, gesperrt hat.

[0040] **Fig.** 11 ein Beispiel für die Interaktion zwischen mehreren Anzeige- und Sperrobjekten für durchlaufende Mutatoren gemäß der bevorzugten Ausführungsform zeigt.

ÜBERSICHT ÜBER DAS SYSTEM

[0041] Mit Bezug auf die Zeichnungen, in denen identische Ziffern in den verschiedenen Ansichten identische Teile kennzeichnen, zeigt Fig. 1 die wichtigsten Hardwarekomponenten eines Computersystems **100** für die Verwendung einer verknüpften Listendatenstruktur gemäß der bevorzugten Ausführungsform der Erfindung. Die Zentraleinheiten (CPUs) **101A**, **101B**, **101C**, **101D** führen grundlegende Maschinenverarbeitungsfunktionen von Befehlen und Daten aus dem Hauptspeicher **102** durch. Vorzugsweise verfügt jeder Prozessor über einen ihm zugehörigen Zwischenspeicher **103A**, **103B**, **103C**, **103D** für das Speichern von Daten unter Einschluss von Befehlen, die von dem Prozessor ausgeführt werden. Der Einfachheit halber sind CPUs und Zwischenspeicher im Folgenden durchgängig durch die Ziffern **101** bzw. **103** gekennzeichnet. Der Speicherbus **109** überträgt Daten zwischen den CPUs **101** und dem Speicher **102**. Die CPUs **101** und der Speicher **102** übertragen auch über den Speicherbus **109** und die Busschnittstelle **105** Daten mit dem E/A-Bus **110**. Verschiedene E/A-Verarbeitungseinheiten (E/A-VEs) **111** bis **115** sind mit dem System-E/A-Bus **110** verbunden und unterstützen die Datenübertragung mit einer Anzahl verschiedener Speicher- und E/A-Einheiten, zu denen beispielsweise Direktzugriffsspeicher (DASDs) **121** bis **123**, Datenspeichereinheiten mit sich drehenden Magnetlaufwerken, Bandlaufwerke, Arbeitsplatzrechner, Drucker und Fern-Datenübertragungsleitungen für den Datenaustausch mit entfernten Einheiten oder anderen Computersystemen gehören.

[0042] Es sollte ferner klar sein, dass Fig. 1 lediglich als ein Beispiel für eine Systemkonfiguration zu verstehen ist und dass die tatsächliche Anzahl, Art und Konfiguration von Komponenten in einem Computersystem davon abweichen kann. Insbesondere können Ausführungsformen der vorliegenden Erfindung in Systemen mit einer einzigen CPU sowie in Systemen mit mehreren CPUs eingesetzt werden, wobei die Anzahl der CPUs variieren kann. Obwohl in Fig. 1 verschiedene Busse dargestellt sind, sollte klar sein, dass sie auf einem Entwurfsniveau für verschiedene Datenübertragungspfade stehen sollen und dass die tatsächliche physische Konfiguration von Bussen davon abweichen und in der Realität sehr viel komplexer sein kann. Obwohl in der Figur jeder CPU **101** ein einziger Zwischenspeicher **103** zugehörig ist, können außerdem auch mehrere Ebenen von Zwischenspeichern vorhanden sein, und einige Zwischenspeicher können für Befehle vorgesehen sein, während andere für nicht ausführbare Daten vorgesehen sein können, oder es können alternativ dazu keinerlei Zwischenspeicher vorhanden sein. Bei der bevorzugten Ausführungsform ist das Computersystem **100** ein IBM AS/400-Computersystem, wobei klar sein sollte, dass auch andere Computersysteme verwendet werden können.

[0043] Fig. 2 ist eine Entwurfsdarstellung der wichtigsten Softwarekomponenten des Speichers **102**. Das Betriebssystem **201** stellt verschiedene einfache Softwarefunktionen bereit, zu denen Einheitenschnittstellen, die Verwaltung von mehreren Aufgaben, die Verwaltung der Seitenspeichertechnik usw. gehören, wie in der Technik hinreichend bekannt ist. Bei der bevorzugten Ausführungsform beinhaltet das Betriebssystem **201** Code, der in der Regel Betriebssystemfunktionen für ein IBM AS/400-System ausführt, darunter eine Funktion, die mitunter auch als lizenzierter interner Code bezeichnet wird, sowie OS/400-Betriebssystemcode, wobei klar sein sollte, dass auch andere Betriebssysteme verwendet werden können. Die Datenbank **202** wird auf Entwurfsniveau als eine allgemeine Zusammenstellung von Daten dargestellt, die für die Ausführung von Anwendungen verwendet werden. Dabei sollte klar sein, dass viele verschiedene Arten von Datenbanken beispielsweise mit Finanzdaten, Formularen und Textdaten, Bibliotheksdaten, medizinischen Daten usw. verwendet werden können. Obwohl in Fig. 2 eine einzige Beispieldatenbank abgebildet ist, kann das Computersystem **100** mehrere Datenbanken oder auch überhaupt keine Datenbanken enthalten. Die Anwendungen **203** bis **205** sind Programme, die bestimmte Aufgaben für einen Benutzer ausführen, indem sie üblicherweise auf Daten in einer Datenbank **202** zugreifen. So können die Anwendungen beispielsweise Daten in der Datenbank **202** suchen und abrufen, komplexe Berechnungen mit Daten durchführen, um weitere Daten zu erhalten, oder eine beliebige Anzahl von anderen Anwendungen ausführen. Drei Anwendungen werden in Fig. 2 gezeigt, wobei klar sein dürfte, dass die Anzahl der Anwendungen schwanken kann und in der Regel weit über drei liegt.

[0044] Das Betriebssystem **201**, die Datenbank **202** und die Anwendung **203** beinhalten verknüpfte Listendatenstrukturen **210**, **211**, **212**. Diese Strukturen enthalten verknüpfte Listen von Datenelementen. Diese Datenelemente, die im Folgenden als „Verknüpfungen" bezeichnet werden, sind die grundlegenden Einheiten der Liste, wobei die Liste eine beliebige Anzahl derartiger Elemente oder Verknüpfungen enthält. Bei der bevorzugten Ausführungsform ist die Liste eine doppelt verknüpfte Liste, und somit beinhaltet jedes Element bzw. jede Verknüpfung mindestens zwei Zeiger, einen auf die als nächstes folgende Verknüpfung und den anderen auf die vorhergehende Verknüpfung in der Abfolge von Verknüpfungen. Verknüpfungen können wahlweise auch zusätzliche Zeiger, z.B. auf ein Listenobjekt, beinhalten. Alternativ könnte auch eine Liste mit nur einer einzigen Verknüpfung verwendet werden, bei der jedes Element einen Vorwärtszeiger auf die nächste Verknüpfung, jedoch keinen Rückwärtszeiger auf die vorherige Verknüpfung beinhaltet. Gemäß der bevorzugten Ausführungsform beinhaltet jede verknüpfte Listendatenstruktur **210** bis **212** ferner Hilfsdatenstrukturen, die für den Zugriff auf und die Aktualisierung der Daten in der verknüpften Liste verwendet werden, wie im Folgenden ausführlicher beschrieben wird.

[0045] Obwohl in Fig. 2 eine bestimmte Anzahl und Art von Software-Einheiten abgebildet sind, dürfte offensichtlich sein, dass diese lediglich aus Gründen der Anschaulichkeit abgebildet sind und dass die tatsächliche Anzahl derartiger Einheiten variieren kann. Konkret sind die verknüpften Datenlistenstrukturen aus Gründen der Anschaulichkeit als im Betriebssystem **201**, in der Datenbank **202** und im Anwendungsprogramm **203** befindlich abgebildet, wobei sich derartige verknüpfte Datenlistenstrukturen jedoch auch ausschließlich im Betriebssystem oder ausschließlich in der Datenbank befinden oder aber in einem Anwendungsprogramm bzw. als separate Datenstrukturen vorhanden sein können, die nicht Teil anderer strukturierter Daten oder zu eines anderen Programms sind. Obwohl die Softwarekomponenten aus Fig. 2 in der Entwurfsdarstellung als im Speicher befindlich dargestellt sind, dürfte außerdem klar sein, dass der Speicher eines Computersystems im Allgemeinen zu klein ist, um alle Programme und sonstigen Daten gleichzeitig aufzunehmen, und dass die Daten üblicherweise in einem Datenspeicher wie beispielsweise den Datenspeichereinheiten **121** bis **123** gespeichert werden, von denen aus sie bei Bedarf durch das Betriebssystem seitenweise in den Speicher umgelagert werden.

[0046] Die Routinen, die zur Realisierung der abgebildeten Ausführungsformen der Erfindung ausgeführt werden, werden im Folgenden generell und unabhängig davon, ob sie als Teil eines Betriebssystems oder einer bestimmten Anwendung, eines bestimmten Programms, Objekts, Moduls oder einer bestimmten Abfolge von Befehlen realisiert sind, als „Computerprogramme" bezeichnet. Die Computerprogramme umfassen üblicherweise Befehle, die, wenn sie von einem oder mehreren Prozessoren in den Einheiten oder Systemen eines Computersystems, das den Ausführungsformen der Erfindung entspricht, gelesen und ausgeführt werden, diese Einheiten bzw. Systeme dazu veranlassen, die notwendigen Schritte auszuführen, um Schritte auszuführen oder Elemente zu erzeugen, welche die verschiedenen Aspekte der vorliegenden Erfindung beinhalten. Obwohl die Ausführungsformen der Erfindung bisher hierher in Zusammenhang mit voll funktionsfähigen Computersystemen beschrieben wurden und im Folgenden auch weiterhin so beschrieben werden, können die verschiedenen Ausführungsformen der Erfindung auch in verschiedenen Formen als ein Programmprodukt verteilt werden und ist die Erfindung gleichermaßen anwendbar, unabhängig von der konkreten Art des signaltragenden Mediums, das für die Verteilung tatsächlich verwendet wird. Beispiele für signaltragende Medien sind, ohne darauf beschränkt zu sein, beschreibbare Medien wie flüchtige und nichtflüchtige Speichereinheiten, Disketten, Festplattenlaufwerke, CD-ROMS, DVDs und Magnetbänder sowie Medien des Übertragungstyps wie digitale und analoge Datenübertragungsleitungen einschließlich Funkverbindungen. Beispiele für signaltragende Medien sind in Fig. 1 als DASD **121** bis **123** und Speicher **102** abgebildet.

VERKNÜPFTE LISTENSTRUKTUR: EXTERNE SCHNITTSTELLE

[0047] Bei der bevorzugten Ausführungsform sind die relevanten Softwaredatenstrukturen und der Programmiercode unter Verwendung objektorientierter (OO) Programmiermethoden geschrieben, wobei klar sein sollte, dass auch herkömmliche prozedurorientierte oder anderweitige Programmiermethoden verwendet werden könnten. In der folgenden Beschreibung wird OO-Terminologie verwendet, wobei klar sein sollte, dass es generell auch analoge Datenstrukturen, Codesegmente oder anderweitige Einheiten gibt, die bei anderen Programmiermethoden zum Einsatz kommen. Die verknüpfte Listendatenstruktur **210** bis **212** ist ein Objekt mit der Bezeichnung SharedList.

[0048] Ein SharedList-Objekt stellt ein Mittel zur Verwaltung einer doppelt verknüpften Liste von Clientobjekten bereit, weist jedoch eine Reihe von eindeutigen Merkmalen auf, d.h. (a) Clients können die Liste sicher durchlaufen, während Verknüpfungen zu ihr hinzugefügt und/oder aus ihr entfernt werden; und (b) mehrere Clients können unabhängig voneinander und gleichzeitig Verknüpfungen zu der Liste hinzufügen bzw. aus der

Liste entfernen. Das SharedList-Objekt ermöglicht allen Clients den gleichzeitigen Zugriff auf die Liste. Es stellt die Sperrung von Aufgaben auf der Verknüpfungsebene bereit, wobei nur diejenigen Aufgaben von der Sperrung betroffen sind, die an den jeweiligen Verknüpfungskonflikten beteiligt sind.

SharedList<T>-Objekte

[0049] Mit Blick auf Fig. 3, bei der es sich um eine Übersichtsdarstellung der verknüpften Listendatenstruktur **210**, **211** handelt, stellt die SharedList<T>-Klasse eine vorlagenbasierte Unterstützung (templatized support) für eine SharedList (gemeinsam genutzte Liste) von Clientobjekten eines beliebigen Typs T bereit. Das SharedList<T>-Objekt **301** enthält Zeiger auf das erste Client-Verknüpfungsobjekt **302** und das letzte Client-Verknüpfungsobjekt **303** in der Liste sowie einen Zählwert für die Anzahl der Verknüpfungen in der Liste. Es enthält außerdem Versatzwerte für die Vorwärts- und Rückwärtszeiger in jeder Verknüpfung; die Definition von Feldern für diese Zeiger ist Aufgabe der Verknüpfungsobjekte.

[0050] Zusätzlich kann jede Verknüpfung wahlweise ein Feld definieren, das auf das SharedList<T>-Objekt selbst zeigt. Wenn dieses Feld definiert ist und eine Verknüpfung in die Liste eingefügt wird, wird der Zeiger gesetzt; wenn die Verknüpfung entfernt wird, wird der Zeiger auf NULL zurückgesetzt. Je nach Anwendung können Clients mit dem Zeiger ermitteln, ob sich eine gegebene Verknüpfung in einer Liste befindet und in welcher Liste sie sich befindet, indem sie dieses Feld abfragen. So kann es z.B. vorkommen, dass der Destruktor eines Clientobjekts das Objekt aus der Liste entfernen möchte, in der es sich befindet.

[0051] Fig. 3 zeigt, wie ein SharedList<T>-Objekt betrachtet werden sollte. Der Versatz des wahlfreien Felds in den Verknüpfungsobjekten, das auf das SharedList<T>-Objekt zeigt, ist eine Ganzzahl mit Vorzeichen. Ein negativer Wert des Versatzfeldes gibt an, dass die Verknüpfungen kein Zeigerfeld enthalten.

Funktionen von SharedList<T>

[0052] Es folgt eine Beschreibung der wichtigsten Funktionen, die durch das SharedList<T>-Objekt **301** unterstützt werden:
void insertFirst(T* newLink): Diese Funktion fügt die angegebene Verknüpfung als erste Verknüpfung in die Liste ein.

[0053] void insertLast(T* newLink): Diese Funktion fügt die angegebene Verknüpfung als letzte Verknüpfung in die Liste ein.

[0054] void insertBeforeArbitraryLink(T* newLink, T* refLink): Diese Funktion fügt die angegebene Verknüpfung unmittelbar vor der frei wählbaren Verknüpfung ein, auf die verwiesen wird. Eine frei wählbare Verknüpfung ist eine Verknüpfung, die der Client von vorn herein kennt und deren Vorhandensein in der Liste er sicherstellt und die jede beliebige Position in der Liste einnehmen kann. Wie hier erläutert, kann eine Liste durchlaufen werden, um einen Einfügepunkt für eine Verknüpfung oder eine zu löschende Verknüpfung zu finden, wobei es jedoch auch möglich ist, mitten in die Liste zu springen und eine frei wählbare Verknüpfung einzufügen oder zu löschen, sofern der Client über einen Zeiger auf die Verknüpfung mitten in der Liste verfügt. Die Funktion insertBeforeArbitraryLink ist eine von mehreren Funktionen, die verwendet werden, wenn der Client über einen derartigen Zeiger verfügt. Es ist Aufgabe des Client (oder einer anderen externen Einrichtung) sicherzustellen, dass die Verknüpfung tatsächlich in der Liste enthalten ist; wenn dies nicht der Fall ist, kann es zu unvorhersagbaren Ergebnissen kommen.

[0055] void insertAfterArbitraryLink(T* newLink, T* refLink): Diese Funktion fügt die angegebene Verknüpfung unmittelbar nach der frei wählbaren Verknüpfung ein, auf die verwiesen wird.

[0056] T* removeFirstLink(): Diese Funktion entfernt die erste Verknüpfung aus der Liste und gibt einen Zeiger auf sie zurück.

[0057] T* removeLastLink(): Diese Funktion entfernt die letzte Verknüpfung aus der Liste und gibt einen Zeiger auf sie zurück. Diese Funktion erfordert eine (weiter unten beschriebene) atomare Sicht der Liste und kann daher restriktiver sein, als dies für manche Anwendungen wünschenswert ist.

[0058] void removeArbitraryLink(T* oldLink): Diese Funktion entfernt die angegebene frei wählbare Verknüpfung aus der Liste.

[0059] T* removeBeforeArbitraryLink(T* refLink): Diese Funktion entfernt die Verknüpfung aus der Liste, die der angegebenen frei wählbaren Verknüpfung unmittelbar vorangeht, und gibt einen Zeiger auf sie zurück. Sie erfordert ebenfalls eine atomare Sicht der Liste.

[0060] T* removeAfterArbitraryLink(T* refLink): Diese Funktion entfernt die Verknüpfung aus der Liste, die der angegebenen frei wählbaren Verknüpfung unmittelbar folgt, und gibt einen Zeiger auf sie zurück.

[0061] getLinkCount() const: Diese Funktion gibt eine Momentaufnahme des Zählwerts der Verknüpfungen in der Liste zurück. Der Wert wird ohne wie auch immer geartete Synchronisierung zurückgegeben.

[0062] Die folgenden Funktionen beziehen sich auf eine konkrete Verknüpfung, d.h., sie gegen davon aus, dass der Client bereits über einen Verweis auf eine Verknüpfung, jedoch nicht notwendigerweise auf das SharedList<T>-Objekt **301** verfügt:
LinkStatus getLinkStatus(const T* refLink) const: Diese Funktion gibt den Status der angegebenen Verknüpfung zurück. Wenn für die Verknüpfungen das Feld definiert wurde, das auf das SharedList<T>-Objekt zeigt, in dem sie enthalten sind, gibt diese Funktion an, ob sich die angegebene Verknüpfung in der Liste befindet. Wenn das Zeigerfeld für die Verknüpfungen nicht definiert wurde, gibt diese Funktion lediglich an, dass ihr nicht bekannt ist, ob sich die Verknüpfung in der Liste befindet.

[0063] static LinkStatus getLinkStatus(const T* refLink, int32 Offset) const: Dies ist eine statische Funktion, die den Status der angegebenen Verknüpfung zurückgibt. Wenn der angegebene Versatz negativ ist, gibt die Funktion eine Meldung zurück, dass ihr nicht bekannt ist, ob sich die Verknüpfung in einem SharedList<T>-Objekt befindet. Wenn der angegebene Versatz nicht negativ ist, gibt die Funktion eine Meldung zurück, die angibt, ob sich die Verknüpfung in einem SharedList<T>-Objekt befindet.

[0064] static SharedList<T>* getSharedList(const T* refLink, int32 Offset): Dies ist eine statische Funktion, die einen Zeiger auf das SharedList<T>-Objekt zurückgibt, das die angegebene Verknüpfung enthält. Wenn der angegebene Versatz negativ oder wenn die Verknüpfung nicht in einem SharedList<T>-Objekt enthalten ist, wird der NULL-Zeiger zurückgegeben.

[0065] static SharedList<T>* removeArbitraryLink(T* refLink, int32 Offset): Dies ist eine statische Funktion, welche die angegebene Verknüpfung aus einem beliebigen SharedList<T>-Objekt entfernt, in dem sie sich befindet, und einen Zeiger auf das SharedList<T>-Objekt zurückgibt. Wenn der angegebene Versatz nicht negativ ist oder wenn die Verknüpfung nicht in einem SharedList<T>-Objekt enthalten war, wird der NULL-Zeiger zurückgegeben (und die Verknüpfung wird nicht entfernt).

[0066] Alle oben genannten Funktionen, die Verknüpfungen einfügen oder entfernen, erzeugen verschiedene Arten von Iterator-Objekten, welche die Funktionen dann tatsächlich ausführen. Clients können anhand dieser (hier beschriebenen) Iterator-Objekte diese und auch andere Funktionen ausführen. Neben den oben beschriebenen Funktionen können verschiedene andere zweitrangige Funktionen für spezielle Anwendungen definiert werden, die hier nicht beschrieben werden. Dabei dürfte klar sein, dass die obige Liste von Funktionen lediglich als eine einzige Ausführungsform beschrieben wird und dass gemäß verschiedenen Ausführungsformen der vorliegenden Erfindung auch andere oder zusätzliche Funktionen definiert werden könnten.

SharedListIterator<T>-Objekte

[0067] SharedListIterator<T>-Objekte **401** ermöglichen es, die Verknüpfungen eines SharedList<T>-Objekts **301** zu durchlaufen und Verknüpfungen in die Liste aufzunehmen bzw. daraus zu entfernen. Es gibt verschiedene Arten von SharedListIterator<T>-Objekten, und diese werden in einer Vererbungshierarchie angeordnet. Es gibt Iteratoren, welche die Liste lediglich überprüfen können, indem sie sie durchlaufen, und Iteratoren, die sie nicht nur durchlaufen, sondern auch ändern (mutieren) können (indem sie Verknüpfungen einfügen oder entfernen); es gibt Iteratoren, die über eine nichtatomare Sicht der Liste verfügen, und Iteratoren, die über eine atomare Sicht verfügen. Um mit der Liste arbeiten zu können, erzeugt ein Client einen Iterator des gewünschten Typs und ruft anschließend die von diesem bereitgestellten Funktionen auf.

[0068] In diesem Zusammenhang bedeutet atomare Sicht, dass der Iterator sicher sein kann, dass keine anderen Mutatoren zeitgleich versuchen, die Liste zu ändern, d.h., der Iterator hat eine Art von exklusivem Zugriff, obwohl er Inspektoren den gleichzeitigen Zugriff auf die Liste gewähren kann. Eine nichtatomare Sicht bedeutet, dass es keine solche Gewissheit gibt. Die Ausführungsformen der vorliegenden Erfindung zielen darauf ab, gleichzeitige Änderungen an verschiedenen Teilen einer verknüpften Liste zu gestatten, weshalb der nichtato-

maren Sicht der Vorteil zu geben ist. Allerdings kann unter besonderen Umständen eine gewisse Art von Exklusivität notwendig sein, weshalb eine atomare Sicht bereitgestellt wird, die ähnliche Auswirkungen hat wie die Sperrverfahren nach dem Stand der Technik.

Iterator-Zustände

[0069] Ein Iterator kann die folgenden Zustände aufweisen:
Inaktiv: Dies ist der Ausgangszustand eines Interators, wenn er erzeugt wird. In diesem Zustand enthält er keine Listenressourcen und verfügt über keine Informationen zu einer aktuellen Verknüpfung. Ein Iterator muss in diesen Zustand zurückkehren, wenn er die Liste abgeschlossen hat (oder er wird implizit wieder in diesen Zustand zurückversetzt, wenn er gelöscht wird).

[0070] Aktiv: Dies ist der Zustand eines Iterators, wenn er Listenressourcen hält und über Informationen zu einer aktuellen Verknüpfung verfügt. Ein Iterator erreicht diesen Zustand in der Regel nur dann, wenn die Methode getFirstLink() (oder getLinkAfterArbitraryLink()) ausgeführt wurde. Wie weiter unten beschrieben, gibt es jedoch einige Mutatorfunktionen, die einen Iterator in den aktiven Zustand versetzen können.

[0071] Übergang: Dieser Zustand bezieht sich ausschließlich auf Mutatoren. In diesem Zustand verfügt ein Mutator nicht mehr über Informationen zu einer aktuellen Verknüpfung, hält jedoch noch Listenressourcen. Aus diesem Grund sollte ein Mutator nicht „sich selbst überlassen bleiben", wenn er diesen Zustand erreicht. Vorgesehen ist vielmehr, eine Funktion insertBeforeCurrentLink() zu ermöglichen, um einen erfolgreichen Abschluss am Ende einer Liste zu gewährleisten.

Objekthierarchie

[0072] **Fig.** 4 zeigt die von außen betrachtete Hierarchie der Objektklassenvererbung und die Funktionen, die von jeder Klasse unterstützt werden, welche von der SharedListrterator<T>-Objektklasse **401** erbt. Diese Objektklassen werden wie folgt beschrieben:
SharedListIterator<T> **401**: Dies ist die höchste Klasse in der Vererbungshierarchie, wobei es sich lediglich um eine abstrakte Basisklasse handelt. Sie ermöglicht es, die Liste in Vorwärtsrichtung zu durchlaufen und jeweils auf eine Verknüpfung zu verweisen. Ob sie die Liste ändern oder nur überprüfen kann, wird von abgeleiteten Klassen bestimmt. Sie verfügt entweder über eine atomare oder eine nichtatomare Sicht der Liste, die ebenfalls von abgeleiteten Klassen bereitgestellt wird.

[0073] SharedListInspector<T> **402**: Diese Klasse leitet sich aus SharedListIterator<T> **401** ab. Es handelt sich ebenfalls um eine abstrakte Klasse. Sie ist nicht mehr als ein Iterator, sichert jedoch zu, die Liste unter keinen Umständen zu ändern. Sie verfügt über eine atomare oder nichtatomare Sicht der Liste, wobei dies von abgeleiteten Klassen bereitgestellt wird.

[0074] SharedListAtomicInspector<T> **404**: Dies ist eine konkrete Klasse, die sich aus SharedListInspector<T> **402** ableitet. Objekte dieser Klasse verfügen über eine atomare Sicht der Liste, was heißt, dass sie sicher sein können, dass kein Mutator gleichzeitig die Liste verarbeitet. Objekte dieser Klasse können die Liste (entweder vorwärts oder rückwärts) durchlaufen, jedoch nicht ändern.

[0075] SharedListNonatomicInspector<T> **405**: Dies ist eine konkrete Klasse, die sich aus SharedListInspector<T> **402** ableitet. Objekte dieser Klasse verfügen über eine nichtatomare Sicht der Liste, was heißt, dass Mutatoren die Liste gleichzeitig verarbeiten können. Objekte dieser Klasse können die Liste in Vorwärtsrichtung durchlaufen, jedoch nicht ändern.

[0076] SharedListMutator<T> **403**: Diese Klasse leitet sich aus SharedListIterator<T> **401** ab. Sie ist ebenfalls eine abstrakte Klasse, ist jedoch in der Lage, die Liste zu ändern, indem sie Verknüpfungen in sie einfügt oder aus ihr entfernt. Sie verfügt entweder über eine atomare oder eine nichtatomare Sicht der Liste, die von abgeleiteten Klassen bereitgestellt wird.

[0077] SharedListAtomicMutator<T> **407**: Dies ist eine konkrete Klasse, die sich aus SharedListMutator<T> **403** ableitet. Objekte dieser Klasse verfügen über eine atomare Sicht der Liste, d.h. sie können sicher sein, dass kein anderer Mutator die Liste ebenfalls verarbeitet. Objekte dieser Klasse können die Liste entweder vorwärts oder rückwärts durchlaufen und sie ändern.

[0078] SharedListNonatomaticMutatorT> **406**: Dies ist eine konkrete Klasse, die sich aus SharedListMuta-

tor<T> **403** ableitet. Objekte dieser Klasse verfügen über eine nichtatomare Sicht der Liste, was bedeutet, dass andere nichtatomare Mutatoren sie zur gleichen Zeit verarbeiten können. Objekte dieser Klasse können die Liste in Vorwärtsrichtung durchlaufen und Verknüpfungen in sie einfügen bzw. aus ihr entfernen.

SharedListIterator<T>-Objekte

[0079] Iteratoren sind in der Lage, die Verknüpfungen in der Liste zu durchlaufen. Im aktiven oder Übergangszustand verfügen sie über eine aktuelle Verknüpfung, bei der es sich um die Verknüpfung handelt, auf die sie zu diesem Zeitpunkt verweisen. Gegenüber einem Inspektor wird sichergestellt, dass die aktuelle Verknüpfung nicht aus der Liste entfernt werden kann. Gegenüber einem Mutator wird sichergestellt, dass kein anderer Mutator seine aktuelle Verknüpfung oder die ihr vorausgehende Verknüpfung entfernen bzw. keine Verknüpfung zwischen sie einfügen kann.

[0080] Mit Blick auf die abstrakte Basisklasse werden für alle Iteratoren aller Typen die folgenden Funktionen bereitgestellt:
T* getFirstLink(): Diese Funktion setzt die aktuelle Verknüpfung des Iterators auf die erste Verknüpfung in der Liste und gibt einen Zeiger auf sie zurück. Der Iterator befindet sich infolge des Aufrufens dieser Funktion im aktiven Zustand. Wenn jedoch keine Verknüpfungen in der Liste vorhanden sind, hat der zurückgegebene Zeiger den Wert NULL, und der Iterator befindet sich im inaktiven Zustand, wenn es sich um einen Inspektor handelt, bzw. im Übergangszustand, wenn es sich um einen Mutator handelt.

[0081] T* getLinkAfterArbitraryLink(T* refLink): Diese Funktion setzt die aktuelle Verknüpfung des Iterators auf die Verknüpfung unmittelbar nach der angegebenen frei wählbaren Verknüpfung und gibt einen Zeiger darauf zurück. Der Iterator befindet sich infolge des Aufrufs dieser Funktion im aktiven Zustand. Wenn jedoch keine Verknüpfung nach der angegebenen frei wählbaren Verknüpfung vorhanden ist, hat der zurückgegebene Zeiger den Wert NULL, und der Iterator befindet sich im inaktiven Zustand, wenn es sich um einen Inspektor handelt, bzw. im Übergangszustand, wenn es sich um einen Mutator handelt.

[0082] T* getNextLink(): Mit dieser Funktion wechselt der Iterator zur nächsten Verknüpfung in der Liste. Sie setzt die aktuelle Verknüpfung auf die nächste Verknüpfung in der Liste und gibt einen Zeiger auf sie zurück. Die tatsächlichen Ergebnisse sind jedoch abhängig vom Zustand des Iterators vor dem Aufrufen der Funktion. Wenn sich der Iterator im aktiven Zustand befindet und das Ende der Liste nicht erreicht ist, bleibt er im aktiven Zustand. Wenn sich der Iterator im aktiven Zustand befindet und das Ende der Liste erreicht wird, wechselt er in den inaktiven Zustand, wenn es sich um einen Inspektor handelt, bzw. in den Übergangszustand, wenn es sich um einen Mutator handelt. In beiden Fällen wird der NULL-Zeiger zurückgegeben. Wenn sich der Iterator im Übergangszustand befindet, wechselt er in den inaktiven Zustand, und der NULL-Zeiger wird zurückgegeben. Wenn sich der Iterator im inaktiven Zustand befindet, bleibt er in diesem Zustand, und der NULL-Zeiger wird zurückgegeben.

[0083] T* getCurrentLink(): Diese Funktion gibt einen Zeiger auf die aktuelle Verknüpfung des Iterators zurück. Im Allgemeinen gibt diese Funktion einfach den gleichen Wert zurück wie der letzte Aufruf einer beliebigen anderen „getLink"-Funktion. Es gibt jedoch einige Mutatorfunktionen, die eine Änderung der aktuellen Verknüpfung des Mutators verursachen können, und wenn das Ergebnis einer derartigen Funktion angibt, dass die aktuelle Verknüpfung des Mutators geändert wurde, sollte diese Funktion aufgerufen werden, um die neue aktuelle Verknüpfung zu erhalten.

[0084] void rest(): Mit dieser Funktion wechselt der Iterator in den inaktiven Zustand. Sie wird automatisch aufgerufen, wenn der Destruktor des Iterators ausgeführt wird, kann jedoch auch durch Clients aufgerufen werden, um beispielsweise den Übergangszustand eines Mutators zu beenden.

[0085] IteratorState getState() const: Diese Funktion gibt den aktuellen Zustand des Iterators zurück (aktiv, inaktiv oder Übergang).

[0086] getLinkCount() const: Diese Funktion gibt eine Momentaufnahme der Anzahl von Verknüpfungen in der Liste zurück.

[0087] Konstruktoren: Die Konstruktoren für alle konkreten abgeleiteten Klassen benötigen einen Verweis auf das SharedList<T>-Objekt **301**, dem der Iterator zugehörig ist. Zusätzlich unterstützen die Konstruktoren als optionalen Parameter eine Angabe zum Umfang der Iterator-Sperre für das SharedList<T>-Objekt. Die Sperre kann entweder an das Vorhandensein des SharedList<T>-Objekts gebunden oder aber dynamisch sein, wobei

sie dann nur aktiviert wird, wenn sich der Iterator im aktiven oder Übergangszustand befindet oder (falls es sich um einen Mutator handelt) während er eine Änderungsfunktion ausführt. Bei der dynamischen Option können Iteratoren vorhanden sein, ohne dass sie über eine Sperre für das SharedList<T>-Objekt verfügen, „solange sie nicht für eine andere Aufgabe benötigt werden". Dies kann nützlich sein, wenn der Zusatzaufwand für das wiederholte Erzeugen und Löschen des Iterators andernfalls zu hoch wäre. Standardmäßig ist die Sperre an die Lebensdauer des Iterators gebunden.

[0088] Destruktoren: Die Destruktoren aller konkreten, abgeleiteten Klassen stellen sicher, dass der Iterator wieder in den inaktiven Zustand versetzt wird.

SharedListInspector<T>-Objekte

[0089] Diese abstrakte Klasse stellt keine zusätzlichen Funktionen bereit, die über diejenigen der SharedListIterator<T>-Basisklasse hinausgehen. Objekte von Klassen, die aus dieser Klasse abgeleitet werden, können die Liste überprüfen, aber nicht ändern.

SharedListAtomicInspector<T>-Objekte

[0090] Dies ist eine konkrete Klasse, die aus SharedListInspector<T> abgeleitet wird. Objekte dieser Klasse verfügen über eine atomare Sicht der Liste. Objekte dieser Klasse können die Liste sowohl vorwärts als auch rückwärts durchlaufen, sie jedoch nicht ändern.

[0091] Aufgrund der atomaren Sicht stellt diese Klasse außerdem die folgenden Funktionen bereit:
T* getLastLink(): Diese Funktion gibt einen Zeiger auf die letzte Verknüpfung in der Liste zurück. Nach dem Aufrufen dieser Funktion befindet sich der Mutator im aktiven Zustand, es sei denn, die Liste enthält keine Verknüpfungen. In diesem Fall wird der NULL-Zeiger zurückgegeben, und der Mutator befindet sich im inaktiven Zustand.

[0092] T* getPrevLink(): Diese Funktion führt eine „Sicherung" des Iterators auf die vorherige Verknüpfung in der Liste durch. Sie setzt die aktuelle Verknüpfung auf die vorherige Verknüpfung in der Liste und gibt einen Zeiger darauf zurück. Die tatsächlichen Ergebnisse sind jedoch abhängig vom Zustand des Inspektors vor dem Aufrufen der Funktion. Wenn sich der Inspektor im aktiven Zustand befindet und der Anfang der Liste nicht erreicht wird, bleibt er im aktiven Zustand. Wenn sich der Inspektor im aktiven Zustand befindet und der Anfang der Liste erreicht wird, wechselt er in den inaktiven Zustand, und der NULL-Zeiger wird zurückgegeben. Wenn sich der Inspektor im inaktiven Zustand befindet, bleibt er in diesem Zustand, und der NULL-Zeiger wird zurückgegeben.

SharedListeNonatomicInspector<T>-Objekte

[0093] Dies ist eine konkrete Klasse, die aus SharedListInspector<T> **402** abgeleitet wird. Sie stellt keine zusätzlichen Funktionen zur Basisklasse bereit. Objekte dieser Klasse verfügen über eine nichtatomare Sicht der Liste. Sie können die Liste in Vorwärtsrichtung durchlaufen, sie jedoch nicht selbst ändern.

SharedListMutator<T>-Objekte

[0094] Diese abstrakte Klasse wird aus SharedListIterator<T> **401** abgeleitet. Sie fügt die Möglichkeit des Änderns der Liste hinzu, indem Verknüpfungen zu ihr hinzugefügt bzw. aus ihr entfernt werden. Sie verfügt entweder über eine atomare oder eine nichtatomare Sicht der Liste, die von den aus ihr abgeleiteten Klassen bereitgestellt wird.

[0095] Der Mutator eines Client kann sich in einem beliebigen Zustand befinden, wenn er eine Änderungsfunktion aufruft. Während der eigentlichen Ausführung der Funktion muss sich der Mutator im Allgemeinen (zumindest intern) im inaktiven Zustand befinden. Damit ein Mutator im aktiven oder Übergangszustand während einer Änderungsfunktion „seine Position in der Liste behält", muss nach Abschluss der Funktion der Zustand wiederhergestellt werden können. Allgemeiner ausgedrückt, alle Änderungsfunktionen beinhalten einen Parameter MutatorControl, der angibt, wie die aktuelle Verknüpfung des Mutators (und damit sein Zustand) gehandhabt werden soll, wenn die Funktion ausgeführt wird. Dabei können die folgenden Werte festgelegt werden:
conditionalUpdate = 0: Dieser Wert gibt an, dass die aktuelle Verknüpfung des Mutators nur dann geändert werden soll, wenn dies notwendig ist, um die Funktion auszuführen. Wenn die aktuelle Verknüpfung geändert wird, „verliert der Mutator nicht seine Position in der Liste"; die neue aktuelle Verknüpfung ist diejenige Ver-

knüpfung, die nun auf diejenige folgt, die vor der alten aktuellen Verknüpfung stand. Die aktuelle Verknüpfung eines atomaren Mutators wird nur dann geändert, wenn sie zufälligerweise die entfernte Verknüpfung ist. Die aktuelle Verknüpfung eines nichtatomaren Mutators wird von allen Änderungsfunktionen mit Ausnahme von insertAfterCurrentLink() und removeAfterCurrentLink() geändert.

[0096] unconditionalUpdate = 1: Dieser Wert gibt an, dass die aktuelle Verknüpfung des Mutators geändert werden soll, wenn die Funktion erfolgreich ausgeführt wurde. Wenn eine Verknüpfung eingefügt wird, ist die neue aktuelle Verknüpfung die neu eingefügte Verknüpfung. Wenn eine Verknüpfung entfernt wird, ist die neue aktuelle Verknüpfung diejenige Verknüpfung, die nun auf die entfernte Verknüpfung folgt.

[0097] unconditionalReset = 2: Dieser Wert gibt an, dass der Mutator wieder in den inaktiven Zustand versetzt werden soll, wenn die Funktion erfolgreich ausgeführt wurde.

[0098] Die verschiedenen Einfüge- und Entfernungsfunktionen geben einen Wert MutatorResult zurück, der angibt, ob die Funktion erfolgreich ausgeführt wurde und ob die aktuelle Verknüpfung geändert wurde. Dabei können die folgenden Werte zurückgegeben werden:
successfulAsRequested = 0: Dieser Wert gibt an, dass die angeforderte Funktion erfolgreich ausgeführt wurde und dass entweder die aktuelle Verknüpfung des Mutators nicht geändert wurde oder dass sie nichtbedingt aktualisiert wurde oder dass der Mutator nichtbedingt zurückgesetzt wurde, wie dies durch den Wert für MutatorControl angegeben wird.
failedNoChange = 1: Dieser Wert gibt an, dass die angeforderte Funktion fehlgeschlagen ist und dass die aktuelle Verknüpfung des Mutators nicht geändert wurde. Diese Bedingung kann das Ergebnis eines Benutzerfehlers sein.
successfulWithChange = 2: Dieser Wert gibt an, dass die angeforderte Funktion erfolgreich ausgeführt wurde, die aktuelle Verknüpfung des Mutators jedoch nicht geändert wurde. Dieser Wert wird nur zurückgegeben, wenn als Wert für MutatorControl conditionalUpdate angegeben wird. In diesem Fall wird er für einen nichtatomaren Mutator, der sich im aktiven oder Übergangszustand befunden hat, oder für einen atomaren Mutator zurückgegeben, wodurch eine Verknüpfung entfernt wird, bei der es sich um seine aktuelle Verknüpfung gehandelt hat.
failedWithChange = 3: Dieser Wert gibt an, dass die angeforderte Funktion fehlgeschlagen ist und dass die aktuelle Verknüpfung des Mutators geändert werden musste, um eine gegenseitige Sperre zu vermeiden. Dieser Fall kann ausschließlich für einen nichtatomaren Mutator eintreten und unabhängig vom angegebenen Wert für MutatorControl geschehen. Allerdings „verliert der Mutator nicht seine Position in der Liste". Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die vor der alten aktuellen Verknüpfung stand.

[0099] Clients können den zurückgegebenen Wert für MutatorResult mit den oben angegebenen Werten vergleichen oder sie als Maske verwenden und die einzelnen Bits innerhalb der Maske überprüfen. Dabei haben die einzelnen Bits folgende Bedeutung:
failed = 0×01: Dieses Bit ist aktiv, wenn die Funktion fehlgeschlagen ist, und inaktiv, wenn sie erfolgreich ausgeführt wurde.
changed = 0×02: Dieses Bit ist aktiv, wenn die aktuelle Verknüpfung des Mutators „unerwartet" geändert wurde. Wenn es inaktiv ist, gibt es an, dass die Funktion entweder fehlgeschlagen ist und die aktuelle Verknüpfung des Mutators nicht geändert wurde oder dass die Funktion zwar erfolgreich ausgeführt werden konnte, die aktuelle Verknüpfung des Mutators jedoch nur geändert wurde, wenn der Wert für MutatorControl ein nichtbedingtes Aktualisieren oder Zurücksetzen angefordert hat.

[0100] Die SharedListMutator<T>-Klasse **403** stellt die im Folgenden genannten Funktionen bereit. Dabei erfordert jede Funktion einen Parameter für MutatorControl und gibt entweder einen Wert oder Parameter für MutatorControl zurück bzw. aktualisiert diesen.

[0101] T* removeFirstLink(MutatorControl, MutatorResult&): Diese Methode entfernt die erste Verknüpfung aus der Liste und gibt einen Zeiger darauf zurück. Wenn keine Verknüpfungen in der Liste vorhanden sind, wird NULL zurückgegeben. Der Parameter für MutatorResult wird aktualisiert, um anzuzeigen, ob die Funktion erfolgreich durchgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 1.

[0102] T* removeAfterArbitraryLInk(T* refLink, MutatorControl, MutatorResult&): Diese Methode entfernt die Verknüpfung aus der Liste, die auf die angegebene frei wählbare Verknüpfung folgt, und gibt einen Zeiger dar-

auf zurück. Wenn die angegebene frei wählbare Verknüpfung die letzte Verknüpfung in der Liste war, wird NULL zurückgegeben. Der Parameter für MutatorResult wird aktualisiert, um anzuzeigen, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 1.

Tabelle 1: MutatorResult-Werte für ausgewählte Entfernungsfunktionen

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die entfernte Verknüpfung war nicht die aktuelle Verknüpfung. Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL, oder es war keine zu entfernende Verknüpfung vorhanden. 2 = successfulWithChange Die entfernte Verknüpfung war die alte aktuelle Verknüpfung. Die aktuelle Verknüpfung wurde geändert und ist nun die Verknüpfung, die auf sie folgte.	0 = successfulAsRequested Der Mutator befand sich im inaktiven Zustand und verbleibt darin. 1 = failedWithoutChange (a) Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL, oder (b) der Mutator befand sich im inaktiven Zustand, und es war keine zu entfernende Verknüpfung vorhanden. 2 = sucessfulWithChange Der Mutator befand sich im aktiven oder Übergangszustand. Die aktuelle Verknüpfung wurde geändert und ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte. 3 = failedWithChange Der Mutator befand sich im aktiven oder Übergangszustand, und es war keine zu entfernende Verknüpfung vorhanden. Der Mutator befindet sich nun im Übergangszustand.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL, oder es war keine zu entfernende Verknüpfung vorhanden.	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte, als diese entfernt wurde. 1 = failedWithoutChange (a) Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL, oder (b) der Mutator befand sich im inaktiven Zustand und es war keine zu entfernende Verknüpfung vorhanden. 3 = failedWithChange Der Mutator befand sich im aktiven oder Übergangszustand, und es war keine zu entfernende Verknüpfung vorhanden. Der Mutator befindet sich nun im Übergangszustand.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL, oder es war keine zu entfernende Verknüpfung vorhanden.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange (a) Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL, oder (b) der Mutator befand sich im inaktiven Zustand und es war keine zu entfernende Verknüpfung vorhanden. 3 = failedWithChange Der Mutator befand sich im inaktiven oder Übergangszustand, und es war keine zu entfernende Verknüpfung vorhanden. Der Mutator befindet sich nun im Übergangszustand.
Der Teil dieser Tabelle, der sich auf den atomaren Mutator bezieht, gilt auch für die Funktionen removeLastLink() und removeBeforeArbitraryLink().		

[0103] MutatorResult removeArbitraryLink(T* oldLink, MutatorControl): Diese Methode entfernt die angegebene frei wählbare Verknüpfung aus der Liste und gibt ein Ergebnis zurück, das angibt, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 2.

Tabelle 2: MutatorResult-Werte für die Funktion removeArbitraryLink()

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die entfernte Verknüpfung war nicht die aktuelle Verknüpfung. Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL. 2 = successfulWithChange Die entfernte Verknüpfung war die alte aktuelle Verknüpfung. Die aktuelle Verknüpfung wurde geändert und ist nun die Verknüpfung, die auf sie folgte.	0 = successfulAsRequested Der Mutator befand sich im inaktiven Zustand und verbleibt darin. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL. 2 = sucessfulWithChange Der Mutator befand sich im aktiven oder Übergangszustand. Die aktuelle Verknüpfung wurde geändert und ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die entfernte Verknüpfung folgte, als diese entfernt wurde. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Die frei wählbare Verknüpfung, auf die verwiesen wurde, war NULL.

[0104] MutatorResult removeCurrentLink(MutatorControl): Diese Methode versucht, die aktuelle Verknüpfung des Mutators aus der Liste zu entfernen. Die Funktion kann unter Umständen fehlschlagen. Ein Fehlschlag liegt daran, dass sich ein anderer Mutator entweder dazu verpflichtet hat, eine Verknüpfung vor der aktuellen Verknüpfung des Mutators einzufügen oder die aktuelle Verknüpfung dieses Mutators zu entfernen. Unabhängig davon, ob die Funktion erfolgreich ausgeführt wird, verfügt der Mutator über eine neue aktuelle Verknüpfung. Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der aktuellen Verknüpfung zum Zeitpunkt der versuchten Entfernung voranging. Auf diese Weise „verliert der Mutator nicht seine Position in der Liste". Die Funktion gibt ein Ergebnis zurück, das anzeigt, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 3.

Tabelle 3: MutatorResult-Werte für die Funktion removeCurrentLink()

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	1 = failedwithoutChange Der Mutator befand sich nicht im aktiven Zustand. 2 = successfulWithChange Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die aktuelle Verknüpfung folgte, als diese entfernt wurde.	1 = failedWithoutChange Der Mutator befand sich nicht im aktiven Zustand. 2 = sucessfulWithChange Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die aktuelle Verknüpfung folgte, als diese entfernt wurde. 3 = failedWithChange Die aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die aktuelle Verknüpfung folgte, als diese entfernt wurde. 1 = failedWithoutChange Der Mutator befand sich nicht im aktiven Zustand.	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist nun die Verknüpfung, die auf die aktuelle Verknüpfung folgte, als diese entfernt wurde. 1 = failedwithoutChange Der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Der Mutator befand sich nicht im aktiven Zustand.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.

[0105] T* removeAfterCurrentLink(MutatorControl, MutatorResult&): Diese Funktion versucht, die Verknüpfung unmittelbar nach der aktuellen Verknüpfung zu entfernen, und gibt einen Zeiger auf sie zurück. Wenn die aktuelle Verknüpfung die letzte Verknüpfung in der Liste ist, wird NULL zurückgegeben. Die Funktion kann unter Umständen fehlschlagen. Ein Fehlschlag liegt daran, dass sich ein anderer Mutator dazu verpflichtet hat, die aktuelle Verknüpfung dieses Mutators zu entfernen. In diesem Fall verfügt dieser Mutator über eine neue aktuelle Verknüpfung, bei der es sich um diejenige handelt, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Entfernung voranging. Auf diese Weise „verliert der Mutator nicht seine Position in der Liste". Der Wert für MutatorResult wird aktualisiert um anzuzeigen, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 4.

Tabelle 4: MutatorResult-Werte für die Funktion removeAfterCurrentLink()

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder Mutator befand sich im inaktiven Zustand.	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist die Verknüpfung, die auf die entfernte Verknüpfung folgte. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Die aktuelle Verknüpfung ist die Verknüpfung, die auf die entfernte Verknüpfung folgte. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Die aktuelle Verknüpfung war entweder die letzte Verknüpfung, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung voranging, als versucht wurde, diese zu entfernen.

[0106] MutatorResult insertFirst (T* newLink, MutatorControl): Diese Funktion fügt die angegebene Verknüpfung am Anfang der Liste ein und gibt ein Ergebnis zurück, das die Auswirkungen auf die aktuelle Verknüpfung des Mutators angibt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 5.

[0107] MutatorResult insertLast(T* newLink, MutatorControl): Diese Funktion fügt die angegebene Verknüpfung am Ende der Liste ein und gibt ein Ergebnis zurück, das die Auswirkungen auf die aktuelle Verknüpfung des Mutators angibt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 5.

[0108] MutatorResult insertBeforeArbitraryLink(T* newLink, T* refLink, MutatorControl): Diese Funktion fügt die angegebene Verknüpfung unmittelbar vor der angegebenen Verweisverknüpfung ein und gibt ein Ergebnis zurück, das die Auswirkungen auf die aktuelle Verknüpfung des Mutators angibt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 5.

[0109] MutatorResult insertAfterArbitraryLink(T* newLink, T* refLink, MutatorControl): Diese Funktion fügt die angegebene Verknüpfung unmittelbar nach der angegebenen Verweisverknüpfung ein und gibt ein Ergebnis zurück, das die Auswirkungen auf die aktuelle Verknüpfung des Mutators angibt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 5.

Tabelle 5: MutatorResult-Werte für ausgewählte Einfügefunktionen

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.	0 = successfulAsRequested Der Mutator befand sich im inaktiven Zustand und verbleibt darin. 1 = failedwithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL. 2 = sucessfulWithChange Der Mutator befand sich im aktiven oder Übergangszustand. Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung oder die frei wählbare Verknüpfung, auf die verwiesen wird, war NULL.

[0110] MutatorResult insertBeforeCurrexitLink (T* newLink, MutatorControl): Diese Methode versucht, die angegebene Verknüpfung unmittelbar vor der aktuellen Verknüpfung des Mutators einzufügen. Die Funktion kann unter Umständen fehlschlagen. Ein Fehlschlag liegt daran, dass sich ein anderer Mutator entweder dazu verpflichtet hat, eine Verknüpfung vor der aktuellen Verknüpfung dieses Mutators einzufügen oder die aktuelle Verknüpfung dieses Mutators zu entfernen. Unabhängig davon, ob die Funktion erfolgreich ausgeführt wird, verfügt der Mutator über eine neue aktuelle Verknüpfung. Die neue aktuelle Verknüpfung ist diejenige, die auf die Verknüpfung folgt, die der aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging. Auf diese Weise „verliert der Mutator nicht seine Position in der Liste". Der Rückgabewert gibt an, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 6.

Tabelle 6: MutatorResults-Werte für die Funktion insertBeforeCurrentLink()

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.

[0111] MutatorResult insertAfterCurrentLink (T* newLink, MutatorControl): Diese Funktion versucht, die angegebene Verknüpfung unmittelbar nach der aktuellen Verknüpfung des Mutators einzufügen. Die Funktion kann unter Umständen fehlschlagen. Ein Fehlschlag liegt daran, dass sich ein anderer Mutator bereits dazu verpflichtet hat, die aktuelle Verknüpfung dieses Mutators zu entfernen. In diesem Fall verfügt der Mutator über eine neue aktuelle Verknüpfung, bei der es sich um diejenige handelt, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging. Auf diese Weise „verliert der Mutator nicht seine Position in der Liste". Der Rückgabewert gibt an, ob die Funktion erfolgreich ausgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt.

[0112] Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 7.

Tabelle 7: MutatorResults-Werte für die Funktion insertAfterCurrentLink()

MutatorControl	Atomarer Mutator	Nichtatomarer Mutator
0 = Bedingte Aktualisierung	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = successfulAsRequested Die aktuelle Verknüpfung bleibt unverändert. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.
1 = Unbedingte Aktualisierung	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Die neu eingefügte Verknüpfung ist nun die aktuelle Verknüpfung. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.
2 = Unbedingtes Zurücksetzen	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich im inaktiven Zustand.	0 = sucessfulAsRequested Der Mutator wurde in den inaktiven Zustand zurückversetzt. 1 = failedWithoutChange Entweder die einzufügende Verknüpfung war NULL, oder der Mutator befand sich nicht im aktiven Zustand. 3 = failedWithChange Die neue aktuelle Verknüpfung ist diejenige, die nun auf die Verknüpfung folgt, die der alten aktuellen Verknüpfung zum Zeitpunkt der versuchten Einfügung voranging.

SharedListAtomicMutator<T>-Objekte

[0113] Dies ist eine konkrete Klasse, die sich aus SharedListMutator<T> **403** ableitet. Objekte dieser Klasse verfügen über eine atomare Sicht der Liste, was bedeutet, dass sie sicher sein können, dass kein anderer Mutator die Liste verarbeitet. Objekte dieser Klasse können die Liste vorwärts oder rückwärts durchlaufen und Verknüpfungen in sie einfügen bzw. daraus entfernen.

[0114] Aufgrund der atomaren Sicht stellt diese Klasse außerdem die folgenden Funktionen bereit:
T* getLastLink(): Diese Funktion gibt einen Zeiger auf die letzte Verknüpfung in der Liste zurück. Nach dem Aufrufen dieser Funktion befindet sich der Mutator im aktiven Zustand, es sei denn, die Liste enthält keine Verknüpfungen. In diesem Fall wird der NULL-Zeiger zurückgegeben, und der Mutator befindet sich im Übergangszustand.

[0115] T* getPrevLink(): Diese Funktion führt eine „Sicherung" des Mutators auf die vorherige Verknüpfung in der Liste durch. Sie setzt seine aktuelle Verknüpfung auf die vorherige Verknüpfung in der Liste und gibt einen

Zeiger darauf zurück. Die tatsächlichen Ergebnisse sind jedoch abhängig vom Zustand des Mutators vor dem Aufrufen der Funktion. Wenn sich der Mutator im aktiven Zustand befindet und der Anfang der Liste nicht erreicht wird, bleibt er im aktiven Zustand. Wenn sich der Mutator im Übergangszustand befindet und der Anfang der Liste nicht erreicht wird, wechselt er in den aktiven Zustand. Wenn sich der Mutator im aktiven oder Übergangszustand befindet und der Anfang der Liste erreicht wird, wechselt er in den inaktiven Zustand, und der NULL-Zeiger wird zurückgegeben. Wenn sich der Mutator im inaktiven Zustand befindet, bleibt er in diesem Zustand, und der NULL-Zeiger wird zurückgegeben.

[0116] T* removeLastLink(MutatorControl, MutatorResult&): Diese Methode entfernt die letzte Verknüpfung aus der Liste und gibt einen Zeiger darauf zurück. Wenn keine Verknüpfungen in der Liste vorhanden waren, wird NULL zurückgegeben. Der Parameter für MutatorResult wird aktualisiert, um anzuzeigen, ob die Funktion erfolgreich durchgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 1.

[0117] T* removeBeforeArbitraryLink(T* refLink, MutatorControl, MutatorResult&): Diese Funktion entfernt die Verknüpfung aus der Liste, die der angegebenen frei wählbaren Verknüpfung unmittelbar vorangeht, und gibt einen Zeiger auf sie zurück. Wenn die angegebene frei wählbare Verknüpfung die erste Verknüpfung in der Liste war, wird NULL zurückgegeben. Der Parameter für MutatorResult wird aktualisiert um anzuzeigen, ob die Funktion erfolgreich durchgeführt wurde und wie sie sich auf die aktuelle Verknüpfung des Mutators auswirkt. Eine Beschreibung der Kontexte, in denen die verschiedenen Werte für MutatorResult zurückgegeben werden können, findet sich in Tabelle 1.

SharedListNonatomicMutator<T>-Objekte

[0118] Dies ist eine konkrete Klasse, die sich aus SharedListMutator<T> **403** ableitet. Objekte dieser Klasse können die Liste in Vorwärtsrichtung durchlaufen und Verknüpfungen in sie einfügen bzw. aus ihr entfernen, während andere Objekte derselben Klasse die Liste ebenfalls andern.

Überblick über den gleichzeitigen Zugriff auf SharedListIterator<T>-Objekte

[0119] Tabelle 8 fasst die verschiedenen Arten von Konflikten zusammen, die auftreten können, wenn zwei Iteratoren gleichzeitig versuchen, auf ihr SharedList<T>-Objekt zuzugreifen. Wenn bei zwei Iteratoren ein Konflikt auf Listenebene vorliegt, können sie nicht gleichzeitig auf die Liste zugreifen. (Ein Iterator, dessen Sperre für die Liste an sein Vorhandensein gebunden ist, wartet während seiner Erzeugung darauf, dass der den Konflikt verursachende Zugriff beendet wird. Ein Iterator mit einer dynamischen Sperre, die nicht an sein Vorhandensein gebunden ist, wartet an dem Punkt, an dem er aus dem inaktiven Zustand wechselt, oder beginnt eine Änderungsfunktion und hebt seine Sperre auf, sobald er sich im inaktiven Zustand befindet und keine Änderungsfunktion ausführt.) Wenn zwei Iteratoren in einem Konflikt auf Verknüpfungsebene stehen, können sie zwar gleichzeitig auf die Liste zugreifen, müssen sich beim Verweis auf ein und dieselben Verknüpfungszeiger jedoch abwechseln.

Tabelle 8: Überblick über Zugriffskonflikte bei SharedListIterator<T>-Objekten

	Nichtatomarer Inspektor	Nichtatomarer Mutator	Atomarer Inspektor	Atomarer Mutator
Nichtatomarer Inspektor	(keiner)	Verknüpfungsebene*	(keiner)	Verknüpfungsebene*
Nichtatomarer Mutator	Verknüpfungsebene*	Verknüpfungsebene*	Listenebene	Listenebene
Atomarer Inspektor	(keiner)	Listenebene	(keiner)	Listenebene*
Atomarer Mutator	Verknüpfungsebene*	Listenebene	Listenebene	Listenebene

*Nichtatomare Inspektoren müssen bei Konflikten auf Verknüpfungsebene nie auf Mutatoren warten. Die Mutatoren müssen hingegen darauf warten, dass die nichtatomaren Inspektoren die Verknüpfungen, bei denen es zu einem Konflikt kommt, freigeben.

Client-Aufgaben

[0120] Von Clients, die ein SharedList<T>-Objekt verwenden, wird die Einhaltung gewisser Regeln erwartet, um fehlerhafte Verweise usw. zu vermeiden. Das heißt, der Client sollte die definierte Schnittstelle verwenden und nicht versuchen, Zeiger direkt zu ändern. Außerdem sollte der Client die Richtigkeit bestimmter Parameter überprüfen, wenn er Funktionen in Zusammenhang mit dem SharedList<T>-Objekt aufruft; so sollte der Client beispielsweise nicht versuchen, eine Verknüpfung, die bereits in der Liste enthalten ist, in eine Liste einzufügen. Eine umfassende Auflistung derartiger möglicher Problemfälle ist hier nicht enthalten.

[0121] Eine der wichtigeren Client-Aufgaben besteht darin sicherzustellen, dass etwaige Verweise auf frei wählbare Verknüpfungen, die beim Aufrufen von Funktionen wie insertBeforeArbitraryLink(), removeArbitraryLink() usw. weitergegeben werden, tatsächlich Verweise auf Verknüpfungen in der Liste sind und dass sie während der Operation tatsächlich in der verknüpften Liste verbleiben (sofern sie nicht wie hier beschrieben durch einen Mutator entfernt werden).

[0122] Clients können ihre Klassen wahlweise von SharedList<T> und den entsprechenden Iteratoren ableiten oder diese Klassen in eigene Klassen einhüllen. Mittels Einhüllen können der MutatorControl-Parameter für die Mutatorfunktionen und etwaige Funktionen, die nicht offensichtlich sein sollen (gegebenenfalls Funktionen, die sich auf frei wählbare Verknüpfungen beziehen), verborgen werden.

STRUKTUR DER VERKNÜPFTEN LISTE: INTERNE FUNKTIONSWEISE

Einleitung

SharedChain- und SharedChainIterator-Objekte

[0123] Die bisher beschriebenen SharedList<T>- und SharedListIterator<T>-Objekte stellen die Schnittstellen zu dem Client bereit, sind jedoch lediglich Hüllen um die realen Objekte, die für die Durchführung der eigentlichen Aufgaben zuständig sind. Die Hüllklassen stellen die Vorlagenerstellung bereit, indem sie von der T*-Darstellung zur internen Darstellung von Zeigern auf Verknüpfungen wechseln. Ihre gesamten Funktionen befinden sich inline und rufen lediglich Outline-Funktionen für die inneren Objekte auf, so dass die Erstellung von Vorlagen für verschiedene Client-Objekte den Code-Umfang nicht nennenswert erhöht.

[0124] Bei den inneren Objekten handelt es sich um diejenigen Objekte, die im Rest dieser Anmeldung beschrieben werden. Sie sind in derselben Hierarchie angeordnet wie die Hüllobjekte und unterstützen dieselben Funktionen plus einer Vielzahl zusätzlicher interner Funktionen. Die Realisierung der Hüllklassenfunktionen dient lediglich dazu, die entsprechenden Funktionen für die inneren Klassen aufzurufen.

[0125] **Fig.** 5 zeigt die Beziehungen zwischen den verschiedenen externen und internen Klassen und ihren Vererbungshierarchien. Dabei sind die im Folgenden genannten Dinge zu beachten. Die Namen lauten ähnlich: Der Begriff „List" in den Namen der Hüllklassen wird bei den inneren Klassen durch den Begriff „Chain" ersetzt, und die Vorlagenerstellung entfällt. Die Iterator-Klassenhierarchien sind mit Ausnahme des Namenswechsels identisch. Die SharedChainiterator-Basisklasse verfügt über einen Zeiger zu dem SharedChain-Objekt, zu dem sie gehört.

Helferobjekte

[0126] Neben den SharedChain- und SharedChainIterator-Objekten gibt es eine Reihe von internen Objekten, die im Folgenden aufgeführt sind und bei der Ausführung verwendet werden.

[0127] SharedChainBlocker-Objekte: SharedChainBlocker-Objekte (auch kurz als „Sperrobjekte" bezeichnet) werden von Mutatoren dazu verwendet, Verknüpfungen zu sperren bzw. die Sperre von Verknüpfungen aufzuheben sowie die notwendige Synchronisierung mit anderen nichtatomaren Iteratoren (Inspektoren und Mutatoren) durchzuführen. Das SharedChain-Objekt verwaltet eine Gruppe von Sperrobjekten und weist eines davon für die Dauer der Änderungsfunktion einem Mutator zu. Sperrobjekte sowie das Sperren bzw. Entsperren von Verknüpfungszeigern werden weiter unten ausführlicher erläutert.

[0128] SharedChainView-Objekte: Die SharedChainView-Klasse ist eine abstrakte Basisklasse. Aus dieser Klasse abgeleitete Objekte werden von Iteratoren zur Ausführung der eigentlichen Durchlauffunktionen und zur Synchronisierung mit (anderen) Mutatoren verwendet. Aus dieser Basisklasse werden zwei Arten von An-

zeigeobjekten abgeleitet: SharedChainAtomicView und SharedChainNonatomicView.

[0129] SharedChainAtomicView: Dieser Objekttyp wird innerhalb eines atomaren Iterators (Inspektor oder Mutator) erzeugt. Es kann die Verknüpfungen in der Kette verarbeiten und „weiß", dass kein (anderer) Mutator Verknüpfungen einfügen oder entfernen kann. SharedChainAtomicView-Objekte sind vergleichsweise einfach aufgebaut und enthalten im Wesentlichen nur einen Zeiger auf die aktuelle Verknüpfung des Iterators.

[0130] SharedChainNonatomicView: Dieser Objekttyp (auch kurz als „Anzeigeobjekt" bezeichnet) wird von nichtatomaren Inspektoren und Mutatoren verwendet. Es kann die Verknüpfungen in der Kette verarbeiten und „weiß", dass unter Umständen auch (andere) Mutatoren Verknüpfungen einfügen oder entfernen. Das SharedChain-Objekt verwaltet zwei Gruppen von SharedChainNonatomicView-Objekten, von denen eine zur Verwendung durch nichtatomare Inspektoren und die andere zur Verwendung durch nichtatomare Mutatoren vorgesehen ist. Es weist eines der Anzeigeobjekte einem nichtatomaren Iterator (Inspektor oder Mutator) zu, während sich der Iterator im aktiven oder Übergangszustand befindet. Nichtatomare Anzeigeobjekte werden weiter unten ausführlicher erläutert.

[0131] Weitere, hier nicht beschriebene Helferobjekte können zur Unterstützung sekundärer Funktionen, für die Code-Fehlersuche, die Analyse der Systemleistung oder für andere Zwecke verwendet werden.

Überblick über die Funktionsweise

[0132] Das SharedChain-Objekt verwaltet einen Zeiger auf die erste und einen Zeiger auf die letzte Verknüpfung in der Kette. Benachbarte Verknüpfungen zeigen in Vorwärts- und Rückwärtsrichtung aufeinander. Der Vorwärtszeiger der letzten Verknüpfung und der Rückwärtszeiger der ersten Verknüpfung sind NULL. **Fig. 6** zeigt eine vereinfachte Darstellung eines beispielhaften SharedChain-Objekts **601** mit drei Verknüpfungen.

[0133] Da atomare Inspektoren sicher sein können, dass keine Mutatoren mit der verknüpften Liste arbeiten, können sie Verknüpfungszeiger ungehindert in Vorwärts- und Rückwärtsrichtung folgen. Da sie vergleichsweise trivial sind, werden hier lediglich atomare Mutatoren, nichtatomare Mutatoren sowie nichtatomare Inspektoren ausführlicher beschrieben.

[0134] In der folgenden Beschreibung werden atomare Mutatoren relativ ausführlich erörtert. Dabei sollte jedoch klar sein, dass atomare Mutatoren (sowie atomare Inspektoren) in der bevorzugten Ausführungsform aus Gründen ihrer Rückwärtskompatibilität, aufgrund der Unterstützung eventueller spezieller Betriebssystemfunktionen sowie aus ähnlichen Gründen bereitgestellt werden. Der Schwerpunkt der Ausführungsformen der vorliegenden Erfindung besteht darin, die Verwendung von Sperrmechanismen auf Listenebene nach Möglichkeit zu vermeiden, und die Unterstützung atomarer Mutatoren ist für die praktische Durchführung der Ausführungsformen der vorliegenden Erfindung nicht zwingend notwendig.

Nichtatomare Iteratoren

[0135] Nichtatomare Inspektoren durchlaufen die Kette lediglich. Da sie über eine nichtatomare Sicht der Kette verfügen, ist es unerheblich, ob sie eine bestimmte Verknüpfung sehen, die möglicherweise während ihres Vorrückens eingefügt oder entfernt wird. Wichtig ist nur, dass sie sicher von einer Verknüpfung zur nächsten vorrücken können und dass eine bestimmte Verknüpfung nicht aus der Kette entfernt werden kann, solange sie ihre aktuelle Verknüpfung ist.

[0136] Nichtatomare Mutatoren können die Kette ebenfalls durchlaufen und müssen wie nichtatomare Inspektoren auch in der Lage sein, sicher von einer Verknüpfung zur nächsten vorzurücken, und sich darauf verlassen können, dass ihre aktuelle Verknüpfung nicht aus der Kette entfernt wird. Da sie jedoch Verknüpfungen ausgehend von ihrer aktuellen Verknüpfung einfügen oder entfernen können (wenn die Kette z.B. in einer bestimmten Prioritätsfolge verwaltet wird), ist es außerdem wichtig, dass die vorherige Verknüpfung nicht entfernt werden kann und dass zwischen der aktuellen und der vorherigen Verknüpfung keine andere Verknüpfung eingefügt werden kann.

Mutatoren

[0137] Um eine Verknüpfung einzufügen, muss ein Mutator die Vorwärts- und Rückwärtszeiger in der neuen Verknüpfung so initialisieren, dass sie auf die nächste bzw. vorhergehende Verknüpfung verweisen, und den Vorwärtszeiger in der vorherigen Verknüpfung sowie den Rückwärtszeiger in der nächsten Verknüpfung so än-

dern, dass sie auf die neue Verknüpfung verweisen. Um eine Verknüpfung zu entfernen, muss ein Mutator den Vorwärtszeiger in der vorhergehenden Verknüpfung so ändern, dass er auf die nächste Verknüpfung verweist, und den Rückwärtszeiger in der nächsten Verknüpfung so ändern, dass er auf die vorherige Verknüpfung verweist.

[0138] Da die vielen Aktualisierungen nicht atomar erfolgen können, werden sie nacheinander durchgeführt, indem zunächst alle betroffenen Verknüpfungszeiger in einer genau festgelegten Reihenfolge gesperrt werden. (Die Sperre wird weiter unten ausführlicher erläutert.) Ein Mutator, der einen Verknüpfungszeiger gesperrt hat, verfügt über das ausschließliche Recht, ihn zu ändern. Die Tatsache, dass ein Verknüpfungszeiger gesperrt wurde, hindert andere Mutatoren daran, ihn zu sperren oder zu durchlaufen. Nachdem ein Mutator alle betreffenden Verknüpfungszeiger gesperrt hat, hebt er ihre Sperre auf, indem er sie durch ihre beabsichtigten neuen Werte ersetzt (wobei dies ebenfalls in einer genau festgelegten Reihenfolge geschieht); auf diese Weise fügt er die gewünschte Verknüpfung ein bzw. entfernt sie.

Interaktionen zwischen Iteratoren und Mutatoren

[0139] Um eine Verknüpfung einzufügen oder zu entfernen, sperrt ein Mutator zunächst alle betroffenen Verknüpfungszeiger. Ein nichtatomarer Inspektor ist von einem gesperrten Verknüpfungszeiger nicht betroffen. Er kann den gesperrten Verknüpfungszeiger durchlaufen und die nächste Verknüpfung in der Kette als seine (neue) aktuelle Verknüpfung festlegen. Ein durchlaufender (nichtatomarer) Mutator wird von einem gesperrten Verknüpfungszeiger gestoppt. Wenn er bei seinem Vorrücken auf einen gesperrten Verknüpfungszeiger trifft, muss er abwarten, bis dieser wieder entsperrt wird. Ein bereits gesperrter Verknüpfungszeiger kann von einem anderen Mutator nicht ebenfalls gesperrt werden. Wenn er dies wünscht, kann der Mutator abwarten, bis der Verknüpfungszeiger entsperrt wird, und muss dann erneut versuchen, den Verknüpfungszeiger zu sperren.

[0140] Nachdem ein Mutator alle Verknüpfungszeiger gesperrt hat, die für das Einfügen oder Entfernen einer Verknüpfung erforderlich sind, muss er darauf warten, bis alle (nichtatomaren) Inspektoren und durchlaufenden (nichtatomaren) Mutatoren, die auf einen der gesperrten Verknüpfungszeiger angewiesen sind, diese passiert haben. Der sperrende Mutator macht gegenüber jedem derartigen Iterator deutlich, dass dieser den Mutator benachrichtigen muss, sobald er die betreffenden Verknüpfungszeiger passiert hat, und wartet dann, bis alle derartigen Benachrichtigungen eingegangen sind.

[0141] Nachdem ein Mutator sicher ist, dass keine Iteratoren mehr auf die gesperrten Verknüpfungszeiger angewiesen sind, hebt er die Sperre auf und fügt so die gewünschte Verknüpfung ein bzw. entfernt sie. Sobald ein Verknüpfungszeiger entsperrt wurde, kann er von anderen Iteratoren durchlaufen oder durch einen anderen Mutator gesperrt werden, ohne dass diesen bekannt ist, dass seine Sperre erst kurz zuvor aufgehoben wurde. Wenn zwischenzeitlich noch andere Mutatoren darauf warten sollten, dass der Verknüpfungszeiger entsperrt wird, werden sie von dem entsperrenden Mutator geweckt. Als Folge hiervon haben alle durchlaufenden Mutatoren, die auf den gesperrten Vorwärts-Verknüpfungszeiger gewartet haben, ihr Vorrücken atomar abgeschlossen und verwenden als ihre neue aktuelle Verknüpfung diejenige Verknüpfung, auf die der entsperrte Vorwärtszeiger nun zeigt. Alle sperrenden Mutatoren, die auf einen gesperrten Verknüpfungszeiger gewartet haben, müssen erneut versuchen, den Verknüpfungszeiger zu sperren, nachdem dieser entsperrt wurde.

Durchlaufen der Kette

[0142] Wenn sich ein nichtatomarer Iterator (Inspektor oder Mutator) im aktiven oder Übergangszustand befindet, ist ihm ein SharedChainNonatomicView-Objekt zugewiesen. Dieses Anzeigeobjekt ermöglicht dem Iterator, die Kette zu durchlaufen und seine aktuelle Verknüpfung zu erfassen. Das SharedChain-Objekt verwaltet zwei Listen dieser Anzeigeobjekte. Eine Liste wird von nichtatomaren Inspektoren, die andere von nichtatomaren Mutatoren verwendet. Wenn ein nichtatomarer Iterator den inaktiven Zustand verlässt, wird in der entsprechenden Liste ein verfügbares Anzeigeobjekt gefunden und ihm zugewiesen. Wenn alle Anzeigeobjekte in der Liste belegt sind, wird ein neues erzeugt und in die Liste aufgenommen. Wenn der Iterator in den inaktiven Zustand zurückversetzt wird, kann das Anzeigeobjekt von einem anderen nichtatomaren Iterator (desselben Typs) wiederverwendet werden.

[0143] Ein SharedChainNonatomicView-Objekt enthält Daten, welche die aktuelle Verknüpfung des Iterators kenntlich machen. Da alle Anzeigeobjekte in einer der beiden Listen vorhanden sind, können Mutatoren beim Einfügen oder Entfernen von Verknüpfungen alle Anzeigeobjekte auffinden, ermitteln, welche von ihnen von den gerade geänderten Verknüpfungszeigern abhängig sind, und mit ihnen dann entsprechend interagieren.

[0144] Ein Iterator durchläuft die Verknüpfungen in der Kette anhand eines Prozesses, der Ähnlichkeit mit einer mathematischen Herleitung aufweist: Er benötigt eine Vorgehensweise, mit der er aus dem inaktiven Zustand zur ersten Verknüpfung (oder zu der Verknüpfung nach einer frei wählbaren Verknüpfung) gelangen und von einer beliebigen Verknüpfung zur nächsten Verknüpfung vorrücken kann. Das SharedChainNonatomicView-Objekt des Iterators verfügt über zwei Felder, mit denen er seine aktuelle Verknüpfung erfassen kann: curLink **706** enthält die Adresse der aktuellen Verknüpfung des Iterators. Während dieses Feld auf eine Verknüpfung zeigt, kann die Verknüpfung nicht aus der Kette entfernt werden.

[0145] curPtr **705** enthält die Adresse eines Vorwärtszeigers innerhalb einer Verknüpfung. Während ein Iterator über eine aktuelle Verknüpfung verfügt, ist der Vorwärtszeiger, auf den dieses Feld zeigt, davon abhängig, ob es sich bei dem Iterator um einen Inspektor oder einen Mutator handelt. Bei einem Inspektor zeigt dieses Feld auf den Vorwärtszeiger innerhalb der aktuellen Verknüpfung. Bei einem Mutator zeigt es auf den Vorwärtszeiger innerhalb der vorherigen Verknüpfung (die, sofern sie nicht gesperrt ist, auf die aktuelle Verknüpfung zeigt). Dies verhindert auch, dass die vorherige Verknüpfung aus der Kette entfernt wird und dass eine andere Verknüpfung zwischen der vorherigen und der aktuellen Verknüpfung eingefügt wird.

[0146] Fig. 7 zeigt das Anzeigeobjekt **701** für einen durchlaufenden nichtatomaren Mutator sowie das Anzeigeobjekt **702** für einen nichtatomaren Inspektor **702**. Der nichtatomare Mutator sieht „Verknüpfung B" als seine aktuelle Verknüpfung, da in seinem Anzeigeobjekt curLink auf „Verknüpfung B" und curPtr auf den Vorwärtszeiger innerhalb „Verknüpfung A" zeigt. Der nichtatomare Inspektor sieht „Verknüpfung C" als seine aktuelle Verknüpfung, da in seinem Anzeigeobjekt curLink auf „Verknüpfung C" und curPtr auf den Vorwärtszeiger innerhalb von „Verknüpfung C" zeigt.

Starten eines Iterators – Erhalten eines Zeigers auf die erste Verknüpfung

[0147] Im Folgenden wird der Prozess beschrieben, mit dem curLink **706** und curPtr **705** aktualisiert werden, wenn ein Iterator gestartet wird. Bei dieser Beschreibung wird den Interaktionen mit Sperrobjekten nur wenig Beachtung geschenkt, da dies größtenteils weiter unten ausführlich erläutert wird.

1. Wenn die Funktion getFirstLink() aufgerufen wird, um den Iterator in den aktiven Zustand zu versetzen, wird diesem ein Anzeigeobjekt zugewiesen.
2. curPtr **705** wird so gesetzt, dass es auf das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts zeigt.
3. Mit einem weiter unten ausführlicher beschriebenen Algorithmus wird curPtr **705** dereferenziert, um so einen Zeiger auf die erste Verknüpfung zu erhalten, und ihre Adresse wird in curLink **706** gespeichert. Die erste Verknüpfung in der Liste ist nun die aktuelle Verknüpfung des Iterators. Mit Blick auf Fig. 7 zeigt curPtr auf das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts **601**, während curLink auf „Verknüpfung A" zeigt. An dieser Stelle kann „Verknüpfung A" nicht mehr aus der Kette entfernt werden, und vor ihr kann keine Verknüpfung eingefügt werden.
4. Wenn der Iterator ein Mutator ist, endet der Prozess hier. Wenn der Iterator ein Inspektor ist, wird curPtr **705** so gesetzt, dass es auf den Vorwärtszeiger innerhalb der aktuellen Verknüpfung zeigt. Wiederum mit Blick auf Fig. 7 verhindert dies ebenfalls, dass „Verknüpfung A" aus der Kette entfernt wird, wobei nun jedoch andere Verknüpfungen vor ihr eingefügt werden können.

Vorrücken eines Iterators – Erhalten eines Zeigers auf die nächste Verknüpfung

[0148] Inspektoren und Mutatoren verwenden bei ihrem Vorrücken dieselben Schritte, um curLink **706** und curPtr **705** zu setzen, führen diese allerdings in entgegen gesetzter Reihenfolge durch.

1. Wenn die Funktion getNextLink() aufgerufen wird und der Iterator ein Inspektor ist, fährt der Prozess mit Schritt 2 fort. Wenn es sich um einen Mutator handelt, entspricht die Verarbeitung derjenigen, die oben in Schritt 4 für einen Inspektor beschrieben wurde, der einen Zeiger auf die erste Verknüpfung erhält: curPtr **705** wird so gesetzt, dass es auf den Vorwärts-Verknüpfungszeiger innerhalb der aktuellen Verknüpfung zeigt. Mit Blick auf Fig. 7 zeigt curPtr **705** auf den Vorwärtszeiger innerhalb von „Verknüpfung A" und curLink **706** auf „Verknüpfung A". Auch dies verhindert, dass die aktuelle Verknüpfung („Verknüpfung A") aus der Kette entfernt wird, wobei die vorherige Verknüpfung allerdings nicht mehr geschützt ist und andere Verknüpfungen nun unmittelbar vor der aktuellen Verknüpfung eingefügt werden können.
2. Unter Verwendung desselben, weiter unten ausführlicher beschriebenen Algorithmus wird curPtr **705** dereferenziert, um einen Zeiger auf die nächste Verknüpfung zu erhalten, und die Adresse wird in curLink **706** gespeichert. Dabei handelt es sich nun um die neue aktuelle Verknüpfung des Iterators. Mit Blick auf Fig. 7 zeigt curPtr auf den Vorwärtszeiger innerhalb von „Verknüpfung A", während curLink auf „Verknüpfung B" zeigt. An dieser Stelle können weder „Verknüpfung A" noch „Verknüpfung B" aus der Kette entfernt werden,

und es kann keine Verknüpfung zwischen ihnen eingefügt werden.
3. Wenn der Iterator ein Mutator ist, endet der Prozess hier. Wenn der Iterator ein Inspektor ist, wird curPtr **705** so gesetzt, dass es auf den Vorwärts-Verknüpfungszeiger innerhalb der aktuellen Verknüpfung zeigt. Mit Blick auf Fig. 7 verhindert dies wiederum, dass „Verknüpfung B" aus der Kette entfernt wird, stellt jedoch keinen Schutz für „Verknüpfung A" bereit und ermöglicht, dass andere Verknüpfungen dazwischen eingefügt werden.

Vorrücken zum Ende der Kette

[0149] Wenn ein Iterator schließlich am Ende der Kette angelangt ist, wird für die aktuelle Verknüpfung NULL zurückgegeben. Wenn der Iterator ein Inspektor ist, wird er dann in den inaktiven Zustand zurückversetzt. Dadurch wird sein Anzeigeobjekt zurückgesetzt, so dass es über keine aktuelle Verknüpfung mehr verfügt, seine Zuweisung zu dem Iterator wird aufgehoben und es steht für die Wiederverwendung durch einen anderen nichtatomaren Inspektor zur Verfügung.

[0150] Wenn der Iterator jedoch ein Mutator ist, befindet er sich nun im Übergangszustand, und das Anzeigeobjekt muss beibehalten werden, wobei curPtr **705** auf den Vorwärtszeiger innerhalb der letzten Verknüpfung zeigt und curLink **706** NULL enthält. In diesem Zustand kann die letzte Verknüpfung in der Kette nicht entfernt werden, und nach ihr kann keine andere Verknüpfung eingefügt werden. Allerdings kann für diesen Mutator die Funktion insertBeforeCurrentLink() aufgerufen werden, die dann die neue Verknüpfung an das Ende der Kette anhängt. Alternativ kann die Funktion reset() für den Mutator aufgerufen werden, um ihn in den inaktiven Zustand zu versetzen. Das Zurücksetzen des Mutators führt dazu, dass auch sein Anzeigeobjekt zurückgesetzt wird, so dass es über keine aktuelle Verknüpfung verfügt, seine Zuweisung zu dem Mutator aufgehoben wird und es für die Wiederverwendung durch einen anderen nichtatomaren Mutator zur Verfügung steht.

Interaktion mit sperrenden Mutatoren

[0151] Es gibt zwei Punkte, an denen ein sich vorwärts bewegender Iterator mit sperrenden Mutatoren interagieren kann. Dies ist zum einen, wenn ein Iterator das curPtr-Feld **705** in seinem Anzeigeobjekt ändert, d.h., wenn davon ausgegangen wird, dass er zur nächsten Verknüpfung vorgerückt ist. Während der Änderung von curPtr bemerkt ein vorrückender Iterator, ob ein sperrender Mutator darauf wartet, dass er ihn von seinem Vorrücken benachrichtigt. Wenn ein Iterator das (neue) curPtr-Feld in seinem Anzeigeobjekt dereferenziert, stellt er unter Umständen fest, dass der Verknüpfungszeiger gesperrt ist. Ein Inspektor ist in der Lage, über einen gesperrten Verknüpfungszeiger hinwegzugehen, während ein Mutator darauf warten muss, dass der Verknüpfungszeiger entsperrt wird.

Sperren eines Verknüpfungszeigers

[0152] Fig. 8 zeigt die Beziehungen zwischen SharedChain-Objekten, einem Sperrobjekt und Verknüpfungen. Wenn ein (atomarer oder nichtatomarer) Mutator eine Verknüpfung einfügt oder entfernt, ist ihm ein SharedChainBlocker-Objekt **801** zugehörig. Das Sperrobjekt verleiht dem Mutator die Fähigkeit, Verknüpfungszeiger sperren und entsperren zu können sowie mit anderen Mutatoren oder Iteratoren zu interagieren (d.h. nach Bedarf aufeinander zu warten oder einander zu wecken). Das SharedChain-Objekt **601** verwaltet eine Liste von Sperrobjekten und enthält einen Zeiger **803** auf das erste Sperrobjekt in der Liste. Wenn ein Mutator eine Änderungsfunktion beginnt, wird ein verfügbares Sperrobjekt in der Liste ausfindig gemacht und ihm zugewiesen. Wenn alle Sperrobjekte in der Liste belegt sind, wird ein neues erzeugt und zu der Liste hinzugefügt. Wenn die Änderungsfunktion abgeschlossen ist und keine ausstehenden Verweise auf das Sperrobjekt mehr vorhanden sind, steht es für die Wiederverwendung durch eine andere Änderungsfunktion zur Verfügung.

[0153] Ein Mutator sperrt einen Verknüpfungszeiger, um diesen exklusiv nutzen zu können. Er sperrt den Verknüpfungszeiger, indem er ihn so ändert, dass er nicht mehr auf die benachbarte Verknüpfung, sondern auf das Sperrobjekt des Mutators zeigt, während der ursprüngliche Inhalt des Verknüpfungszeigers atomar erhalten wird. Bei der Sperre eines Vorwärts-Verknüpfungszeigers wird der ursprüngliche Inhalt ebenfalls atomar innerhalb des eigentlichen Sperrobjekts (im Feld **802** next_link) gespeichert, so dass nichtatomare Inspektoren über einen gesperrten Verknüpfungszeiger hinweggehen können.

[0154] Die Änderung eines Verknüpfungszeigers hat Auswirkungen auf alle anderen Iteratoren, die darauf verweisen, und es kann vorzugsweise durch eine einfache Überprüfung ermittelt werden, ob der Verknüpfungszeiger auf die benachbarte Verknüpfung oder auf ein Sperrobjekt zeigt. In der Umgebung der bevorzug-

ten Ausführungsform müssen sowohl Verknüpfungen als auch Sperrobjekte an einer mindestens M Bytes umfassenden Grenze liegen, so dass sichergestellt ist, dass die N niederwertigen Bits ihrer Adressen immer Null sind. Bei der bevorzugten Ausführungsform ist M gleich 16 und N gleich 4. Diese Bits in Vorwärts- und Rückwärts-Verknüpfungszeigern können somit als Markierungen verwendet werden. Weitere Einzelheiten hierzu folgen später, an dieser Stelle soll lediglich festgestellt werden, dass, wenn diese Bits inaktiv gesetzt sind, der Verknüpfungszeiger auf die benachbarte Verknüpfung zeigt, und wenn ein Mutator die Adresse seines Sperrobjekts in einem Verknüpfungszeiger speichert, das niederwertige Bit (×01) immer aktiv gesetzt ist. Dieses Bit gibt an, dass der Verknüpfungszeiger gesperrt ist, und zeigt auf ein Sperrobjekt (indem es die vier niederwertigen Bits ausblendet).

[0155] Fig. 8 zeigt ein Sperrobjekt **801**, das den Vorwärts-Verknüpfungszeiger **805** innerhalb „Verknüpfung A" gesperrt hat. Andere durchlaufende und sperrende Mutatoren, die zu dem gesperrten Verknüpfungszeiger gelangen, müssen warten, bis die Sperre wieder aufgehoben ist. Inspektoren, die zu dem gesperrten Verknüpfungszeiger gelangen, können den Zeiger auf „Verknüpfung B" aus dem Sperrobjekt erhalten (über das Feld **802** next_link) und müssen nicht darauf warten, dass der Verknüpfungszeiger entsperrt wird.

Atomare Mutatoren

[0156] Dieser Abschnitt beschreibt die Verarbeitung in Zusammenhang mit verschiedenen Änderungsfunktionen eines atomaren Mutators. Aufgrund der atomaren Sicht des Mutators auf die Kette kann er den Vorwärts- und Rückwärtszeigern innerhalb der Verknüpfungen frei folgen, wobei dies zur Vereinfachung des Prozesses beiträgt. Dies stellt eine gute Orientierungshilfe bereit, die bei der späteren Erörterung von nichtatomaren Mutatoren von Nutzen sein kann. Für eine Übersicht über die verschiedenen Verknüpfungszeiger wird dabei auf die vorhergehenden Figuren verwiesen.

Einfügen einer Verknüpfung – atomare Methoden

[0157] Es gibt verschiedene Funktionen (Methoden), mit denen eine neue Verknüpfung in die Kette eingefügt werden kann. Sie alle können als einfache frontseitige Funktionen betrachtet werden, welche die vorherige Verknüpfung ausfindig machen und anschließend die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur erläuterten Schritte durchführen.

[0158] insertAfterArbitraryLink(): Diese Methode fügt eine neue Verknüpfung unmittelbar nach der angegebenen frei wählbaren Verknüpfung ein (von der angenommen wird, dass sie sich in der Kette befindet). Die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte werden ausgeführt, wobei ein Verweis auf die frei wählbare Verknüpfung weitergegeben wird, die als die vorherige Verknüpfung dient.

[0159] insertAfterCurrentLink(): Diese Methode fügt eine neue Verknüpfung unmittelbar nach der aktuellen Verknüpfung des Mutators ein. Wenn der Mutator über keine aktuelle Verknüpfung verfügt, schlägt die Methode fehl. Andernfalls werden die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte durchgeführt, wobei ein Verweis auf die aktuelle Verknüpfung als die vorherige Verknüpfung weitergegeben wird.

[0160] insertFirst(): Diese Methode fügt eine neue Verknüpfung als die erste Verknüpfung in die Kette ein. Die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte werden durchgeführt, wobei als Verweis auf die vorherige Verknüpfung NULL weitergegeben wird.

[0161] insertBeforeArbitraryLink(): Diese Methode fügt unmittelbar vor der angegebenen frei wählbaren Verknüpfung eine neue Verknüpfung ein (von der angenommen wird, dass sie sich in der Kette befindet). Sie folgt dem Rückwärtszeiger innerhalb der frei wählbaren Verknüpfung, um die vorherige Verknüpfung zu finden. Die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte werden durchgeführt, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0162] insertBeforeCurrentLink(): Diese Methode fügt unmittelbar vor der aktuellen Verknüpfung des Mutators eine neue Verknüpfung ein. Wenn der Mutator über keine aktuelle Verknüpfung verfügt, schlägt die Methode fehl. Andernfalls wird dem Rückwärtszeiger innerhalb der aktuellen Verknüpfung gefolgt, um die vorherige Verknüpfung zu finden, und die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte werden durchgeführt, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0163] insertLast(): Diese Methode fügt eine neue Verknüpfung als die letzte Verknüpfung in die Kette ein. Sie

erhält einen Zeiger auf die zu diesem Zeitpunkt letzte Verknüpfung aus dem Feld **603** last_link_pointer innerhalb des SharedChain-Objekts. Die unter Einfügen einer Verknüpfung – allgemeine atomare Prozedur genannten Schritte werden durchgeführt, wobei ein Verweis auf die letzte Verknüpfung als die vorherige Verknüpfung weitergegeben wird.

Einfügen einer Verknüpfung – allgemeine atomare Prozedur

[0164] Diese Prozedur fügt eine neue Verknüpfung unmittelbar nach der vorherigen Verknüpfung ein. Ein Verweis auf die vorherige Verknüpfung wird als Eingangsparameter weitergegeben.

1. Der Zählwert für die Verknüpfungen in der Kette wird inkrementiert.
2. Der Vorwärtszeiger innerhalb der vorherigen Verknüpfung wird ausfindig gemacht. Wenn die vorherige Verknüpfung allerdings NULL ist, bedeutet dies, dass die neue Verknüpfung als die erste Verknüpfung in die Kette eingefügt wird. In diesem Fall macht die Funktion das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts ausfindig.
3. Dem in Schritt 2 gefundenen Zeiger wird zur nächsten Verknüpfung gefolgt.
4. Der Rückwärtszeiger innerhalb der nächsten Verknüpfung wird ausfindig gemacht. Wenn die nächste Verknüpfung allerdings NULL ist, bedeutet dies, dass die neue Verknüpfung als die letzte Verknüpfung in die Kette eingefügt wird. In diesem Fall macht die Funktion das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts ausfindig.
5. Der Vorwärtszeiger der neuen Verknüpfung wird so initialisiert, dass er auf die nächste Verknüpfung zeigt, und der Rückwärtszeiger wird so initialisiert, dass er auf die vorherige Verknüpfung zeigt. Wenn für die Verknüpfung ein Feld definiert ist, das auf das Kettenobjekt zeigt, wird auch dieses Feld initialisiert.
6. Der (in Schritt 2 gefundene) Vorwärtszeiger innerhalb der vorherigen Verknüpfung wird gesperrt. Hierdurch wird auch ein Zeiger auf die nächste Verknüpfung im Feld **802** next_link innerhalb des Sperrobjekts atomar gespeichert. (Auf diese Weise kann ein beliebiger nichtatomarer Inspektor über den gesperrten Verknüpfungszeiger hinweg zur nächsten Verknüpfung gehen.)
7. Das Feld next_link innerhalb des Sperrobjekts wird so geändert, dass es auf die neue Verknüpfung zeigt. Dadurch wird sichergestellt, dass von diesem Zeitpunkt an jeder nichtatomare Inspektor, der über den gesperrten Verknüpfungszeiger hinweggeht, die neu eingefügte Verknüpfung sieht.
8. Die Methode wartet, bis alle nichtatomaren Inspektoren, die gerade dabei waren, über den gesperrten Verknüpfungszeiger zur nächsten (nicht zur neuen) Verknüpfung hinwegzugehen, dies getan haben.
9. Der Rückwärtszeiger innerhalb der nächsten (in Schritt 4 gefundenen) Verknüpfung wird so geändert, dass er auf die neue Verknüpfung zeigt.
10. Der Vorwärtszeiger innerhalb der vorherigen (in Schritt 6 gesperrten) Verknüpfung wird so geändert, dass er auf die neue Verknüpfung zeigt.

Entfernen einer Verknüpfung – atomare Methoden

[0165] Es gibt verschiedene Funktionen (Methoden), mit denen eine Verknüpfung aus der Kette entfernt werden kann. Sie alle können als einfache frontseitige Funktionen betrachtet werden, welche die vorherige Verknüpfung ausfindig machen und anschließend die im Anschluss daran unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur erläuterten Schritte durchführen.

[0166] removeArbitraryLink(): Diese Methode entfernt eine angegebene frei wählbare Verknüpfung (von der angenommen wird, dass sie sich in der Kette befindet). Sie folgt dem Rückwärtszeiger in der frei wählbaren Verknüpfung, um so die vorherige Verknüpfung ausfindig zu machen. Danach führt sie die unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur erläuterten Schritte durch, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0167] removeLastLink(): Diese Methode entfernt die letzte Verknüpfung aus der Kette. Wenn das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts NULL ist, gibt es keine zu entfernende Verknüpfung. Andernfalls folgt sie dem Feld **603** last_link_pointer, um die letzte Verknüpfung in der Kette ausfindig zu machen, und folgt dem darin enthaltenen Rückwärtszeiger, um die vorherige Verknüpfung ausfindig zu machen. Danach führt sie die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0168] removeFirstLink(): Diese Methode entfernt die erste Verknüpfung aus der Kette. Sie führt die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei NULL als Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0169] removeBeforeArbitraryLink(): Diese Methode entfernt die Verknüpfung unmittelbar vor der angegebenen frei wählbaren Verknüpfung (von der angenommen wird, dass sie sich in der Kette befindet). Wenn der Rückwärtszeiger innerhalb der frei wählbaren Verknüpfung NULL ist, gibt es keine zu entfernende Verknüpfung. Andernfalls folgt sie dem Rückwärtszeiger innerhalb der frei wählbaren Verknüpfung, um die zu entfernende Verknüpfung ausfindig zu machen, und folgt dem darin enthaltenen Rückwärtszeiger, um die vorherige Verknüpfung ausfindig zu machen. Danach führt sie die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0170] removeAfterArbitraryLink(): Diese Methode entfernt die Verknüpfung unmittelbar nach der angegebenen frei wählbaren Verknüpfung (von der angenommen wird, dass sie sich in der Kette befindet). Danach führt sie die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei ein Verweis auf die frei wählbare Verknüpfung als die vorherige Verknüpfung weitergegeben wird.

[0171] removeCurrentLink(): Diese Methode entfernt die aktuelle Verknüpfung des Mutators. Wenn der Mutator über keine aktuelle Verknüpfung verfügt, gibt es keine zu entfernende Verknüpfung. Andernfalls folgt sie dem Rückwärtszeiger innerhalb der aktuellen Verknüpfung, um die vorherige Verknüpfung ausfindig zu machen. Danach führt sie die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei ein Verweis auf die vorherige Verknüpfung weitergegeben wird.

[0172] removeAfterCurrentLink(): Diese Methode entfernt die Verknüpfung, die auf die aktuelle Verknüpfung des Mutators folgt. Danach führt sie die Schritte unter Entfernen einer Verknüpfung – allgemeine atomare Prozedur durch, wobei ein Verweis auf die aktuelle Verknüpfung als die vorherige Verknüpfung weitergegeben wird.

Entfernen einer Verknüpfung – allgemeine atomare Prozedur

[0173] Mit den unten beschriebenen Schritten wird eine Verknüpfung entfernt. Dabei wird ein Verweis auf die vorherige Verknüpfung als Eingangsparameter weitergegeben. Die zu entfernende Verknüpfung ist diejenige, die unmittelbar auf die vorherige Verknüpfung folgt.

1. Der Vorwärtszeiger innerhalb der vorherigen Verknüpfung wird ausfindig gemacht. Wenn die vorherige Verknüpfung allerdings NULL ist, wird die erste Verknüpfung in der Kette entfernt. In diesem Fall macht die Funktion das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts ausfindig.
2. Wenn der in Schritt 1 ausfindig gemachte Verknüpfungszeiger NULL ist, gibt es keine zu entfernende Verknüpfung. Andernfalls sperrt die Methode den Verknüpfungszeiger und fährt mit den folgenden Schritten fort. (Durch die Sperre des Verknüpfungszeigers wird ein Zeiger auf die gerade entfernte Verknüpfung im Feld **802** next_link innerhalb des Sperrobjekts atomar gespeichert. Auf diese Weise können alle nichtatomaren Inspektoren, die auf den gesperrten Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung treffen, über den gesperrten Verknüpfungszeiger hinweggehen und die Verknüpfung finden, die noch zu entfernen ist.)
3. Die zu entfernende Verknüpfung wird ausfindig gemacht, wobei der alte Wert des in Schritt 2 gesperrten Verknüpfungszeigers verwendet wird.
4. Die Methode sperrt den Vorwärtszeiger innerhalb der gerade entfernten Verknüpfung. Hierdurch wird auch ein Zeiger auf die nächste Verknüpfung im Feld **802** next_link innerhalb des Sperrobjekts atomar gespeichert. Von diesem Zeitpunkt an kann ein nichtatomarer Inspektor, der entweder innerhalb der vorherigen oder der gerade entfernten Verknüpfung auf den gesperrten Vorwärtszeiger trifft, die gerade entfernte Verknüpfung nicht mehr sehen, sondern er überspringt sie und sieht stattdessen die nächste Verknüpfung.
5. Die Methode wartet, bis alle (nichtatomaren) Inspektoren, die sich noch auf die Verknüpfung, die gerade entfernt wird, zubewegen oder deren aktuelle Verknüpfung diejenige Verknüpfung ist, die gerade entfernt wird, über diese hinweggegangen sind.
6. Die nächste Verknüpfung wird anhand des alten Werts des in 4 gesperrten Verknüpfungszeigers ausfindig gemacht.
7. Die Methode macht den Rückwärtszeiger innerhalb der nächsten Verknüpfung ausfindig. Wenn die nächste Verknüpfung allerdings NULL ist, handelt es sich bei der gerade entfernten Verknüpfung um die letzte Verknüpfung in der Kette. In diesem Fall macht sie das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts ausfindig.
8. Der in Schritt 7 ausfindig gemachte Rückwärtszeiger wird so geändert, dass er auf die vorherige Verknüpfung zeigt.
9. Der in Schritt 2 gesperrte Vorwärtszeiger in der vorherigen Verknüpfung wird so geändert, dass er auf die nächste Verknüpfung zeigt.
10. Der Zählwert für die Verknüpfungen in der Kette wird dekrementiert.

11. Die Felder für die Vorwärts- und Rückwärts-Verknüpfungszeiger (und das optionale Feld für den Kettenzeiger, falls vorhanden) innerhalb der entfernten Verknüpfung werden auf NULL gesetzt.

Überlegungen zur aktuellen Verknüpfung eines atomaren Mutators

[0174] Ein Mutator kann, muss jedoch nicht eine aktuelle Verknüpfung aufweisen, wenn er mit der Ausführung einer Änderungsfunktion beginnt. Wenn ein atomarer Mutator über eine aktuelle Verknüpfung verfügt, kann er diese aktuelle Verknüpfung bei einer beliebigen von ihm ausgeführten Einfüge- oder Entfernungsfunktion beibehalten, sofern sie nicht die Verknüpfung entfernt, die seine aktuelle Verknüpfung ist. In diesem Fall muss entweder der Mutator zurückgesetzt werden oder eine neue aktuelle Verknüpfung muss ausgewählt werden. Es kann auch sinnvoll sein, einem Mutator eine aktuelle Verknüpfung zuzuweisen oder die aktuelle Verknüpfung des Mutators zu ändern, wenn er eine Änderungsfunktion ausführt. In beiden Fällen sind die Überlegungen nicht weiter von Bedeutung, da der Mutator über eine atomare Sicht verfügt. Von größerer Bedeutung sind diese Überlegungen allerdings für nichtatomare Mutatoren, und um die Kompatibilität mit ihnen zu ermöglichen, verfügen auch atomare Mutatorfunktionen über einen Eingangsparameter MutatorControl, geben MutatorResult zurück (bzw. aktualisieren diesen Parameter) und unterstützen sämtliche Optionen, die zu der aktuellen Verknüpfung gehören.

Nichtatomare Mutatoren

[0175] Dieser Abschnitt beschreibt die Verarbeitung in Zusammenhang mit verschiedenen Änderungsfunktionen eines nichtatomaren Mutators. Aufgrund der nichtatomaren Sicht des Mutators stellt die Erfassung und Sperrung von Verknüpfungszeigern ein anspruchsvolleres Unterfangen dar, da auch andere nichtatomare Mutatoren unter Umständen versuchen, diese zu sperren. Besonders zu achten ist dabei auf die Vermeidung einer gegenseitigen Blockierung. Verknüpfungszeiger müssen daher in einer genau festgelegten Reihenfolge gesperrt und entsperrt werden. In manchen Fällen ist es notwendig, einen vorläufig gesperrten Verknüpfungszeiger auszusetzen und vorübergehend einem anderen Mutator den Vortritt zu lassen. Mit zwei Ausnahmen (insertAfterCurrentLink ()F und removeAfterCurrentLink()) kann ein Mutator eine aktuelle Verknüpfung nicht verwalten, während er eine Änderungsfunktion ausführt. Allerdings kann dem Mutator nach Abschluss der Änderungsfunktion eine neue aktuelle Verknüpfung (welche die Position des Mutators in der Kette beibehält) zugewiesen werden.

Einfügen einer Verknüpfung – nichtatomare Methoden

[0176] Um eine Verknüpfung in die Kette einzufügen, gibt es zwei Verknüpfungszeiger, die zunächst gesperrt werden müssen: den Vorwärtszeiger innerhalb der vorherigen Verknüpfung und den Rückwärtszeiger innerhalb der nächsten Verknüpfung. Dabei ist die Sperre des Vorwärtszeigers in der vorherigen Verknüpfung ausschlaggebend. Indem dieser Verknüpfungszeiger gesperrt wird, steht fest, wo die neue Verknüpfung eingefügt wird.

[0177] Fig. 9 zeigt ein Sperrobjekt für einen Mutator, der die Verknüpfungszeiger gesperrt hat, die notwendig sind, um „Verknüpfung B" zwischen „Verknüpfung A" und „Verknüpfung C" in die Kette einzufügen. Die gesperrten Verknüpfungszeiger (d.h. der Vorwärts-Verknüpfungszeiger **901** in „Verknüpfung A" und der Rückwärts-Verknüpfungszeiger **902** in „Verknüpfung C") zeigen beide auf das Sperrobjekt, wobei das Bit ×01 aktiv gesetzt wird. Das Feld **802** next_link innerhalb des Sperrobjekts zeigt auf „Verknüpfung B", so dass nichtatomare Inspektoren, die auf den gesperrten Vorwärtszeiger innerhalb von „Verknüpfung A" treffen, „Verknüpfung B" als die nächste Verknüpfung sehen.

[0178] Es gibt verschiedene Methoden, die eine Verknüpfung in die Kette einfügen können. Sie alle können als einfache frontseitige Methoden betrachtet werden, mit denen begonnen und von denen ausgehend dann mit anderen Methoden fortgefahren wird.

[0179] insertFirst(): Diese Methode macht das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts ausfindig. Dabei handelt es sich um den Verknüpfungszeiger, der zuerst gesperrt werden muss und der letztlich auf die neue eingefügte Verknüpfung zeigt, die dann die erste Verknüpfung in der Kette ist. Die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur werden ausgeführt, wobei ein Verweis auf das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts weitergegeben wird.

[0180] insertAfterArbitraryLink(): Die angegebene frei wählbare Verknüpfung wird die vorherige Verknüpfung. Die Methode macht den darin enthaltenen Vorwärtszeiger ausfindig. Dies ist der Verbindungszeiger, der zuerst

gesperrt werden muss und der letztlich auf die neu eingefügte Verbindung zeigt, die dann unmittelbar auf die angegebene frei wählbare Verknüpfung folgt. Die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur werden durchgeführt, wobei ein Verweis auf den Vorwärtszeiger innerhalb der angegebenen frei wählbaren Verknüpfung weitergegeben wird.

[0181] insertAfterCurrentLink(): Wenn die aktuelle Verknüpfung des Mutators NULL ist, schlägt die Funktion zwangsläufig fehl. Andernfalls verwendet sie die folgenden Schritte, um zu versuchen, die neue Verknüpfung unmittelbar nach der aktuellen Verknüpfung einzufügen. Dabei besteht die Möglichkeit, dass diese Funktion fehlschlägt, so dass der Mutator zwar über eine neue aktuelle Verknüpfung verfügt, jedoch „seine Position in der Kette" beibehält.

(a) Das curLink-Feld **706** innerhalb des Anzeigeobjekts **702** des Mutators zeigt auf die aktuelle Verknüpfung. Der Mutator versucht, den Vorwärts-Verknüpfungszeiger innerhalb der aktuellen Verknüpfung zu sperren. Bei Bedarf wartet er ab und wiederholt den Sperrversuch, solange der Verknüpfungszeiger nicht durch einen Mutator gesperrt wurde, der diese Verknüpfung entfernen möchte. Wenn der Sperrversuch erfolgreich ist, fährt er mit Schritt (b) fort. Andernfalls schlägt der Sperrversuch fehl, da ein anderer Mutator den Verknüpfungszeiger bereits gesperrt hat und sich dazu verpflichtet hat, die Verknüpfung zu entfernen, die ihn enthält (d.h. die aktuelle Verknüpfung dieses Mutators). Dieser andere (sperrende) Mutator hat auch den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt (auf den das curPtr-Feld **705** dieses Mutators zeigt). Der sperrende Mutator weiß nichts von dem fehlgeschlagenen Versuch dieses Mutators, den Verknüpfungszeiger zu sperren, sieht ihn jedoch unter Umständen (letztlich) nur als einen durchlaufenden Mutator, der dem gesperrten Verknüpfungszeiger zugehörig ist, und wartet – wenn dem so ist – darauf, dass er weiter vorrückt. Um eine gegenseitige Blockierung zu vermeiden, macht das Anzeigeobjekt dieses Mutators dem sperrenden Mutator Platz. Anstelle weiter vorzurücken, wird das Anzeigeobjekt dieses Mutators hierfür geändert, so dass es scheinbar nur bis zu dem gesperrten Vorwärtszeiger innerhalb der vorherigen Verknüpfung (worauf sein curPtr-Feld zeigt) vorrückt. Falls notwendig teilt es dem sperrenden Mutator mit, dass es vorgerückt ist, und wartet dann darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Wenn der sperrende Mutator deren Sperre aufhebt, zeigt er auf das Anzeigeobjekt dieses Mutators und setzt das darin enthaltene curLink-Feld **706** so, dass es auf seine neue aktuelle Verknüpfung zeigt (was das Vorrücken abschließt), und weckt ihn (wie er das für jeden durchlaufenden Mutator tun würde, der an dem gesperrten Verknüpfungszeiger warten würde). Nachdem dieser Mutator geweckt wurde, behält er seine Position in der Kette bei (d.h., das CurPtr-Feld **705** in seinem Anzeigeobjekt bleibt unverändert), da er jedoch über eine neue aktuelle Verknüpfung verfügt, ist es nicht sinnvoll, die Einfügefunktion zu wiederholen. Stattdessen endet der Prozess und gibt eine Fehlermeldung an den Client zurück, der die Funktion getCurrentLink() aufrufen, die neue aktuelle Verknüpfung überprüfen und ermitteln kann, wie er von hier aus fortfahren möchte.

(b) Wenn der Mutator seine aktuelle Verknüpfung nicht beibehalten soll, wird das Anzeigeobjekt des Mutators zurückgesetzt. Wenn er eine neue aktuelle Verknüpfung erhalten soll, wird diese später festgelegt.

(c) Die Methode sperrt den Rückwärtszeiger innerhalb der nächsten Verknüpfung, wobei sie bei Bedarf wartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. (Wenn die nächste Verknüpfung NULL ist, sperrt sie stattdessen das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts). Wenn der zu sperrende Verknüpfungszeiger jedoch von einem anderen Mutator vorläufig gesperrt wurde, versucht sie nicht, ihn zu sperren. Stattdessen verwendet sie den (unten ausführlicher beschriebenen) Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis des anderen Mutators zur Verwendung des Verknüpfungszeigers zu warten. (Der andere Mutator verwendet den Mechanismus für die Verknüpfungsübertragung, um seine Sperre des Verknüpfungszeigers auszusetzen, erteilt so die Erlaubnis zu seiner Verwendung und wartet dann auf die Erlaubnis, fortzufahren.)

(d) Die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur werden durchgeführt, wobei mit Schritt 11 begonnen wird.

[0182] insertLast(): Diese Methode macht das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts ausfindig. Dies ist der Verknüpfungszeiger, der letztlich auf die neue eingefügte Verknüpfung zurück zeigt, die dann die letzte Verknüpfung in der Kette ist. Beginnend mit Schritt 6 werden die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt, wobei ein Verweis auf das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts weitergegeben wird.

[0183] insertBeforeArbitraryLink(): Die angegebene frei wählbare Verknüpfung ist die nächste Verknüpfung. Die Methode macht den darin enthaltenen Rückwärtszeiger ausfindig. Dieser Verknüpfungszeiger zeigt letztlich auf die neue eingefügte Verknüpfung zurück, die dann unmittelbar vor der angegebenen frei wählbaren Verknüpfung steht. Beginnend mit Schritt 6, werden die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt, wobei ein Verweis auf den Rückwärtszeiger innerhalb der angegebenen

frei wählbaren Verknüpfung weitergegeben wird.

[0184] insertBeforeCurrentLink(): Wenn sich der Mutator im inaktiven Zustand befindet, gibt es keinen Platz, in dem die neue Verknüpfung eingefügt werden könnte, und die Funktion schlägt zwangsläufig fehl. Andernfalls verwendet sie die folgenden Schritte, um zu versuchen, die neue Verknüpfung unmittelbar vor der aktuellen Verknüpfung einzufügen. Dabei besteht die Möglichkeit, dass diese Funktion fehlschlägt, so dass der Mutator zwar über eine neue aktuelle Verknüpfung verfügt, jedoch „seine Position in der Kette" beibehält.

(a) Das CurPtr-Feld **705** innerhalb des Anzeigeobjekts des Mutators zeigt auf den Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung. Der Mutator versucht, diesen Verknüpfungszeiger zu sperren, nimmt jedoch weder einen erneuten Versuch vor noch wartet er darauf, dass dies erfolgreich ist. Wenn der Sperrversuch erfolgreich ist, fährt er mit Schritt (b) fort. Andernfalls ist der Sperrversuch fehlgeschlagen, da ein anderer Mutator den Verknüpfungszeiger bereits gesperrt hat. Der andere (sperrende) Mutator weiß nichts von dem fehlgeschlagenen Versuch dieses Mutators, den Verknüpfungszeiger zu sperren, sieht ihn jedoch unter Umständen (letztlich) nur als einen durchlaufenden Mutator, der dem gesperrten Verknüpfungszeiger zugehörig ist, und wartet – wenn dem so ist – auf sein Vorrücken. Um eine gegenseitige Blockierung zu vermeiden, macht das Anzeigeobjekt dem sperrenden Mutator Platz. Anstelle weiter vorzurücken, wird das Anzeigeobjekt dieses Mutators hierfür geändert, so dass es scheinbar nur bis zu dem gesperrten Verknüpfungszeiger vorrückt, und dem sperrenden Mutator wird bei Bedarf mitgeteilt, dass dieser Mutator vorgerückt ist. Dieser Mutator wartet dann darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Wenn der gesperrte Verknüpfungszeiger von einem Mutator gesperrt wurde, der die Verknüpfung mit dem Zeiger entfernen wollte, hat dieser auch den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt. In diesem Fall wird das Anzeigeobjekt dieses Mutators so weiterbewegt, dass es scheinbar nur bis zu dem vorherigen Zeiger vorrückt, und der sperrende Mutator wird bei Bedarf über das Vorrücken unterrichtet. Danach wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Zu diesem Zeitpunkt sieht der sperrende Mutator das Anzeigeobjekt dieses wartenden Mutators, setzt das darin enthaltene curLink-Feld **706** so, dass es auf die Verknüpfung nach der entfernten Verknüpfung zeigt (wodurch sein vorrücken abgeschlossen ist), und weckt ihn, wie er dies für jeden durchlaufenden Mutator tun würde, der auf den gesperrten Verknüpfungszeiger wartet. Nachdem dieser Mutator geweckt wurde, verfügt er über ein neues CurPtr-Feld **705**, während das curLink-Feld **706** gleichgeblieben ist. Das Einfügen wird daraufhin erneut versucht, indem der Prozess an den Anfang von Schritt (a) zurückkehrt. Wenn der gesperrte Verknüpfungszeiger von einem Mutator gesperrt wurde, der die Verknüpfung mit dem Zeiger nicht entfernen möchte, rückt das Anzeigeobjekt dieses Mutators weiter vor, so dass es scheinbar nur bis zu dem gesperrten Verknüpfungszeiger vorrückt, und der sperrende Mutator wird bei Bedarf über das Vorrücken unterrichtet. Dann wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Zu diesem Zeitpunkt sieht der sperrende Mutator das Anzeigeobjekt dieses Mutators, setzt das darin enthaltene curLink-Feld **706** so, dass es auf die Verknüpfung nach der entfernten Verknüpfung zeigt (wodurch sein Vorrücken abgeschlossen wird), und weckt ihn, wie er dies für jeden durchlaufenden Mutator tun würde, der auf den gesperrten Verknüpfungszeiger wartet. Nachdem dieser Mutator geweckt wurde, behält er seine Position in der Kette bei (d.h., das CurPtr-Feld **705** in seinem Anzeigeobjekt bleibt unverändert); da er jedoch über eine neue aktuelle Verknüpfung verfügt, ist es nicht sinnvoll, die Einfügefunktion zu wiederholen. Stattdessen endet der Prozess und gibt eine Fehlermeldung an den Client zurück, der die Funktion getCurrentLink() aufrufen, die neue aktuelle Verknüpfung überprüfen und ermitteln kann, wie er von hier aus fortfahren möchte.

(b) An dieser Stelle wurde der Vorwärtszeiger in der vorherigen Verknüpfung erfolgreich gesperrt. Wenn der Mutator seine aktuelle Verknüpfung beibehalten soll, wird das curPtr-Feld innerhalb seines Anzeigeobjekts so geändert, dass es auf den Vorwärtszeiger innerhalb der gerade eingefügten Verknüpfung zeigt. (Dies ist problemlos möglich, da noch kein anderer Iterator diese neue Verknüpfung sehen kann.) Andernfalls wird das Anzeigeobjekt des Mutators zurückgesetzt, so dass er nicht mehr über eine aktuelle Verknüpfung verfügt. Dies ist notwendig, um eine mögliche gegenseitige Blockierung zu vermeiden, wobei die aktuelle Verknüpfung nicht mehr benötigt wird, um zu ermitteln, vor welcher Verknüpfung die Einfügung stattfinden soll. Die aktuelle Verknüpfung des Mutators kann bei Bedarf später wiederhergestellt werden.

(c) Der Mutator sperrt den Rückwärtszeiger innerhalb der (früheren) aktuellen Verknüpfung, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. (Wenn die (frühere) aktuelle Verknüpfung NULL war, sperrt er stattdessen das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts.) Wenn der zu sperrende Verknüpfungszeiger jedoch von einem anderen Mutator vorläufig gesperrt wurde, versucht er nicht, ihn zu sperren. Stattdessen verwendet er den (weiter unten ausführlich beschriebenen) Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis des anderen Mutators zur Verwendung des Verknüpfungszeigers zu warten. (Der andere Mutator verwendet den Mechanismus für die Verknüpfungsübertragung, um seine Sperre des Verknüpfungszeigers auszusetzen, erteilt so die Erlaubnis zu seiner Verwendung und wartet dann auf die Erlaubnis, fortzufahren.)

a) Die Schritte unter Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur werden durchgeführt.

Einfügen einer Verknüpfung – allgemeine nichtatomare Prozedur

[0185] Hierbei handelt es sich um eine Fortsetzung der oben beschriebenen Funktionen. Dabei gibt es mehrere Startpunkte: Die Methoden insertFirst() und insertAfterArbitraryLink() beginnen mit Schritt 1, wobei ein Verweis auf den Vorwärtszeiger innerhalb der vorherigen Verknüpfung als Eingangsparameter weitergegeben wird, während die Methoden insertLast() und insertBeforeArbitraryLink() mit Schritt 6 beginnen, wobei ein Verweis auf den Rückwärtszeiger innerhalb der nächsten Verknüpfung als Eingangsparameter weitergegeben wird. Die Methoden insertBeforeCurrentLink() und insertAfterCurrentLink() beginnen mit Schritt 11.

1. Der Mutator wird zurückgesetzt um sicherzustellen, dass er über keine aktuelle Verknüpfung verfügt. Dies ist notwendig, um eine gegenseitige Blockierung zu vermeiden. Die aktuelle Verknüpfung des Mutators kann bei Bedarf später wiederhergestellt werden.
2. Der Mutator sperrt den Vorwärtszeiger innerhalb der vorherigen Verknüpfung, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Das Sperren dieses Zeigers gibt einen Zeiger auf die nächste Verknüpfung in der Liste zurück und speichert außerdem automatisch einen Zeiger auf diese Verknüpfung im Feld **802** next_link innerhalb des Sperrobjekts. Die neue Verknüpfung wird zwischen der vorherigen und der nächsten Verknüpfung eingefügt.
3. Der Rückwärtszeiger innerhalb der nächsten Verknüpfung wird ausfindig gemacht. Wenn die nächste Verknüpfung allerdings NULL ist, wird das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts ausfindig gemacht.
4. Der Mutator sperrt den in Schritt 3 gefundenen Verknüpfungszeiger, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Wenn der Verknüpfungszeiger jedoch von einem anderen (sperrenden) Mutator vorläufig gesperrt wurde, versucht er nicht, ihn zu sperren. Stattdessen verwendet er den Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis des sperrenden Mutators zur Verwendung des Verknüpfungszeigers zu warten. (Der sperrende Mutator verwendet den Mechanismus für die Verknüpfungsübertragung, um seine Sperre des Verknüpfungszeigers auszusetzen, erteilt so diesem Mutator die Erlaubnis zu seiner Verwendung und wartet dann auf die Erlaubnis, fortzufahren.)
5. Der Prozess fährt mit Schritt 11 fort.
6. Der Mutator wird zurückgesetzt, um sicherzustellen, dass er über keine aktuelle Verknüpfung verfügt. Dies ist notwendig, um eine gegenseitige Blockierung zu vermeiden. Die aktuelle Verknüpfung des Mutators kann bei Bedarf später wiederhergestellt werden.
7. Der Mutator sperrt den Rückwärtszeiger innerhalb der nächsten Verknüpfung, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Das Sperren dieses Verknüpfungszeigers gibt einen Zeiger auf die vorherige Verknüpfung zurück. Diese Sperre ist vorläufig und muss ausgesetzt werden, wenn der in Schritt 4 noch vorzunehmende Sperrversuch nicht erfolgreich durchgeführt werden kann. Eine (in den Figuren nicht abgebildete) Markierung im gesperrten Verknüpfungszeiger gibt an, dass er nur vorläufig gesperrt ist.
8. Das Feld mit dem Vorwärtszeiger innerhalb der vorherigen Verknüpfung wird ausfindig gemacht. Wenn die vorherige Verknüpfung jedoch NULL ist, wird stattdessen das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts ausfindig gemacht.
9. Der Mutator versucht, den in Schritt 8 gefundenen Verknüpfungszeiger zu sperren, wartet jedoch nicht ab und wiederholt den Versuch auch nicht, bis er erfolgreich ist. Wenn der Sperrversuch erfolgreich ist, fährt er mit Schritt 10 fort. Andernfalls ist der Sperrversuch fehlgeschlagen, da der Verknüpfungszeiger bereits durch einen anderen (sperrenden) Mutator gesperrt wurde. In beiden Fällen muss der sperrende Mutator den in Schritt 7 vorläufig gesperrten Verknüpfungszeiger letztlich sperren. Wenn er feststellt, dass der Verknüpfungszeiger vorläufig gesperrt wurde, verwendet er den Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis zu seiner Verwendung zu warten. Dieser Mutator verwendet den Mechanismus für die Verknüpfungsübertragung, um die Sperre des Verknüpfungszeigers auszusetzen und dem sperrenden Mutator so die Erlaubnis zu seiner Verwendung zu erteilen, während gleichzeitig die in Schritt 7 durchgeführte Sperre aufrechterhalten wird. Indem er die Erlaubnis zur Verwendung des gesperrten Verknüpfungszeigers erteilt, stellt er dem sperrenden Mutator einen Zeiger auf die vorherige Verknüpfung bereit. Dieser Mutator wartet dann unter Verwendung des Mechanismus für die Verknüpfungsübertragung darauf, dass der sperrende Mutator zum Abschluss kommt, woraufhin er die Erlaubnis zum Fortfahren erteilt und einen Zeiger auf die neue vorherige Verknüpfung zurückgibt. Mit einem Zeiger auf die neue vorherige Verknüpfung kehrt dieser Mutator zu Schritt 8 zurück, um erneut zu versuchen, den darin enthaltenen Verknüpfungszeiger zu sperren.
10. An dieser Stelle wurde die Sperre des Vorwärtszeigers der vorherigen Verknüpfung erfolgreich vorgenommen, und der Zeiger auf die nächste Verknüpfung wurde im Feld **802** next_link innerhalb des Sperrob-

jekts atomar gesetzt. Da es nun keinen Grund mehr dafür gibt, dass die in Schritt 7 vorgenommene Sperre vorläufig ist, wird eine Markierung innerhalb des gesperrten Verknüpfungszeigers inaktiv gesetzt und zeigt so an, dass die Sperre nicht mehr vorläufig ist.
11. Der Zählwert für die Verknüpfungen in der Kette wird inkrementiert.
12. Die Vorwärts- und Rückwärtszeiger innerhalb der neuen Verknüpfung werden so gesetzt, dass sie auf das nächste bzw. vorherige Feld zeigen. Wenn es definiert wurde, wird das optionale Feld mit dem Kettenzeiger so gesetzt, dass es auf das SharedChain-Objekt zeigt.
13. Das Feld **802** next_link des Sperrobjekts wird so gesetzt, dass es auf die neue Verknüpfung zeigt. Danach sehen alle Inspektoren, die auf den gesperrten Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung treffen, die neue Verknüpfung.
14. Der Mutator wartet, bis alle vorrückenden nichtatomaren Iteratoren (Inspektoren oder Mutatoren), die auf den gesperrten Vorwärts-Verknüpfungszeiger angewiesen sind, diesen passiert haben. Dabei handelt es sich um Iteratoren, die an dem Vorwärts-Verknüpfungszeiger angekommen waren, bevor er gesperrt wurde, oder aber um Inspektoren, die gerade dabei sind, über den gesperrten Verknüpfungszeiger hinweg zur nächsten (nicht zur neuen) Verknüpfung zu gelangen.
15. Um den Einfügeprozess abzuschließen, werden die Schritte unter Einfügen oder Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt.

Entfernen einer Verknüpfung – nichtatomare Methoden

[0186] Beim Entfernen einer Verknüpfung aus der Kette gibt es mit Blick auf Fig. 10 vier Verknüpfungszeiger, die zunächst gesperrt werden müssen: (a) den Vorwärtszeiger **1002** innerhalb der vorherigen Verknüpfung; (b) den Rückwärtszeiger **1003** innerhalb der gerade entfernten Verknüpfung; (c) den Vorwärtszeiger **1004** innerhalb der gerade entfernten Verknüpfung; und (d) den Rückwärtszeiger **1005** innerhalb der nächsten Verknüpfung. Die Sperre des Vorwärtszeigers in der vorherigen Verknüpfung ist ausschlaggebend. Sobald diese Sperre vorgenommen wurde, steht fest, welche Verknüpfung entfernt wird.

[0187] Fig. 10 zeigt ein Sperrobjekt für einen Mutator, der die Verknüpfungszeiger gesperrt hat, die für die Entfernung von „Verknüpfung B" benötigt werden. Die gesperrten Verknüpfungszeiger zeigen durchweg auf das Sperrobjekt, wobei das Bit ×01 aktiv gesetzt wird. (Zusätzlich ist für den Vorwärtszeiger **1004** in der gerade entfernten Verknüpfung das Bit ×08 ebenfalls aktiv gesetzt um anzuzeigen, dass er sich innerhalb einer Verknüpfung befindet, die gerade entfernt wird.) Das Feld **802** next_link innerhalb des Sperrobjekts zeigt auf „Verknüpfung C", so dass nichtatomare Inspektoren, die auf den gesperrten Vorwärtszeiger innerhalb von „Verknüpfung A" oder innerhalb „Verknüpfung B" treffen, „Verknüpfung B" überspringen und „Verknüpfung C" als nächste Verknüpfung sehen. Das Feld **1001** previous_link_pointer innerhalb des Sperrobjekts zeigt auf den Vorwärtszeiger innerhalb von „Verknüpfung A". Dies wird mitunter von durchlaufenden Mutatoren verwendet, die auf den gesperrten Vorwärtszeiger innerhalb von „Verknüpfung B" treffen und curPtr auf den Vorwärtszeiger innerhalb von „Verknüpfung A" zurücksetzen müssen, um „Verknüpfung C" als ihre neue aktuelle Verknüpfung zu sehen, nachdem „Verknüpfung B" entfernt wurde.

[0188] Es gibt verschiedene Funktionen, mit denen eine Verknüpfung aus der Kette entfernt werden kann. Sie alle können als frontseitige Funktionen betrachtet werden, um den Vorwärtszeiger innerhalb der vorherigen Verknüpfung zu sperren, die danach die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchführen, um die verbleibenden Verknüpfungszeiger zu sperren und die Verknüpfung zu entfernen.

[0189] removeFirstLink(): Das Anzeigeobjekt des Mutators wird zunächst zurückgesetzt um sicherzustellen, dass er über keine aktuelle Verknüpfung verfügt. Dies ist notwendig, um eine gegenseitige Blockierung zu vermeiden. Die aktuelle Verknüpfung des Mutators kann bei Bedarf später wiederhergestellt werden. Danach sperrt der Mutator das Feld **602** first_link_pointer innerhalb des SharedChain-Objekts, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis es erfolgreich gesperrt wurde. Das Sperren dieses Verknüpfungszeigers gibt einen Zeiger auf die erste Verknüpfung in der Kette zurück, bei der es sich um die zu entfernende Verknüpfung handelt. Danach werden die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt, wobei ein Verweis auf das Anzeigeobjekt des gesperrten Verknüpfungszeigers weitergegeben wird.

[0190] removeAfterArbitraryLink(): Der Mutator wird zunächst zurückgesetzt um sicherzustellen, dass er über keine aktuelle Verknüpfung verfügt. Bei der angegebenen frei wählbaren Verknüpfung handelt es sich um die vorherige Verknüpfung. Der Mutator sperrt den darin enthaltenen Vorwärtszeiger, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Das Sperren des Verknüpfungs-

zeigers gibt einen Zeiger auf die Verknüpfung zurück, die unmittelbar auf die frei wählbare Verknüpfung folgt, wobei es sich um die zu entfernende Verknüpfung handelt. Danach werden die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt, wobei ein Verweis auf den gesperrten Verknüpfungszeiger zurückgegeben wird.

[0191] removeArbitraryLink(): Da der Mutator über eine nichtatomare Sicht der Liste verfügt, ist das Auffinden der vorherigen Verknüpfung und das Sperren ihres Vorwärtszeigers ein etwas schwierigeres Unterfangen. Dabei werden die folgenden Schritte durchgeführt:

(a) Das Anzeigeobjekt des Mutators wird zurückgesetzt, um sicherzustellen, dass er über keine aktuelle Verknüpfung verfugt.

(b) Der Mutator sperrt den Rückwärtszeiger innerhalb der gerade entfernten frei wählbaren Verknüpfung, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Das Sperren des Verknüpfungszeigers gibt einen Zeiger auf die vorherige Verknüpfung zurück. Diese Sperre ist vorläufig und muss ausgesetzt werden, wenn der in Schritt (d) noch ausstehende Sperrversuch nicht erfolgreich durchgeführt werden kann. Eine (nicht abgebildete) Markierung in dem gesperrten Verknüpfungszeiger gibt an, dass er nur vorläufig gesperrt ist.

(c) Der Vorwärtszeiger innerhalb der vorherigen Verknüpfung wird ausfindig gemacht. Wenn die vorherige Verknüpfung jedoch NULL ist, wird stattdessen das Feld **602** first_link_pointer innerhalb des Shared-Chain-Objekts ausfindig gemacht.

(d) Der Mutator versucht, den in Schritt (c) gefundenen Verknüpfungszeiger zu sperren, wartet jedoch nicht ab und wiederholt den Versuch auch nicht, bis er erfolgreich ist. Wenn der Sperrversuch erfolgreich ist, fährt er mit Schritt (e) fort. Andernfalls ist der Sperrversuch fehlgeschlagen, da der Verknüpfungszeiger bereits durch einen anderen (sperrenden) Mutator gesperrt wurde. Der sperrende Mutator entfernt entweder die vorherige Verknüpfung, oder er fügt eine neue Verknüpfung nach ihr ein. In beiden Fällen muss er letztlich den in Schritt (b) vorläufig gesperrten Verknüpfungszeiger sperren. Wenn der sperrende Mutator feststellt, dass der Verknüpfungszeiger vorläufig gesperrt wurde, verwendet er den Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis zur Verwendung des Verknüpfungszeigers zu warten. Dieser Mutator verwendet den Mechanismus für die Verknüpfungsübertragung, um die Sperre des Verknüpfungszeigers auszusetzen und dem sperrenden Mutator so die Erlaubnis zu seiner Verwendung zu erteilen, während gleichzeitig die in Schritt (b) durchgeführte Sperre aufrechterhalten wird. Indem ihm die Erlaubnis zur Verwendung des gesperrten Verknüpfungszeigers erteilt wird, erhält der sperrende Mutator einen Zeiger auf die vorherige Verknüpfung. Danach wartet dieser Mutator unter Verwendung des Mechanismus für die Verknüpfungsübertragung ab, bis der sperrende Mutator zum Abschluss gekommen ist, wobei dieser dann dem sperrenden Mutator die Erlaubnis zum Fortfahren gibt und einen Zeiger auf die neue vorherige Verknüpfung zurückgibt. Mit einem Zeiger auf die neue vorherige Verknüpfung kehrt dieser Mutator zu Schritt (c) zurück, um erneut zu versuchen, den darin enthaltenen Verknüpfungszeiger zu sperren.

(e) An dieser Stelle wurde die Sperre des Vorwärtszeigers innerhalb der vorherigen Verknüpfung erfolgreich vorgenommen, und es gibt keinen Grund mehr für die Vorläufigkeit der in Schritt (b) vorgenommenen Sperre. Die Markierung in dem gesperrten Verknüpfungszeiger wird inaktiv gesetzt, so dass sie nicht mehr vorläufig ist.

(f) Danach werden die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt, wobei ein Verweis auf den gesperrten Verknüpfungszeiger zurückgegeben wird.

[0192] removeCurrentLink(): Wenn die aktuelle Verknüpfung des Mutators NULL ist, gibt es keine zu entfernende Verknüpfung, und die Funktion schlägt zwangsläufig fehl. Andernfalls verwendet die Funktion die folgenden Schritte, um zu versuchen, die aktuelle Verknüpfung zu entfernen. Dabei besteht die Möglichkeit, dass diese Funktion fehlschlägt, so dass der Mutator zwar über eine neue Verknüpfung verfügt, jedoch „seine Position in der Kette" beibehält.

(a) Das CurPtr-Feld **705** innerhalb des Anzeigeobjekts des Mutators zeigt auf den Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung. Der Mutator versucht, diesen Verknüpfungszeiger zu sperren, wartet jedoch nicht ab und wiederholt den Versuch auch nicht, bis er erfolgreich ist. Wenn der Sperrversuch erfolgreich ist, fährt er mit Schritt (b) fort. Andernfalls ist der Sperrversuch fehlgeschlagen, da ein anderer (sperrender) Mutator den Verknüpfungszeiger bereits gesperrt hat. Der sperrende Mutator weiß nichts von dem fehlgeschlagenen Versuch dieses Mutators, den Verknüpfungszeiger zu sperren, sieht ihn jedoch (letztlich) nur als einen durchlaufenden Mutator, der dem gesperrten Verknüpfungszeiger zugehörig ist, und wartet – wenn dem so ist – auf sein Vorrücken. Um eine gegenseitige Blockierung zu vermeiden, macht das Anzeigeobjekt dieses Mutators dem Mutator Platz. Anstelle weiter vorzurücken, wird das Anzeigeobjekt dieses Mutators hierfür geändert, so dass es scheinbar nur bis zu dem gesperrten Verknüpfungszeiger vorrückt, und bei Bedarf wird der sperrende Mutator davon unterrichtet, dass es vorgerückt ist. Danach wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Wenn der gesperr-

te Verknüpfungszeiger von einem Mutator gesperrt wurde, der die Verknüpfung mit dem Zeiger entfernen wollte, hat er auch den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt. In diesem Fall wird das Anzeigeobjekt dieses Mutators so weiterbewegt, dass es scheinbar nur bis zu dem vorherigen Zeiger vorrückt, und der sperrende Mutator wird bei Bedarf über das Vorrücken unterrichtet. Danach wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Zu diesem Zeitpunkt sieht der sperrende Mutator das Anzeigeobjekt dieses Mutators, setzt das darin enthaltene curLink-Feld **706** so, dass es auf die Verknüpfung nach der entfernten Verknüpfung zeigt (wodurch sein Vorrücken abgeschlossen wird), und weckt ihn, wie er dies für jeden durchlaufenden Mutator tun würde, der auf den gesperrten Verknüpfungszeiger wartet. Nachdem dieser Mutator geweckt wurde, verfügt er über ein neues CurPtr-Feld **705**, während das curLink-Feld **706** gleichgeblieben ist. Das Entfernen wird daraufhin erneut versucht, indem an den Anfang von Schritt (a) zurückgekehrt wird. Wenn der gesperrte Verknüpfungszeiger jedoch von einem Mutator gesperrt wurde, der die Verknüpfung mit dem Zeiger nicht entfernen möchte, wird das Anzeigeobjekt dieses Mutators so weiterbewegt, dass es scheinbar nur bis zu dem gesperrten Verknüpfungszeiger vorrückt, und der sperrende Mutator wird bei Bedarf über das Vorrücken unterrichtet. Dann wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Zu diesem Zeitpunkt sieht der sperrende Mutator das Anzeigeobjekt dieses wartenden Mutators, setzt das darin enthaltene curLink-Feld **706** so, dass es auf die Verknüpfung nach der entfernten Verknüpfung zeigt (wodurch sein Vorrücken abgeschlossen wird), und weckt ihn, wie er dies für jeden durchlaufenden Mutator tun würde, der auf den gesperrten Verknüpfungszeiger wartet. Nachdem dieser Mutator geweckt wurde, behält er seine Position in der Kette bei (d.h., das curPtr-Feld in seinem Anzeigeobjekt bleibt unverändert); da er jedoch über eine neue aktuelle Verknüpfung verfügt, ist es nicht sinnvoll, die Entfernungsfunktion zu wiederholen. Stattdessen endet der Prozess und gibt eine Fehlermeldung an den Client zurück, der die Funktion getCurrentLink() aufrufen, die neue aktuelle Verknüpfung überprüfen und ermitteln kann, wie er von hier aus fortfahren möchte.

(b) An dieser Stelle wurde der Vorwärtszeiger in der vorherigen Verknüpfung erfolgreich gesperrt. Das Anzeigeobjekt des Mutators wird zurückgesetzt, so dass er über keine aktuelle Verknüpfung mehr verfügt. Dies ist notwendig, um eine mögliche gegenseitige Blockierung zu vermeiden, wobei die aktuelle Verknüpfung nicht mehr benötigt wird, um zu ermitteln, welche Verknüpfung entfernt werden soll. Die aktuelle Verknüpfung des Mutators kann bei Bedarf später wiederhergestellt werden.

(c) Danach werden die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur ausgeführt, wobei ein Verweis auf den gesperrten Verknüpfungszeiger weitergeben wird.

[0193] removeAfterCurrentLink(): Wenn die aktuelle Verknüpfung des Mutators NULL ist, schlägt die Funktion zwangsläufig fehl. Andernfalls werden die folgenden Schritte durchgeführt um zu versuchen, die Verknüpfung unmittelbar nach der aktuellen Verknüpfung zu entfernen. Dabei besteht die Möglichkeit, dass diese Funktion fehlschlägt, so dass der Mutator zwar über eine neue aktuelle Verknüpfung verfügt, jedoch „seine Position in der Kette" beibehält.

a) Das curLink-Feld **706** innerhalb des Anzeigeobjekts des Mutators zeigt auf die aktuelle Verknüpfung. Der Mutator versucht, den Vorwärts-Verknüpfungszeiger innerhalb der aktuellen Verknüpfung zu sperren. Bei Bedarf wartet er ab und wiederholt den Sperrversuch, sofern der Verknüpfungszeiger nicht durch einen Mutator gesperrt wurde, der diese Verknüpfung entfernen möchte. Wenn der Sperrversuch erfolgreich war, fährt er mit Schritt (b) fort. Andernfalls ist der Sperrversuch fehlgeschlagen, da ein anderer (sperrender) Mutator den Verknüpfungszeiger bereits gesperrt hat und die Verknüpfung entfernen möchte, die ihn enthält (d.h. die aktuelle Verknüpfung dieses Mutators). Der sperrende Mutator hat auch den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt (auf den curPtr zeigt). Der sperrende Mutator weiß nichts von dem fehlgeschlagenen Versuch dieses Mutators, den Verknüpfungszeiger zu sperren, sieht ihn jedoch unter Umständen (letztlich) nur als einen durchlaufenden Mutator, der dem gesperrten Verknüpfungszeiger zugehörig ist, und wartet – wenn dem so ist – darauf, dass er weiter vorrückt. Um eine gegenseitige Blockierung zu vermeiden, macht das Anzeigeobjekt dieses Mutators dem sperrenden Mutator Platz. Anstelle weiter vorzurücken, wird das Anzeigeobjekt dieses Mutators hierfür geändert, so dass es scheinbar nur bis zu dem gesperrten Vorwärtszeiger innerhalb der vorherigen Verknüpfung (worauf CurPtr **705** zeigt) vorrückt, und bei Bedarf wird dem sperrenden Mutator mitgeteilt, dass es vorgerückt ist. Daraufhin wartet dieser Mutator darauf, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt. Wenn der sperrende Mutator deren Sperre aufhebt, sieht er das Anzeigeobjekt dieses wartenden Mutators und setzt das darin enthaltene curLink-Feld so, dass es auf die neue aktuelle Verknüpfung dieses Mutators zeigt (wodurch sein Vorrücken abgeschlossen wird), und weckt ihn (wie er das auch für jeden durchlaufenden Mutator tun würde, der an dem gesperrten Verknüpfungszeiger warten würde). Nachdem dieser Mutator geweckt wurde, behält er seine Position in der Kette bei (d.h. curPtr in seinem Anzeigeobjekt bleibt unverändert), da er jedoch über eine neue aktuelle Verknüpfung verfügt, ist es nicht sinnvoll, die Entfernungsfunktion zu wiederholen. Stattdessen endet der Prozess und gibt eine Fehlermeldung an den Client zurück, der die Funktion getCurrentLink()

aufrufen, die neue aktuelle Verknüpfung überprüfen und ermitteln kann, wie er von hier aus fortfahren möchte.

b) Wenn der Mutator seine aktuelle Verknüpfung nicht beibehalten soll, wird das Anzeigeobjekt des Mutators zurückgesetzt. Wenn er eine neue aktuelle Verknüpfung erhalten soll, wird diese später festgelegt.

c) Danach werden die Schritte unter Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur durchgeführt.

Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur

[0194] An dieser Stelle wird ein Verweis auf den Vorwärtszeiger in der vorherigen Verknüpfung als Eingangsparameter weitergegeben. Dieser Verknüpfungszeiger wurde bereits gesperrt, wodurch die zu entfernende Verknüpfung atomar kenntlich gemacht und das Feld **802** in dem Sperrobjekt so gesetzt wird, dass es auf die gerade entfernte Verknüpfung zeigt. Inspektoren, die auf den gesperrten Vorwärts-Verknüpfungszeiger treffen, können über ihn hinweggehen, um die gerade entfernte Verknüpfung ausfindig zu machen. Wenn der Zeiger auf die zu entfernende Verknüpfung NULL ist, gibt es keine zu entfernende Verknüpfung. In diesem Fall werden die unten genannten Schritte übersprungen, und der Prozess fährt mit Einfügen oder Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur fort, um den Vorwärtszeiger in der vorherigen Verknüpfung zu entsperren, und endet dann. Andernfalls werden die folgenden Schritte durchgeführt, um die Verknüpfung zu entfernen.

1. Der Mutator macht den Rückwärtszeiger **1003** innerhalb der gerade entfernten Verknüpfung ausfindig, und sperrt ihn, wenn dieser Mutator ihn nicht bereits gesperrt hat, wobei er bei Bedarf abwartet und den Versuch wiederholt. Dabei hat dieser Mutator den Zeiger nur dann bereits gesperrt, wenn eine frei wählbare Verknüpfung entfernt werden soll. (Wenn der Rückwärts-Verknüpfungszeiger von einem anderen Mutator gesperrt wird, ist zu beachten, dass keine gegenseitige Blockierung entsteht, wenn er darauf wartet, dass dieser entsperrt wird: Erstens kann er nicht nur vorläufig gesperrt worden sein, da die einzigen Funktionen, die den Rückwärtszeiger innerhalb einer Verknüpfung (nicht das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts) vorläufig sperren können, Funktionen sind, die auf diese Verknüpfung als eine frei wählbare Verknüpfung verweisen, wobei dies eine ungültig Aktion wäre, da die Verknüpfung entfernt wird. Zweitens hat jeder Mutator, der den Rückwärtszeiger innerhalb der Verknüpfung bereits gesperrt hat, auch den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt, und da dies nicht mehr der Fall ist, bestünde die einzige Möglichkeit darin, dass der Mutator den Vorwärts-Verknüpfungszeiger in der vorherigen Verknüpfung bereits entsperrt hat und nun dabei ist, den hier relevanten Rückwärtszeiger zu entsperren.)
2. Ein Zeiger auf den gesperrten Vorwärtszeiger **1002** der vorherigen Verknüpfung wird im Feld **1001** previous_link_pointer in dem Sperrobjekt gespeichert. Dies wird benötigt, sobald der Verknüpfungszeiger in Schritt 3 gesperrt wurde, um anderen durchlaufenden Mutatoren zu ermöglichen, sich von der gerade entfernten Verknüpfung zu entfernen.
3. Der Mutator macht den Vorwärtszeiger **1004** innerhalb der gerade entfernten Verknüpfung ausfindig und sperrt ihn. Das Sperren des Verknüpfungszeigers gibt einen Zeiger auf die nächste Verknüpfung zurück. Er setzt auch das Feld **802** next_link innerhalb des Sperrobjekts so, dass es ebenfalls auf die nächste Verknüpfung zeigt, so dass danach nichtatomare Inspektoren über die gesperrten Verknüpfungszeiger hinweggehen können, um die nächste Verknüpfung ausfindig zu machen, und dabei die gerade entfernte Verknüpfung überspringen können. (Falls während der Sperre dieser Verknüpfung gewartet werden muss, gibt es hierfür eine gesonderte Schnittstelle, die ebenfalls den Mechanismus für die Verknüpfungsübertragung beinhaltet, wie weiter unten erläutert wird.)
4. Der Mutator wartet, bis alle nichtatomaren vorrückenden Iteratoren (Inspektoren oder Mutatoren), die auf die gesperrten Verknüpfungszeiger angewiesen sind, diese passiert haben. Dabei handelt es sich um Iteratoren, die bis zu den gesperrten Verknüpfungszeigern gelangt sind, bevor diese gesperrt wurden.
5. Der Rückwärtszeiger **1005** innerhalb der nächsten Verknüpfung wird ausfindig gemacht. Wenn die nächste Verknüpfung jedoch NULL ist, ist die gerade entfernte Verknüpfung die letzte Verknüpfung in der Liste. In diesem Fall wird das Feld **603** last_link_pointer innerhalb des SharedChain-Objekts ausfindig gemacht.
6. Der Mutator sperrt den in Schritt 5 gefundenen Verknüpfungszeiger, wobei er bei Bedarf abwartet und den Versuch so lange wiederholt, bis dieser erfolgreich gesperrt wurde. Wenn der Verknüpfungszeiger jedoch vorläufig von einem anderen Mutator vorläufig gesperrt wurde, versucht er nicht, ihn zu sperren. Stattdessen verwendet er den Mechanismus für die Verknüpfungsübertragung, um auf die Erlaubnis zur Verwendung des Verknüpfungszeigers zu warten. (Der Mutator, der ihn gesperrt hat, verwendet den Mechanismus für die Verknüpfungsübertragung, um seine Sperre des Verknüpfungszeigers auszusetzen, erteilt so die Erlaubnis zu seiner Verwendung und wartet dann auf die Erlaubnis, fortzufahren.)
7. Der Zählwert für die Verknüpfungen in der Kette wird dekrementiert.
8. Die Vorwärts- und Rückwärts-Verknüpfungszeiger (sowie der optionale Kettenzeiger, falls vorhanden) in-

nerhalb der gerade entfernten Verknüpfung werden auf NULL gesetzt.
9. Die Schritte unter Einfügen oder Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur werden durchgeführt, um den Entfernungsprozess abzuschließen.

Einfügen oder Entfernen einer Verknüpfung – allgemeine nichtatomare Prozedur

1. An dieser Stelle kann bei Bedarf eine neue aktuelle Verknüpfung für den Mutator festgelegt (bzw. erneut festgelegt) werden. Die einzige Stelle, an der das Anzeigeobjekt des Mutators problemlos in die Kette aufgenommen werden kann, ist der gesperrte Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung. Das CurPtr-Feld **705** kann so gesetzt werden, dass es auf den Vorwärts-Verknüpfungszeiger zeigt, und das curLink-Feld **706** kann so gesetzt werden, dass es auf die Verknüpfung zeigt, auf welche der Vorwärtszeiger zeigt, wenn er zu einem späteren Zeitpunkt entsperrt wird. Genauer gesagt, wenn eine Verknüpfung entfernt wird, kann die aktuelle Verknüpfung des Mutators so gesetzt werden, dass sie auf die Verknüpfung unmittelbar nach der gerade entfernten Verknüpfung (d.h. die nächste Verknüpfung) zeigt; wenn eine Verknüpfung eingefügt wird, kann die aktuelle Verknüpfung des Mutators so gesetzt werden, dass sie auf die neu eingefügte Verknüpfung zeigt. (Hinweis: Bei den Funktionen insertAfterCurrentLink() und removeAfterCurrentLink() kann der Mutator seine aktuelle Verknüpfung beibehalten haben. Dabei kann sie unverändert bleiben oder so geändert werden, dass sie auf die neu eingefügte Verknüpfung oder auf die Verknüpfung nach der entfernten Verknüpfung zeigt. Alternativ kann der Mutator auch in den inaktiven Zustand zurückversetzt werden, wenn dies gewünscht wird.)
2. Der Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung wird entsperrt, wobei er so geändert wird, dass er auf die nächste Verknüpfung (falls eine Verknüpfung entfernt wurde) oder auf die neue Verknüpfung (falls eine Verknüpfung eingefügt wurde) zeigt. Durch das Entsperren wird der Verknüpfungszeiger atomar geändert, wodurch der Wert zurückgegeben wird, den er zum Zeitpunkt des Sperrens hatte. Der gesperrte Verknüpfungszeiger enthält zusätzliche Markierungen, die anzeigen, ob andere durchlaufende oder sperrende Mutatoren darauf warten, dass der Verknüpfungszeiger entsperrt wird. Dabei ist jedoch ein Sonderfall zu berücksichtigen. Wenn ein sperrender Mutator, der die Verknüpfung mit dem gesperrten Vorwärts-Verknüpfungszeiger entfernen möchte, darauf wartet, dass der Verknüpfungszeiger entsperrt wird, wird dieser nicht entsperrt. Stattdessen wird der gesperrte Vorwärts-Verknüpfungszeiger (atomar) so aktualisiert, dass er von dem anderen Mutator gesperrt zu sein scheint (wobei er auf sein Sperrobjekt zeigt und das Bit ×09 aktiv gesetzt ist), der die ihn enthaltende Verknüpfung entfernen möchte. Danach wird der Mechanismus für die Verknüpfungsübertragung verwendet, um dem Mutator die Erlaubnis zur Verwendung des Verknüpfungszeigers zu erteilen, wobei der Zeiger auf die nächste Verknüpfung an ihn weitergegeben wird (d.h. der Wert, auf den der Verknüpfungszeiger andernfalls beim Entsperren gesetzt worden wäre).
3. Der Rückwärts-Verknüpfungszeiger innerhalb der nächsten Verknüpfung wird entsperrt, wobei er so geändert wird, dass er auf die vorherige Verknüpfung (falls eine Verknüpfung entfernt wurde) oder auf die neue Verknüpfung (falls eine Verknüpfung eingefügt wurde) zeigt. Durch das Entsperren wird der Verknüpfungszeiger atomar geändert, wodurch der Wert zurückgegeben wird, den er zum Zeitpunkt des Sperrens hatte. Der gesperrte Verknüpfungszeiger enthält zusätzliche Markierungen, die anzeigen, ob andere sperrende Mutatoren darauf warten, dass der Verknüpfungszeiger entsperrt wird. Wenn der Verknüpfungszeiger jedoch nicht von diesem Mutator, sondern von einem anderen Mutator vorläufig gesperrt wurde, muss er nicht entsperrt werden. Stattdessen hat der andere Mutator seine Verwendung des Verknüpfungszeigers ausgesetzt und wartet auf die Erlaubnis, fortzufahren. In diesem Fall wird der Mechanismus für die Verknüpfungsübertragung verwendet, um dem Mutator die Erlaubnis zum Fortfahren zu geben, wobei ein Zeiger auf eine Verknüpfung an ihn weitergegeben wird, die für ihn eine neue vorherige Verknüpfung darstellt (d.h. der Wert, auf den der Verknüpfungszeiger andernfalls beim Entsperren gesetzt worden wäre). (Beim Entfernen einer Verknüpfung wird nach dem Sperren des Vorwärts-Verknüpfungszeigers in der vorherigen Verknüpfung mitunter festgestellt, dass es keine zu entfernende Verknüpfung gibt. In diesen Fällen gibt es keinen gesperrten Rückwärts-Verknüpfungszeiger innerhalb der nächsten Verknüpfung, so dass dieser Schritt als Ganzes übersprungen werden sollte.)
4. Wenn es durchlaufende Mutatoren gibt, die darauf warten, dass der Vorwärts-Verknüpfungszeiger in der vorherigen Verknüpfung entsperrt wird, werden sie nun geweckt. Sie können gefunden werden, indem die Liste der Anzeigeobjekte **701** innerhalb des SharedChain-Objekts auf nichtatomare Mutatoren durchsucht wird.
5. Wenn es sperrende Mutatoren gibt, die darauf warten, dass einer der beiden Verknüpfungszeiger entsperrt wird, werden sie nun geweckt. Sie können gefunden werden, indem die Liste der Sperrobjekte innerhalb des SharedChain-Objekts beginnend mit dem Zeiger **803** durchsucht wird.
6. An dieser Stelle ist die Einfüge- bzw. Entfernungsfunktion abgeschlossen. Das Sperrobjekt kann für eine andere Funktion wiederverwendet werden, sobald alle anderen Inspektoren und Mutatoren, die geweckt wurden, nicht mehr darauf verweisen.

Überlegungen zur aktuellen Verknüpfung des nichtatomaren Mutators

[0195] Um eine gegenseitige Blockierung zu vermeiden, muss ein nichtatomarer Mutator zurückgesetzt werden (bzw. muss zumindest sein Anzeigeobjekt intern zurückgesetzt werden), so dass er über keine aktuelle Verknüpfung verfügt, während er eine Änderungsfunktion durchführt (wobei dies nicht für die Funktionen insertAfterCurrentLink()F und removeAfterCurrentLink() gilt). Ein Mutator kann bei einer Änderungsfunktion nur dann seine „Position in der Kette beibehalten", wenn ihm gestattet wird, nach Abschluss der Funktion erneut eine neue aktuelle Verknüpfung festzulegen. Außerdem ist die einzige Verknüpfung, die problemlos zur neuen aktuellen Verknüpfung werden kann, diejenige Verknüpfung, die dem Mutator gestattet, „seine Position beizubehalten". Der Zustand des Mutators lässt sich – wie weiter oben beschrieben – mit den Werten für MutatorControl steuern.

VERKNÜPFTE LISTENDATENSTRUKTUR: INTERNE EINZELHEITEN

[0196] Dieser Abschnitt beschreibt die Einzelheiten der Interaktionen zwischen nichtatomaren Anzeige- und Sperrobjekten sowie zwischen Sperrobjekten. Anstelle des im Abschnitt über die interne Funktionsweise beschriebenen größeren Zusammenhangs konzentriert sich dieser Abschnitt auf einzelne Interaktionen. Informationen, wie einzelne Interaktionen in den größeren Zusammenhang eingebunden sind, finden sich im Abschnitt zur internen Funktionsweise. Als weitere Hilfestellung für das Verständnis der oben beschriebenen Ausführungsform dienen einige zusätzliche Beispiele und Figuren.

Datenbereiche und Definitionen

Verknüpfungszeiger

[0197] Wie bereits erwähnt, bestehen die Adressen von Verknüpfungen in der Kette aus mehreren Bytes, so dass mindestens N niederwertige Bits gleich Null sind (in der Umgebung der bevorzugten Ausführungsform ist N 4, wobei die Anzahl der Bytes gleich 16 ist und wobei klar sein sollte, dass auch andere Werte verwendet werden könnten bzw. dass anstelle von Markierungen in den Adressen auch andere Mechanismen wie beispielsweise zweckbestimmte Markierungsfelder verwendet werden könnten).

[0198] Die erwähnten N Bits sind in den Adressen der Sperrobjekte ebenfalls gleich Null und werden somit als Markierungsbits verwendet. Tabelle 9 erläutert die Bedeutung der einzelnen Markierungsbits innerhalb der Verknüpfungszeiger der Kette.

Tabelle 9: Bedeutung von Markierungsbits in Verknüpfungszeigern

Bit	Bedeutung	Beschreibung
×01	Gesperrter Verknüpfungszeiger (blocked_link_pointer)	Wenn es aktiv gesetzt ist, gibt dieses Markierungsbit an, dass der Verknüpfungszeiger gesperrt ist und dass die Maskierung der N niederwertigen Bits einen Zeiger auf das Sperrobjekt ergibt, der zu dem Mutator gehört, der es gesperrt hat. Die verbleibenden Markierungsbits sind nur dann aktiv gesetzt, wenn dieses Bit aktiv ist.
×02	Durchlaufender Mutator wartet (iterating_mutator_waiting)	Wenn es aktiv gesetzt ist, gibt dieses Markierungsbit an, dass mindestens ein durchlaufender Mutator darauf wartet (bzw. sich vorbereitet, darauf zu warten), dass der Verknüpfungszeiger entsperrt wird. Wenn der entsperrende Mutator den Verknüpfungszeiger entsperrt und dieses Markierungsbit sieht, durchsucht er die Liste der Anzeigeobjekte auf nichtatomare Mutatoren und weckt alle, die sich dazu verpflichtet haben, darauf zu warten, dass dieser Verknüpfungszeiger entsperrt wird. Dieses Markierungsbit wird nur in Vorwärts-Verknüpfungszeigern verwendet.
×04	Sperrender Mutator wartet (blocking_mutator_waiting)	Wenn es aktiv gesetzt ist, gibt dieses Markierungsbit an, dass mindestens ein sperrender Mutator darauf wartet (bzw. sich vorbereit, darauf zu warten), dass der Verknüpfungszeiger entsperrt wird. Wenn der entsperrende Mutator den Verknüpfungszeiger entsperrt und dieses Markierungsbit sieht, durchsucht er die Liste der Sperrobjekte und weckt alle Mutatoren, die sich dazu verpflichtet haben, darauf zu warten, dass er seine Verknüpfungszeiger entsperrt. Dieses Bit kann sowohl in einem Vorwärtsals auch in einem Rückwärts-Verknüpfungszeiger verwendet werden. In einem Vorwärts-Verknüpfungszeiger hat es darüber hinaus eine besondere Bedeutung in Zusammenhang mit dem unten beschriebenen Bit ×08.
×08	Vorläufige Sperre (tentative_block)	Wenn dieses Markierungsbit in einem Rückwärts-Verknüpfungszeiger aktiv gesetzt ist, gibt es an, dass die Sperre für den Verknüpfungszeiger vorläufig ist und ausgesetzt werden kann.
×08	Entfernen erfolgt in Kürze (remove_pending)	Wenn dieses Markierungsbit in einem Vorwärts-Verknüpfungszeiger aktiv gesetzt ist, gibt es an, dass die Verknüpfung mit dem Vorwärtszeiger in Kürze entfernt werden wird. Der Mutator, der die Verknüpfung entfernen möchte, hat (auch) den Vorwärtszeiger in der vorherigen Verknüpfung gesperrt. Die genaue Bedeutung dieses Bits ist abhängig vom Wert des Bits ×04: Wenn das Bit ×04 inaktiv gesetzt ist, bedeutet dies, dass der Mutator, der den Verknüpfungszeiger gesperrt hat, auch derjenige ist, der die Verknüpfung entfernen möchte. Wenn das Bit ×04 aktiv gesetzt ist, bedeutet dies, dass der Mutator, der darauf wartet, den Verknüpfungszeiger zu sperren (es kommt nur ein einziger in Frage), beabsichtigt, die Verknüpfung zu entfernen, wenn es ihm gelingt, den Verknüpfungszeiger zu sperren.

[0199] Aktualisierungen von Verknüpfungszeigern erfolgen atomar, wobei der aktuelle Wert des Verknüpfungszeigers zugrunde gelegt wird. Bei der bevorzugten Ausführungsform wird hierfür eine spezielle Systemfunktion mit der Bezeichnung compareAndSwap() verwendet. Zusätzlich wird in manchen Fällen eine spezielle Systemfunktion namens compareAndSwapQualified() verwendet; sie führt eine vorübergehende Umspeiche-

rung eines einzelnen Zeigers durch, vergleicht jedoch auch ein größeres Segment des Speichers (ein Quadword) und stellt eine Fehlermeldung bereit, wenn ein Teil des größeren Speichersegments während der Operation geändert wurde. Ähnliche Funktionen können in anderen Systemen verwendet werden bzw. alternative Mittel für die Bereitstellung der Atomizität können bei Operationen für die Aktualisierung von Verknüpfungszeigern zum Einsatz kommen.

Nichtatomare Anzeigeobjekte

[0200] Wenn sich ein nichtatomarer Iterator (Inspektor oder Mutator) im aktiven oder Übergangszustand befindet, ist ihm ein Anzeigeobjekt zugehörig. Das Anzeigeobjekt enthält zwei Felder, mit denen die aktuelle Verknüpfung des Iterators erfasst wird. Wenn ein Iterator die Kette durchläuft, müssen diese Felder auf eine genau festgelegte Art und Weise aktualisiert werden. Ihre Bedeutung muss jederzeit klar sein, da Mutatoren jederzeit Verknüpfungszeiger sperren und entsperren können und in der Lage sein müssen zu ermitteln, ob ein gegebenes Anzeigeobjekt für sie von Belang ist.
curPtr **705**: Dieses Feld enthält einen Zeiger auf einen Vorwärts-Verknüpfungszeiger. Der Vorwärts-Verknüpfungszeiger, auf den es zeigt, befindet sich entweder innerhalb der aktuellen Verknüpfung oder innerhalb der vorherigen Verknüpfung. Bei der bevorzugten Ausführungsform befinden sich alle Verknüpfungszeiger innerhalb von Verknüpfungen an einer 8-Byte-Grenze, so dass zumindest die drei niederwertigen Bits ihrer Adressen gleich Null sind. Innerhalb des curPtr-Felds werden diese Bits als Markierungsbits verwendet, wie dies in Tabelle 10 definiert ist.

Tabelle 10: Bedeutung von Markierungsbits in curPtr

Bit	Bedeutung	Beschreibung
×01	Objekt mit Warteverpflichtung (committed_waiter)	Wenn es aktiv gesetzt ist, gibt dieses Markierungsbit an, dass der Mutator, der den Verknüpfungszeiger gesperrt hat, auf den curPtr zeigt, sich dazu verpflichtet hat zu warten, bis der Iterator, zu dem dieses Anzeigeobjekt gehört, den gesperrten Verknüpfungszeiger passiert hat.
×02	Priorität erhöht (priority_bumped)	Diese Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. In aktivem Zustand zeigt es an, dass der Mutator, der sich dazu verpflichtet hat, auf den Iterator zu warten, die Priorität der Iterator-Aufgabe erhöht hat.
×04	Warten entfernen (remove_waiting)	Dieses Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. Wenn es aktiv gesetzt ist, gibt es an, dass der wartende Mutator beabsichtigt, die Verknüpfung, welche die aktuelle Verknüpfung des Iterators ist, zu entfernen. In diesem Fall sollte der Iterator den Mutator erst dann von seinem Vorrücken unterrichten, wenn er zwei Verknüpfungen vorgerückt ist (technisch gesprochen, wenn er zwei Aktualisierungen von curPtr vorgenommen hat).

curLink **706**: Dieses Feld enthält einen Zeiger auf die aktuelle Verknüpfung des Iterators. Da Verknüpfungen an einer 16-Byte-Grenze liegen müssen, sind zumindest die vier niederwertigen Bits innerhalb ihrer Adressen gleich Null. Innerhalb des curLink-Felds werden drei dieser Bits als Markierungsbits verwendet, wie in Tabelle 11 definiert ist.

Tabelle 11: Bedeutung von Markierungsbits in curLink

Bit	Bedeutung	Beschreibung
×01	Objekt mit Warteverpflichtung (committed_waiter)	Wenn dieses Markierungsbit aktiv gesetzt ist, gibt es an, dass der Iterator einen gesperrten Verknüpfungszeiger angetroffen hat und dass er sich dazu verpflichtet hat zu warten, bis dieser entsperrt wird. Anstelle auf eine Verknüpfung innerhalb der Liste zu zeigen, zeigt curLink – wenn dieses Bit aktiv gesetzt ist – auf das Sperrobjekt, das zu dem Mutator gehört, der den Vorwärts-Verknüpfungszeiger gesperrt hat, auf den curPtr zeigt (wobei die vier niederwertigen Bits maskiert werden). Dieses Markierungsbit ist nur in Anzeigeobjekten für Mutatoren aktiv; Inspektoren können gesperrte Verknüpfungszeiger passieren, ohne zu warten.
×02	Priorität erhöht (priority_bumped)	Dieses Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. Wenn es aktiv gesetzt ist, gibt es an, dass der wartende
×04	Priorität nicht erhöht (priority_not_bumped)	Iterator die Priorität der Aufgabe des sperrenden Mutators erhöht hat. Dieses Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. Wenn es aktiv gesetzt ist, gibt es an, dass der wartende Iterator ermittelt hat, dass die Priorität der Aufgabe des sperrenden Mutators nicht erhöht werden muss.

[0201] Neben den obigen Markierungsbits gibt es zwei spezielle Werte, die im curLink-Feld **706** auftreten können:

Tabelle 12: Spezielle curLink-Werte

Wert	Bedeutung	Beschreibung
×00 ... 02	Übergang (transition)	Dieser spezielle Wert gibt an, dass während des Vorrückens des Iterators curPtr zwar gesetzt wurde, das Ergebnis der Dereferenzierung von curPtr jedoch noch nicht in curLink gesetzt wurde. Ein sperrender Mutator, der diesen Wert in curLink sieht, weiß nicht, ob die Dereferenzierung von curPtr bereits stattgefunden hat.
×00 ... 04	Nichtübereinstimmung (miscompare)	Wenn ein Mutator, der den Verknüpfungszeiger, auf den curPtr zeigt, gerade gesperrt oder entsperrt hat, den Übergangswert in curLink sieht, weiß er nicht, ob es sich (aus Sicht des Mutators) um den alten oder den neuen Wert des Verknüpfungszeigers handelt, den der vorrückende Iterator in Kürze in curLink speichern wird. Aus diesem Grund versucht der Mutator, curLink auf den Nichtübereinstimmungswert zu setzen, bevor der Iterator darin speichert, was er durch die Dereferenzierung von curPtr erhalten hat, um so den Speicherversuch des Iterators fehlschlagen zu lassen. Wenn es dem Iterator nicht gelingt, curLink zu ändern, dereferenziert er curPtr, bevor er den Speicherversuch wiederholt. Auf diese Weise kann der Mutator sich über den Wert sicher sein, den der Iterator durch die Dereferenzierung von curPtr erhält und in curLink speichert.

[0202] Diese Felder werden anhand der Funktion compareAndSwap() atomar aktualisiert. Da diese Funktion nicht beide Fälle in einer einzigen atomaren Operation aktualisieren kann, wird die Funktion compareAndSwapQualified() verwendet, die jeweils ein einziges Feld aktualisiert, jedoch fehlschlägt, wenn eines der Felder zwischenzeitlich aktualisiert wird.

[0203] Zusätzlich verfügt ein nichtatomares Anzeigeobjekt über ein Objekt priority_bump_uncertainty_resolver. Dabei handelt es sich um ein Sperrobjekt für eine einzige Aufgabe, d.h. um einen Synchronisierungsmechanismus, der nur jeweils eine Aufgabe betrifft. Er wird verwendet, wenn nicht klar ist, ob ein Mutator die Priorität der Iterator-Aufgabe erhöht hat, um so zu warten, bis die Prioritätserhöhung sicher ist. Auf diese Weise wird verhindert, dass die Prioritätserhöhung vor der Erhöhung bereits wieder aufgehoben wird.

Sperrobjekte

[0204] Wenn ein (atomarer oder nichtatomarer) Mutator eine Änderungsfunktion ausführt, ist ihm ein Sperrobjekt zugehörig. Das Sperrobjekt enthält mehrere Felder, die seinen Zustand definieren:
Aktivierungskennung (activation_id): Jedes Mal, wenn ein Sperrobjekt einem Mutator zugewiesen wird, wird diesem Feld eine eindeutige Nummer (eindeutig mit Bezug auf das SharedChain-Objekt) zugewiesen. Wenn ein Inspektor sieht, dass ein Verknüpfungszeiger gesperrt ist, kann er – wenn er erneut überprüft, ob dieser (noch) gesperrt ist – die Felder mit der Aktivierungskennung miteinander vergleichen, um zu ermitteln, ob der Verknüpfungszeiger weiter gesperrt geblieben ist oder ob er ein zweites Mal gesperrt wurde. Dieses Feld wird zugewiesen, bevor der Mutator anhand des Sperrobjekts einen Verknüpfungszeiger sperrt.

[0205] Nächste Verknüpfung (next_link): Dieses Feld enthält einen Zeiger auf die nächste Verknüpfung in der Liste und ermöglicht in erster Linie Inspektoren, einen gesperrten Vorwärts-Verknüpfungszeiger zu passieren. Dieses Feld besteht eigentlich aus zwei Feldern, dem primären und dem sekundären Feld. Dabei werden beide Felder benötigt, um den Eindruck zu erwecken, dass das Feld atomar gesetzt wird, wenn eine Verknüpfung

entfernt wird. Wenn ein Iterator zu der nächsten Verknüpfung vorrückt und auf einen gesperrten Verknüpfungszeiger trifft, sollte er normalerweise das primäre Feld verwenden, um die nächste Verknüpfung zu finden. Wenn sich der gesperrte Verknüpfungszeiger jedoch innerhalb einer Verknüpfung befindet, die von dem Mutator, der den Zeiger gesperrt hat, entfernt wurde, sollte der Iterator das sekundäre Feld verwenden, um die nächste Verknüpfung zu finden.

[0206] Warten auf Sperrobjekte (waiting_for_blocker): Dieses Feld gibt an, dass der das Sperrobjekt verwendende Mutator darauf wartet, dass ein zweiter Mutator, der ein zweites Sperrobjekt verwendet, seine Verknüpfungszeiger entsperrt. Normalerweise enthält dieses Feld den Wert NULL, um anzugeben, dass der Mutator nicht wartet. Wenn er auf einen zweiten Mutator wartet oder sich darauf vorbereitet, auf ihn zu warten, wird in diesem Feld ein Zeiger auf das Sperrobjekt des zweiten Mutators gespeichert. Die N niederwertigen Bits einer Sperrobjektadresse sind Null. In diesem Feld werden die drei niederwertigen Bits als Markierungsbits verwendet. Tabelle 13 beschreibt die verschiedenen Markierungsbits, wenn eine Sperrobjektadresse gespeichert wird.

Tabelle 13: Warten auf Markierungsbits eines Sperrobjekts

Bit	Bedeutung	Beschreibung
×00	Nicht verpflichtet (uncommitted)	Wenn alle Markierungsbits inaktiv gesetzt sind, bereitet sich der Mutator darauf vor, auf den zweiten Mutator zu warten, hat sich jedoch noch nicht zum Warten verpflichtet.
×01	Verpflichtet (committed)	Wenn dieses Markierungsbit aktiv gesetzt ist, hat sich der Mutator verpflichtet, auf den zweiten Mutator zu warten.
×02	Priorität erhöht (priority_bumped)	Dieses Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. Wenn es aktiv gesetzt ist, gibt es an, dass der Mutator die Priorität der Aufgabe des zweiten Mutators erhöht hat.
×04	Priorität nicht erhöht (priority_not_bumped)	Dieses Markierungsbit ist nur dann aktiv gesetzt, wenn das Markierungsbit ×01 aktiv ist. Wenn es aktiv gesetzt ist, gibt es an, dass der Mutator ermittelt hat, dass die Priorität der Aufgabe des zweiten Mutators nicht erhöht werden muss.

[0207] Vorheriger Verknüpfungszeiger (previous_link_pointer): Dieses Feld wird nur verwendet, wenn der Mutator, der das Sperrobjekt verwendet, eine Verknüpfung entfernt. Nachdem er den Vorwärtszeiger in der vorherigen Verknüpfung entfernt hat, speichert er einen Zeiger auf den Verknüpfungszeiger in diesem Feld und sperrt dann den Vorwärtszeiger in der gerade entfernten Verknüpfung. Ein zweiter Mutator, dessen Versuch, seine aktuelle Verknüpfung zu entfernen oder eine Einfügung vor ihr vorzunehmen, dann fehlschlägt, weil er feststellt, dass curPtr durch den ersten Mutator gesperrt wurde (der die Verknüpfung mit dem Vorwärtszeiger, auf den curPtr zeigt, entfernen möchte), muss curPtr so ändern, dass es auf den Vorwärtszeiger in der vorherigen Verknüpfung zeigt. Dieses Feld ermöglicht dem durchlaufenden Mutator, den Vorwärtszeiger in der vorherigen Verknüpfung ausfindig zu machen.

[0208] Statussteuerung (status_control): Dies ist ein Zustandszähler-Objekt, d.h. ein Zähler, der bei bestimmten Ereignissen inkrementiert oder dekrementiert wird, wobei unter Umständen Aufgaben darauf warten, dass der Zähler einen bestimmten Wert erreicht. Er ist der primäre Mechanismus, der die Warte-/Weck-Interaktionen zwischen Sperrobjekten und nichtatomaren Anzeigeobjekten steuert. Dies wird weiter unten ausführlicher beschrieben.

[0209] Ausgesetzte Verknüpfungssteuerung (suspended_link_control): Dies ist ebenfalls ein Zustandszähler-Objekt. Es ist das zentrale Element des Mechanismus für die Verknüpfungsübertragung, wenn dieser in Zusammenhang mit ausgesetzten Verknüpfungszeigern verwendet wird. Eine ausführlichere Beschreibung findet sich weiter unten.

[0210] Gesperrte Entfernungssteuerung (blocked_remove_control): Auch dies ist ein Zustandszähler-Objekt. Es ist das zentrale Element des Mechanismus für die Verknüpfungsübertragung in Zusammenhang mit dem Entfernen einer Verknüpfung. Eine ausführlichere Beschreibung findet sich weiter unten.

[0211] Zählwert für die Prioritätserhöhung (priority_bump_count): In diesem Feld wird gezählt, wie oft die Priorität der Mutator-Aufgabe von wartenden Objekten erhöht wurde. Nachdem alle wartenden Objekte geweckt wurden, hebt der Mutator letztlich die Erhöhung seiner eigenen Priorität so oft auf, wie dieser Wert es angibt.

Szenario einer Interaktion zwischen nichtatomaren Anzeige- und Sperrobjekten

[0212] **Fig.** 11 zeigt ein Beispiel für die Interaktion der Anzeigeobjekte von zwei durchlaufenden Mutatoren sowie der Sperrobjekte von zwei sperrenden Mutatoren, die alle dieselben Verknüpfungszeiger verwenden. Das Szenario kommt wie folgt zustande:

1. Der Mutator, der das Anzeigeobjekt **1101** verwendet, erreichte den Vorwärtszeiger **1111** innerhalb von „Verknüpfung A" als Erster und machte „Verknüpfung B" zu seiner aktuellen Verknüpfung. Sein curPtr-Feld **1112** zeigt auf den Vorwärtszeiger **1111** innerhalb von „Verknüpfung A", während sein curLink-Feld **1113** auf „Verknüpfung B" zeigt.
2. Als Nächster traf der das Sperrobjekt **1103** verwendende Mutator ein, der eine Verknüpfung zwischen „Verknüpfung A" und „Verknüpfung B" einfügen wollte und den Vorwärtszeiger **1111** innerhalb von „Verknüpfung A" sperrte, indem er das dort befindliche Markierungsbit ×01 blocked_link_pointer aktiv setzte. Danach durchsuchte er alle Anzeigeobjekte, stellte fest, dass das Anzeigeobjekt **1101** mit demselben Vorwärts-Verknüpfungszeiger verbunden war, und setzte die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in curPtr **1112** aktiv, wodurch er sich dazu verpflichtete, darauf zu warten, dass der Iterator, der im Besitz des Anzeigeobjekts **1101** ist, vorrückte, und angab, dass er die Priorität der Iterator-Aufgabe erhöht hatte.
3. In der Zwischenzeit erreichte der das Anzeigeobjekt **1102** verwendende Mutator den Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" (wobei er sein curPtr-Feld **1114** so setzte, dass es darauf zeigte) und stellte fest, dass dieser von dem das Sperrobjekt **1103** verwendenden Mutator gesperrt wurde. Er setzte das Markierungsbit ×02 iterating_mutator_waiting im Vorwärtszeiger **1111** innerhalb von „Verknüpfung A" auf aktiv, um so anzugeben, dass ein (oder mehrere) durchlaufende(r) Mutator(en) wartete(n). Schließlich setzte er curLink **1115** so, dass es auf das Sperrobjekt **1103** zeigte, wobei er die Markierungsbits ×01 committed_waiter und ×04 prioriy_not_bumped aktiv setzte, um kenntlich zu machen, dass er sich dazu verpflichtet hatte, darauf zu warten, dass der das Sperrobjekt **1103** verwendende Mutator den Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" entsperrte, und dass er festgestellt hatte, dass es nicht notwendig war, die Priorität der Mutator-Aufgabe zu erhöhen.
4. Zuletzt wollte auch der das Sperrobjekt **1104** verwendende Mutator den Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" sperren, stellte jedoch fest, dass er bereits von dem Mutator gesperrt worden war, der das Sperrobjekt **1103** verwendete. Er setzte das Markierungsbit ×04 blocking_mutator_waiting im Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" aktiv, um anzugeben, dass ein (oder mehrere) sperrende(r) Mutator(en) wartete(n). Schließlich setzte er das Feld **1116** waiting_for_blocker innerhalb seines Sperrobjekts **1104** so, dass es auf das Sperrobjekt **1103** zeigte, und setzte die Markierungsbits ×01 committed_waiter und ×02 priority_bumped aktiv, um so kenntlich zu machen, dass er sich dazu verpflichtet hatte, darauf zu warten, dass der das Sperrobjekt **1103** verwendende Mutator seine Verknüpfungszeiger entsperrte, und dass er die Priorität der Mutator-Aufgabe erhöht hatte.

[0213] **Fig.** 11 zeigt den Zustand der Zeiger nach der Durchführung der obigen Schritte. Wenn der das Anzeigeobjekt **1101** verwendende Mutator später die Verarbeitung seiner aktuellen Verknüpfung abgeschlossen hat und zur nächsten Verknüpfung vorrückt, sieht er die Markierungsbits ×01 committed_waiter und ×02 priority_bumped, teilt dem das Sperrobjekt **1103** verwendenden Mutator (der den Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" sperrte) mit, dass er vorgerückt ist, und hebt die Prioritätserhöhung seiner eigenen Aufgabe auf. Der Mutator wacht daraufhin auf, schließt das Einfügen der neuen Verknüpfung ab und hebt schließlich die Sperre seiner Verknüpfungszeiger auf. Danach sieht er, dass sowohl durchlaufende als auch sperrende Mutatoren warten (da die Markierungsbits ×02 iterating_mutator_waiting und ×04 blocker_waiting im gesperrten Vorwärts-Verknüpfungszeiger **1111** aktiv gesetzt sind) und macht anhand ihrer Listen im SharedChain-Objekt ihre Anzeige- und Sperrobjekte ausfindig. Er gibt dem Anzeigeobjekt **1102** eine neue aktuelle Verknüpfung (und setzt dabei curLink **1115** auf denselben Wert, den er auch im Vorwärts-Verknüpfungszeiger **1111** innerhalb von „Verknüpfung A" gesetzt hat, als er dessen Sperre aufgehoben hat) und weckt seinen Mutator. Außerdem setzt er das Feld **1116** waiting_for_blocker im Sperrobjekt **1104** auf NULL und weckt seinen Mutator, der dann erneut versuchen kann, den Verknüpfungszeiger zu sperren. In der Zwi-

schenzeit macht der das Sperrobjekt **1103** verwendende Mutator die Prioritätserhöhung seiner eigenen Aufgabe rückgängig und hebt damit die Erhöhung der Priorität auf, die er von dem Mutator erhalten hat, der das Sperrobjekt **1104** verwendet.

SharedChain-Objekt

[0214] Neben den Zeigern auf die erste und letzte Verknüpfung in der Kette hat das SharedChain-Objekt eine Reihe von Feldern mit Zustandsinformationen, die folgendermaßen lauten:
Inaktiver Verknüpfungszeiger (inactive_link_pointer): Dieses Feld enthält den Wert NULL und ist an einer 8-Byte-Grenze ausgerichtet. Wenn ein Anzeigeobjekt inaktiv ist, enthält sein CurPtr-Feld **705** die Adresse dieses Felds, und curLink **706** ist NULL. Somit weist ein inaktives Anzeigeobjekt keine Besonderheit auf. Jeder sperrende Mutator, der es sieht, ignoriert es sofort, da der eindeutige Wert in curPtr **705** nie mit der Adresse eines Vorwärts-Verknüpfungszeigers übereinstimmt, der für diesen Mutator von Belang ist, so dass dies eine bequeme Art und Weise darstellt, wie das Anzeigeobjekt dem sperrenden Mutator Platz machen kann, ohne dass es als Sonderfall behandelt wird.

[0215] Anfang der Liste des ersten und des zweiten Sperrobjekts (primary_blocker_list_head/secondary_blocker_list_head): Diese Felder sind Zeiger auf das erste Sperrobjekt in der Liste der Sperrobjekte. Dabei ist nur jeweils eines von ihnen gültig. Die Liste der Sperrobjekte ist eine einfach verknüpfte Liste. Wenn ein neues Sperrobjekt erzeugt wird, wird mit der Funktion compareAndSwap() das neue Sperrobjekt am Anfang der Liste atomar eingefügt. Das Löschen eines nicht mehr benötigten Sperrobjekts zu einem späteren Zeitpunkt kann jedoch nicht anhand einer einzigen atomaren Aktualisierung des Kettenanfangs erfolgen, wodurch auch sichergestellt wird, dass die Sperrobjekte nach wie vor nicht benötigt werden; stattdessen ist eine einzige atomare Aktualisierung des Anzeiger notwendig, der angibt, welcher Kettenanfang gültig ist.

[0216] Anfang der Liste des ersten und des zweiten Anzeigeobjekts für Inspektoren (primary_inspector_view/list_head/secondary_inspector_view_list_head): Diese Felder sind Zeiger auf das erste Anzeigeobjekt in der Liste der Anzeigeobjekte für Inspektoren. Sie werden auf die gleiche Art und Weise verwaltet wie die Felder primary_blocker_list_head und secondary_blocker_list_head.

[0217] Anfang der Liste des ersten und des zweiten Anzeigeobjekts für Mutatoren (primary_mutator_view_list_head/secondary_mutator_view_list_head): Diese Felder sind Zeiger auf das erste Anzeigeobjekt in der Liste der Anzeigeobjekte für Mutatoren. Sie werden auf die gleiche Art und Weise verwaltet wie die Felder primary_blocker_list_head und secondary_blocker_list_head.

[0218] Zählwerte für aktive Sperr- und Anzeigeobjekte (activated_blocker_and_view_counts): Dieses Feld enthält eine einzige Ganzzahl, die als zwei separate Zähler für die Anzahl der gegenwärtig aktiven Sperr- und Anzeigeobjekte dient, wobei die höherwertigen Bits als Zähler für die Anzahl der aktivierten Sperrobjekte und die niederwertigen Bits als Zähler für die Anzahl der aktivierten Anzeigeobjekte (als Summe aus Inspektoren und Mutatoren) verwendet werden. Beide Teile werden als ein einziges atomares Feld aktualisiert. Dabei wird der Sperrobjekt-Teil um den Wert 1 erhöht, bevor ein Sperrobjekt einem Mutator zugewiesen wird, und um den Wert 1 erniedrigt, nachdem keine anderen Sperr- oder Anzeigeobjekte mehr auf das Sperrobjekt verweisen. Der Anzeigeobjekt-Teil wird um den Wert 1 erhöht, bevor ein Anzeigeobjekt einem Iterator (entweder einem Inspektor oder einem Mutator) zugewiesen wird, und um den Wert 1 erniedrigt, nachdem der Iterator zurückgesetzt wurde. Das höchstwertige Bit eines jeden Teils wird nicht als Teil des Zählwerts verwendet, sondern wird reserviert. Eines dieser reservierten Bits dient als Markierung um anzugeben, dass die Funktion reclaim() ausgeführt wird. (Rückfordern (reclaim) ist der Prozess, bei dem nicht benötigte Sperr- und/oder Anzeigeobjekte gelöscht werden). Dieser Prozess kann nur ausgelöst werden, wenn das gesamte Feld Null ist (d.h., wenn es keine aktiven Sperr- und Anzeigeobjekte gibt und der Reklamationsvorgang nicht bereits läuft).

[0219] Zählwerte für vorhandene Sperr- und Anzeigeobjekte (blocker_and_view_existence_counts): Dieses Feld enthält eine einzige Ganzzahl, die als zwei separate Zähler für die Anzahl der gegenwärtig vorhandenen Sperr- und Anzeigeobjekte verwendet wird, wobei die höherwertigen Bits als Zähler für die Anzahl der vorhandenen Sperrobjekte und die niederwertigen Bits als Zähler für die Anzahl der vorhandenen Anzeigeobjekte (als Summe aus Inspektoren und Mutatoren) verwendet werden. Die beiden Teile werden als ein einziges atomares Feld aktualisiert. Dabei wird der Sperrobjekt-Teil um den Wert 1 erhöht, wenn ein neues Sperrobjekt erzeugt werden muss (nachdem der Sperrobjekt-Teil des Felds activated_blocker_and_view_counts inkrementiert, aber bevor das Sperrobjekt erzeugt und am Anfang der Liste hinzugefügt wurde). Der Anzeigeobjekt-Teil wird um den Wert 1 erhöht, wenn ein neues Anzeigeobjekt erzeugt werden muss (nachdem der Anzeigeobjekt-Teil

des Felds activated_blocker_and_view_counts inkrementiert, aber bevor das Anzeigeobjekt erzeugt und am Anfang der betreffenden Liste hinzugefügt wurde). Beide Teile werden nur durch die Funktion reclaim() dekrementiert. Das höchstwertige Bit eines jeden Teils wird nicht als Teil des Zählwerts, sondern als Markierung verwendet. Die Markierung im Sperrobjekt-Teil gibt an, ob primary_blocker_list_head oder secondary_blocker_list_head gültig ist. Die Markierung im Anzeigeobjekt-Teil gibt an, ob das (unten beschriebene) Feld primary_view_existence_counts oder secondary_view_existence_counts gültig ist. Die Markierungen werden mit der Funktion reclaim() nach Bedarf hin- und hergeschaltet. Das Feld activated_blocker_and_view_counts und das Feld blocker_and_view_existence_counts können nicht gemeinsam als ein einziges atomares Feld aktualisiert werden. Sie liegen jedoch innerhalb desselben Quadwords, so dass sie mit der Funktion compareAndSwapQualified() einzeln aktualisiert werden können, wobei diese Funktion jedoch fehlschlägt, wenn eines der Felder zwischenzeitlich geändert wird.

[0220] Primäre/sekundäre Zählwerte für vorhandene Anzeigeobjekte (primary_view_existence_counts/secondary_view_existence_counts): Nur jeweils eines dieser Felder ist gültig. Um welches es sich handelt, wird durch die Markierung im Anzeigeobjekt-Teil des oben beschriebenen Felds blocker_and_view_existence_counts angegeben. Jedes dieser Felder enthält eine einzige Ganzzahl, die als zwei separate Zähler verwendet wird. Dabei werden die höherwertigen Bits eines jeden Felds als Zähler für die Anzahl der vorhandenen Anzeigeobjekte für Inspektoren und die niederwertigen Bits eines jeden Felds als Zähler für die Anzahl der vorhandenen Anzeigeobjekte für Mutatoren verwendet. Beide Teile werden als ein einziges atomares Feld aktualisiert. Der betreffende Zähler (für Inspektoren oder Mutatoren) wird um den Wert 1 erhöht, wenn ein neues Anzeigeobjekt erzeugt werden muss (nachdem der Summenzählwert im Feld blocker_and_view_existence_counts inkrementiert und nachdem das Anzeigeobjekt erzeugt und am Anfang der entsprechenden Liste eingefügt wurde). Beide Teile werden nur durch die Funktion reclaim() dekrementiert. Das höchstwertige Bit eines jeden Teils wird nicht als Teil des Zählwerts, sondern als Markierung verwendet. Die Markierung im Inspektor-Teil gibt an, ob das Feld primary_inspector_view_list_head oder das Feld secondary_inspector_view_list_head gültig ist. Die Markierung im Mutator-Teil gibt an, ob das Feld primary_mutator_view_list_head oder das Feld secondary_mutator_view_list_head gültig ist. Die Markierungen werden durch die Funktion reclaim() nach Bedarf hin- und hergeschaltet.

[0221] Zählwert der aktiven Anzeigeobjekte für Inspektoren (active_inspector_view_count): Dieses atomare Feld enthält einen Zählwert für die Anzahl der aktiven Anzeigeobjekte für nichtatomare Inspektoren. Es wird um den Wert 1 erhöht, bevor ein Anzeigeobjekt für einen Inspektor aus dem inaktiven Zustand in eine Verknüpfung in der Kette versetzt wird, und um den Wert 1 erniedrigt, nachdem das Anzeigeobjekt aus einer Verknüpfung in der Kette in den inaktiven Zustand zurückversetzt wird. Dieses Feld hat die Aufgabe, die Anzahl der Interaktionen, die Mutatoren mit den Listen von Anzeigeobjekten durchführen müssen, zu verringern: Wenn ein Mutator einen Verknüpfungszeiger sperrt und die Anzeigeobjekte für Inspektoren überprüfen muss, ist dies nur dann erforderlich, wenn dieses Feld ungleich Null ist. Der Wert dieses Felds kann für ein bestimmtes Anzeigeobjekt nur dann inkrementiert oder dekrementiert werden, nachdem der Anzeigeobjekt-Teil des Felds activated_blocker_view_counts für eben dieses Anzeigeobjekt inkrementiert wurde und bevor er für das betreffende Anzeigeobjekt dekrementiert wurde.

[0222] Zählwert der aktiven Anzeigeobjekte für Mutatoren (active_mutator_view_count): Dieses atomare Feld enthält einen Zählwert für die Anzahl der aktiven Anzeigeobjekt für nichtatomare Mutatoren. Es wird um den Wert 1 erhöht, bevor ein Anzeigeobjekt für einen Mutator aus dem inaktiven Zustand in eine Verknüpfung in der Kette versetzt wird, und um den Wert 1 erniedrigt, nachdem das Anzeigeobjekt von einer Verknüpfung in der Kette in den inaktiven Zustand zurückversetzt wird. Dieses Feld hat die Aufgabe, die Anzahl der Interaktionen, die Mutatoren mit den Listen von Anzeigeobjekten durchführen müssen, zu verringern: Wenn ein Mutator einen Verknüpfungszeiger sperrt und die Anzeigeobjekte für Mutatoren überprüfen muss, ist dies nur dann erforderlich, wenn dieses Feld ungleich Null ist. Der Wert dieses Felds kann für ein bestimmtes Anzeigeobjekt nur dann erhöht oder erniedrigt werden, nachdem der Anzeigeobjekt-Teil des Felds activated_blocker_and_view_counts für eben dieses Anzeigeobjekt inkrementiert wurde und bevor er für das betreffende Anzeigeobjekt dekrementiert wurde.

Durchlauf (iterating)

[0223] In der Umgebung der bevorzugten Ausführungsform können die Felder curLink **706** und curPtr **705** innerhalb des Anzeigeobjekts nicht atomar als ein einziges Feld aktualisiert werden. Da sie demnach einzeln aktualisiert werden müssen und da sperrende Mutatoren diese Felder jederzeit einsehen können, müssen alle Aktualisierungen so erfolgen, dass der Zustand des Anzeigeobjekts bei einer Überprüfung von curLink und curPtr zu jedem Zeitpunkt genau bestimmt werden kann. Beide Felder liegen innerhalb desselben Quadworts,

so dass eine Funktion compareAndSwapQualified(), mit der eines der Felder aktualisiert werden soll, fehlschlägt, wenn eines der beiden Felder zwischenzeitlich aktualisiert wird.

Nichtatomarer Iterator in inaktivem Zustand

[0224] Wenn sich ein nichtatomarer Iterator (Inspektor oder Mutator) im inaktiven Zustand befindet, verfügt er über kein Anzeigeobjekt. Alle Anzeigeobjekte, die zu diesem Zeitpunkt keinem Iterator zugewiesen sind, werden ebenfalls als inaktiv betrachtet. Bei einem inaktiven Anzeigeobjekt zeigt curPtr **705** auf das Feld inactive_link_pointer innerhalb des SharedChain-Objekts, während curLink auf NULL gesetzt ist.

[0225] Ein Anzeigeobjekt, das zu einem Mutator gehört, ist während einer Änderungsfunktion (in der Regel) ebenfalls inaktiv. Seine Felder curLink **706** und curPtr **705** sind wie oben beschrieben gesetzt, und es wird keine Inkrementierung der Felder active_inspector_view_count oder active_mutator_view_count für dieses Objekt vorgenommen.

Vorrücken eines nichtatomaren Iterators (Inspektor oder Mutator) zur ersten Verknüpfung in der Kette

[0226] Es wird davon ausgegangen, dass sich der Iterator zu Beginn der Verarbeitung im inaktiven Zustand befindet. Die folgenden Schritte sind durchzuführen:

1. Die Präemptiv-Funktion wird deaktiviert.
2. Der Anzeigeobjekt-Teil des Felds activated_blocker_and_view_counts wird um den Wert 1 erhöht. Ein verfügbares Anzeigeobjekt in der betreffenden Liste wird gesucht. Falls keine Objekte vorhanden sind, wird der Anzeigeobjekt-Teil des Felds blocker_and_view_existence_counts inkrementiert, ein neues Anzeigeobjekt wird erzeugt, das neue Objekt wird zum Anfang der jeweiligen Liste hinzugefügt und der Inspektor- oder Mutator-Teil (je nachdem, was zutreffend ist) des Felds primary_view_existence_counts oder secondary_view_existence_counts wird inkrementiert (je nachdem, welches Feld gültig ist).
3. Das Feld active_inspector_view_count oder active_mutator_view_count (je nachdem, was zutreffend ist) wird um den Wert 1 erhöht.
4. Das curLink-Felds **706** des Anzeigeobjekts wird auf den speziellen Übergangswert (000 ... 002) gesetzt. Da zu diesem Zeitpunkt das curPtr-Feld **705** auf das Feld inactive_link_pointer innerhalb des SharedChain-Objekts zeigt, hat kein Mutator Interesse an dem Anzeigeobjekt. Da dies jedoch im nächsten Schritt relevant wird, muss die Einstellung zunächst vorgenommen werden.
5. Das curPtr-Feld **705** des Anzeigeobjekts wird so geändert, dass es auf das Feld first_link_pointer innerhalb des SharedChain-Objekts zeigt. Für einen Mutator, der den Verknüpfungszeiger sperrt oder entsperrt, ist dieses Anzeigeobjekt nun von Belang. Der spezielle Übergangswert in curLink **706** informiert den Mutator darüber, dass der Iterator dabei ist, curLink auf den Wert zu setzen, den er durch die Dereferenzierung von curPtr erhalten hat. Der Mutator kann nicht wissen, ob die Dereferenzierung bereits stattgefunden hat und – falls sie bereits stattgefunden hat – ob sie vor oder nach der Sperrung bzw. Entsperrung des Verknüpfungszeigers durch den Mutator erfolgte (je nachdem, welche Operation dieser durchführt). Somit weiß der Mutator nicht, auf welchen Wert der Iterator curLink setzen wird. Um dies zu verhindern, versucht der Mutator, curLink so zu ändern, dass es den speziellen Nichtübereinstimmungswert (00 ... 04) enthält. Wenn der Mutator erfolgreich ist, schlägt der Versuch des Iterators fehl, den durch die Dereferenzierung von curPtr erhaltenen Wert in curLink einzusetzen, und er dereferenziert curPtr erneut. Auf diese Weise kann der Mutator sicher sein, dass der Iterator die Änderung sieht, die er an dem Verknüpfungszeiger vorgenommen hat.
6. Das Vorrücken zur ersten Verknüpfung in der Kette ist ein Sonderfall des eher allgemein ausgelegten Prozesses, mit dem von einer Verknüpfung zur nächsten vorgerückt wird. An dieser Stelle sind die Schritte des frontseitigen Sonderfalls abgeschlossen, und die Verarbeitung kann mit dem allgemeiner ausgelegten Prozess fortfahren. Wenn es sich um einen Inspektor handelt, fährt die Verarbeitung mit Schritt 3 aus Vorrücken eines nichtatomaren Inspektors zur nächsten Verknüpfung (siehe unten) fort. Bei einem Mutator fährt die Verarbeitung mit Schritt 4 aus Vorrücken eines nichtatomaren Mutators zur nächsten Verknüpfung fort (siehe unten).

[0227] (Das Vorrücken eines Iterators zu der Verknüpfung, die auf eine frei wählbare Verknüpfung folgt, läuft ähnlich ab. Hierfür wird im oben beschriebenen Schritt 5 curPtr nicht so gesetzt, dass es auf first_link_pointer innerhalb des SharedChain-Objekts zeigt, sondern so, dass es auf den Vorwärts-Verknüpfungszeiger innerhalb der frei wählbaren Verknüpfung zeigt.)

Vorrücken eines nichtatomaren Inspektors zur nächsten Verknüpfung

[0228] Während ein Inspektor über eine aktuelle Verknüpfung verfügt, zeigt das curLink-Feld innerhalb seines Anzeigeobjekts auf die Verknüpfung und das curPtr-Feld auf den Vorwärtszeiger innerhalb der aktuellen Verknüpfung. Beim Vorrücken erreichen beide Felder die gleichen Positionen relativ zueinander, jedoch bezogen auf die nächste Verknüpfung in der Kette. Dabei kann der Vorwärts-Verknüpfungszeiger, auf den curPtr zeigt, zu dem betreffenden Zeitpunkt gesperrt sein. Die folgenden Schritte werden durchgeführt:

1. Die Präemptiv-Funktion wird deaktiviert.
2. curLink wird auf den speziellen Übergangswert gesetzt. Wenn ein Mutator, der den Verknüpfungszeiger, auf den curPtr zeigt, sperrt oder entsperrt, diesen Wert sieht, weiß er, dass der Inspektor dabei ist, curLink auf den Wert zu setzen, den er durch die Dereferenzierung von curPtr erhalten hat. Der Mutator kann nicht wissen, ob die Dereferenzierung bereits stattgefunden hat und – falls sie bereits stattgefunden hat – ob sie vor oder nach der Sperrung bzw. Entsperrung des Verknüpfungszeigers durch den Mutator erfolgte (je nachdem, welche Operation dieser durchführt). Somit weiß der Mutator nicht, auf welchen Wert der Inspektor curLink setzen wird. Um dies zu verhindern, versucht der Mutator, curLink so zu ändern, dass es den speziellen Nichtübereinstimmungswert enthält. Wenn der Mutator erfolgreich ist, schlägt der Versuch des Inspektors fehlt, curLink in Schritt 8 zu setzen, und er dereferenziert curPtr erneut. Auf diese Weise kann der Mutator sicher sein, dass der Inspektor die Änderung sieht, die er an dem Verknüpfungszeiger vorgenommen hat.
3. curPtr wird dereferenziert, um den Inhalt des Verknüpfungszeigers zu erhalten. Wenn er nicht gesperrt ist, wird mit Schritt 8, andernfalls wird mit Schritt 4 fortgefahren.
4. Der Verknüpfungszeiger ist gesperrt und enthält einen Zeiger auf das Sperrobjekt, das zu dem Mutator gehört, der es gesperrt hat. Das Feld activation_id wird von dem Sperrobjekt erhalten und für eine mögliche spätere Verwendung gespeichert. Wenn der Verknüpfungszeiger von einem Mutator gesperrt wurde, der die Verknüpfung mit dem gesperrten Verknüpfungszeiger entfernen möchte, wird das Feld secondary_next_link erhalten und für eine mögliche spätere Verwendung als nächster Verknüpfungszeiger gespeichert; andernfalls wird das Feld primary_next_link erhalten und stattdessen gespeichert.
5. Die Dereferenzierung von curPtr wird wiederholt. Wenn der Verknüpfungszeiger nicht mehr gesperrt ist, wird mit Schritt 8 fortgefahren.
6. Der Verknüpfungszeiger ist entweder noch gesperrt, oder er wurde erneut gesperrt. Das Feld activation_id wird von dem Sperrobjekt erhalten. Wenn activation_id nicht mit dem zuvor erhaltenen Wert übereinstimmt, wurde der Verknüpfungszeiger erneut gesperrt. In diesem Fall kehrt der Prozess zu Schritt 4 zurück.
7. Wenn der Wert für activation_id gleichgeblieben ist, bedeutet dies, dass das in Schritt 4 erhaltene Feld primary_next_link oder secondary_next_link der Zeiger auf die nächste Verknüpfung ist. (Dabei ist zu beachten, dass der Mutator, der den Verknüpfungszeiger gesperrt hat, in der Zwischenzeit die Sperre möglicherweise aufgehoben hat. Beim Entsperren setzt er den Verknüpfungszeiger auf den gleichen Wert, der aus dem Feld primary_next_link oder secondary_next_link erhalten wurde, so dass in diesem Fall der erhaltene Wert noch immer gültig ist. Wenn nach der Entsperrung des Verknüpfungszeigers durch den Mutator der Verknüpfungszeiger jedoch durch eine andere Sperrobjekt-Aktivierung erneut gesperrt wurde, ist der erhaltene Wert unter Umständen nicht mehr korrekt. Bevor der Mutator, der die andere Sperrobjekt-Aktivierung verwendet hat, seine Verknüpfungszeiger jedoch entsperren kann, sucht er nach Anzeigeobjekten, mit denen er interagieren muss, und sieht dieses, wobei curPtr auf den fraglichen Verknüpfungszeiger zeigt. Der Mutator wartet entweder darauf, dass der Inspektor, der dieses Anzeigeobjekt verwendet, vorrückt, oder er ändert curLink so, dass es den speziellen Nichtübereinstimmungswert enthält, um zu veranlassen, dass der Versuch des Inspektors, curLink in Schritt 8 zu setzen, fehlschlägt, wobei dies davon abhängig ist, ob der Versuch des Mutators oder Inspektors, curLink zu setzen, erfolgreich ist.)
8. Mit der Funktion compareAndSwap() wird versucht, den nächsten Verknüpfungszeiger in curLink zu setzen. Dies ist die neue aktuelle Verknüpfung des Inspektors und entweder der Wert, der bei der Dereferenzierung von curPtr in Schritt 3 oder Schritt 5 erhalten wurde, oder der Wert, der aus dem Feld primary_next_link oder secondary_next_link in Schritt 4 erhalten wurde. Die Funktion compareAndSwap() kann nur deshalb fehlschlagen, weil ein Mutator curLink von dem erwarteten Übergangswert zum Nichtübereinstimmungswert geändert hat. Bei einem Fehlschlag, kehrt der Prozess zu Schritt 2 zurück und fährt von dort fort.
9. Wenn die neue aktuelle Verknüpfung NULL ist, gibt sie an, dass das Ende der Kette erreicht wurde. In diesem Fall fährt die Verarbeitung mit Schritt 2 unter Zurücksetzen eines nichtatomaren Iterators (Inspektor oder Mutator) in den inaktiven Zustand fort.
10. Wenn die neue aktuelle Verknüpfung nicht NULL ist, wurde eine neue aktuelle Verknüpfung erzeugt. Nun wird mit compareAndSwap() versucht, curPtr zu ändern, um den Vorwärtszeiger innerhalb der neuen aktuellen Verknüpfung ausfindig zu machen, während der alte Wert von curPtr bekannt ist. Wenn das Mar-

kierungsbit ×04 remove_waiting im alten Wert von curPtr aktiv ist, werden auch die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in den neuen Wert von curPtr aufgenommen, wenn dieser gesetzt wird. Wenn der Versuch fehlschlägt, wird dieser Schritt so lange wiederholt, bis er erfolgreich ist.

11. Wenn das Markierungsbit ×01 committed_waiter in dem in Schritt 10 erhaltenen alten Wert von curPtr nicht aktiv ist, gibt es keinen Mutator, der darauf wartet, dass dieser Inspektor vorrückt. In diesem Fall wird mit Schritt 19, andernfalls wird mit Schritt 12 fortgefahren.

12. Das Markierungsbit ×01 committed_waiter im alten Wert von curPtr war aktiv, so dass ein Mutator darauf wartet, dass er über das Vorrücken unterrichtet wird. Wenn auch das Markierungsbit ×04 remove_waiting aktiv ist, muss die Benachrichtigung über das Vorrücken auf das nächste Vorrücken verzögert werden. In diesem Fall wird mit Schritt 17, andernfalls wird mit Schritt 13 fortgefahren.

13. Das Markierungsbit ×04 remove_waiting ist inaktiv, so dass der Mutator nun informiert werden muss. Der alte Wert von curPtr zeigt auf den Verknüpfungszeiger, den der Mutator gesperrt hat, so dass ein Zeiger auf das Sperrobjekt des Mutators von dem Verknüpfungszeiger erhalten und der Mutator über das Vorrücken unterrichtet wird.

14. Wenn das Markierungsbit ×02 priority_bumped im alten Wert von curPtr aktiv ist, hat der wartende Mutator angegeben, dass er die Priorität der Inspektor-Aufgabe erhöht hat. In diesem Fall fährt der Prozess mit Schritt 16 fort; andernfalls fährt er mit Schritt 15 fort.

15. Das Markierungsbit ×02 priority_bumped im alten Wert von curPtr ist inaktiv, was bedeutet, dass der Inspektor curPtr geändert hat, bevor der wartende Mutator angeben konnte, dass er die Priorität der Inspektor-Aufgabe erhöht hat. Um sicherzustellen, dass der Inspektor die Prioritätserhöhung seiner eigenen Aufgabe nicht aufhebt, bevor der Mutator sie erhöhen kann, wird auf priority_bump_uncertainty_resolver (Sperrobjekt für eine einzige Aufgabe) innerhalb des Anzeigeobjekts gewartet. (Wenn der Mutator versucht, das Markierungsbit ×02 priority_bumped in curPtr aktiv zu setzen, und sieht, dass curPtr geändert wurde, entsperrt er priority_bump_resolver für das Anzeigeobjekt.) Wenn der wartende Inspektor aufwacht, kann er sicher sein, dass die Priorität seiner Aufgabe erhöht wurde.

16. Die Prioritätserhöhung der Inspektor-Aufgabe wird aufgehoben. Der Prozess fährt mit Schritt 19 fort.

17. An dieser Stelle gibt es einen wartenden Mutator, der benachrichtigt werden muss, wobei allerdings das Markierungsbit ×04 remove_waiting im alten Wert von curPtr aktiv ist, so dass die Benachrichtigung bis zum nächsten Vorrücken verzögert wird. (Die Markierungsbits wurden in Schritt 10 entsprechend gesetzt.) Wenn im alten Wert von curPtr auch das Markierungsbit ×02 priority_bumped aktiv ist, hat der wartende Mutator angegeben, dass er die Priorität der Inspektor-Aufgabe erhöht hat. In diesem Fall wird mit Schritt 19, andernfalls wird mit Schritt 18 fortgefahren.

18. Das Markierungsbit ×02 priority_bumped im alten Wert von curPtr ist inaktiv, was bedeutet, dass der Inspektor curPtr geändert hat, bevor der wartende Mutator angeben konnte, dass er die Priorität der Inspektor-Aufgabe erhöht hat. Um sicherzustellen, dass der Inspektor die Prioritätserhöhung seiner eigenen Aufgabe nicht aufhebt, bevor der Mutator sie erhöhen kann, wird auf priority_bump_resolver (Sperrobjekt für eine einzige Aufgabe) innerhalb des Anzeigeobjekts gewartet. (Wenn der Mutator versucht, das Markierungsbit ×02 priority_bumped in curPtr aktiv zu setzen, und sieht, dass curPtr geändert wurde, entsperrt er priority_bump_resolver für das Anzeigeobjekt.) Wenn der wartende Inspektor aufwacht, kann er sicher sein, dass die Priorität seiner Aufgabe erhöht wurde. (Wenn er das nächste Mal vorrückt, sieht er das Markierungsbit ×02 priority_bumped in dem Wert, der zu diesem Zeitpunkt der alte (in Schritt 10 gesetzte) Wert von curPtr ist, und weiß, dass er die Prioritätserhöhung seiner Aufgabe aufheben soll.)

19. And dieser Stelle ist das Vorrücken abgeschlossen, und die Präemptiv-Funktion wird aktiviert.

Vorrücken eines nichtatomaren Mutators zur nächsten Verknüpfung

[0229] Während ein Mutator über eine aktuelle Verknüpfung verfügt, zeigt das curLink-Feld innerhalb seines Anzeigeobjekts auf diese Verknüpfung, während das curPtr-Feld auf den Vorwärtszeiger innerhalb der vorherigen Verknüpfung zeigt. Beim Vorrücken erreichen beide Felder die gleichen Positionen relativ zueinander, jedoch eine Verknüpfung weiter vorne in der Kette. Dabei kann der Vorwärts-Verknüpfungszeiger, auf den curtPtr zeigt, gesperrt sein. (Neben der Verwendung durch nichtatomare Mutatoren kann dieser Codepfad auch durch einen nichtatomaren Inspektor verwendet werden, wenn er wieder in den inaktiven Zustand zurückversetzt wird. Allerdings trifft ein Inspektor, während er wieder in den inaktiven Zustand versetzt wird, nie auf einen gesperrten Verknüpfungszeiger. Die bis hierher erläuterten Pfade können ausschließlich von Inspektoren oder Mutatoren verwendet werden und werden für einen Iterator beschrieben; die ausschließlich auf Mutatoren anwendbaren Pfade werden dagegen für einen durchlaufenden Mutator beschrieben.)

1. Die Präemptiv-Funktion wird deaktiviert. Der Prozess beginnt lediglich für die Funktion getNextLink() mit diesem Schritt (Schritt 1); alle anderen Fälle setzen bei späteren Schritten ein.

2. Anhand der. Funktion compareAndSwap() wird versucht, curPtr so zu ändern, dass das Feld auf den Vorwärtszeiger innerhalb der vorhandenen aktuellen Verknüpfung zeigt, während gleichzeitig der alte Wert von

curPtr bekannt ist. Wenn das Markierungsbit ×04 remove_waiting im alten Wert von curPtr aktiv gesetzt ist und dieser Codepfad für die Funktion getNextLink() aufgerufen wird, werden auch die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in den neuen Wert von curPtr aufgenommen, wenn dieser gesetzt wird. Dieser Schritt wird so lange wiederholt, bis er erfolgreich ist, da ein sperrender Mutator möglicherweise Markierungen in curPtr setzt. (Wenn der Prozess nicht für eine Funktion getNextLink() durchgeführt wird, darf eine eventuell notwendige Benachrichtigung des vorrückenden Iterators nicht bis zu einem späteren Vorrücken verzögert werden.)

3. curLink wird auf den speziellen Übergangswert gesetzt. Wenn ein sperrender Mutator, der den Verknüpfungszeiger, auf den curPtr zeigt, sperrt oder ensperrt, diesen Wert in curLink sieht, weiß er, dass dieser Iterator dabei ist, curLink auf den Wert zu setzen, den er durch die Dereferenzierung von curPtr erhalten hat. Der sperrende Mutator kann nicht wissen, ob die Dereferenzierung bereits stattgefunden hat und – falls sie bereits stattgefunden hat – ob sie vor oder nach seiner Sperrung bzw. Entsperrung des Verknüpfungszeigers erfolgte (je nachdem, welche Operation er durchführt). Somit weiß der sperrende Mutator nicht, auf welchen Wert der Iterator curLink setzen wird. Aus diesem Grund versucht der sperrende Mutator, curLink so zu ändern, dass es den speziellen Nichtübereinstimmungswert enthält. Wenn der sperrende Mutator erfolgreich ist, kann der Iterator in Schritt 6 curLink nicht setzen, unternimmt einen erneuten Versuch und dereferenziert curPtr noch einmal. Auf diese Weise kann der sperrende Mutator sicher sein, dass der Iterator die Änderung sieht, die der sperrende Mutator an dem Verknüpfungszeiger vorgenommen hat.

4. curPtr wird dereferenziert, um den Inhalt des Verknüpfungszeigers zu erhalten. Wenn er nicht gesperrt ist, wird mit Schritt 6 fortgefahren. Wenn er dagegen durch einen Mutator gesperrt ist, der die Verknüpfung mit dem gesperrten Verknüpfungszeiger entfernen wird (Markierungsbit ×08 remove_pending aktiv und Markierungsbit ×04 blocking_mutator_waiting inaktiv), wird mit Schritt 5 fortgefahren. Andernfalls wird der Verknüpfungszeiger gesperrt, wobei jedoch die ihn enthaltende Verknüpfung durch den Mutator, der ihn zu diesem Zeitpunkt gesperrt hat, nicht entfernt wird. Wenn das Markierungsbit ×02 iterating_mutator_waiting in dem gesperrten Verknüpfungszeiger bereits aktiv ist, wird mit Schritt 6 fortgefahren. Andernfalls ist das Markierungsbit ×02 iterating_mutator_waiting in dem gesperrten Verknüpfungszeiger inaktiv, so dass mit der Funktion compareAndSwap() versucht wird, es aktiv zu setzen. Wenn dies fehlschlägt, kehrt der Prozess zu Schritt 3 zurück und beginnt von dort noch einmal. Wenn das Markierungsbit aktiv gesetzt werden konnte, sieht der sperrende Mutator, wenn er seine Verknüpfungszeiger entsperrt, diese Markierung und weiß, dass er die Mutator-Anzeigeobjekte überprüfen muss, um etwaige wartende Mutatoren aufzuwecken. Der Prozess fährt mit Schritt 6 fort.

5. An dieser Stelle wird der Verknüpfungszeiger durch einen Mutator gesperrt, der die Verknüpfung entfernt, die den gesperrten Verknüpfungszeiger enthält. Der sperrende Mutator hat den Zeiger auf die nächste Verknüpfung im Feld secondary_next_block innerhalb seines Sperrobjekts gespeichert, als er den Verknüpfungszeiger gesperrt hat, so dass nun dieser Wert erhalten wird, um ihn als nächsten Verknüpfungszeiger zu verwenden. (Der sperrende Mutator wartet (letztlich) darauf, dass dieser durchlaufende Mutator vorrückt, so dass es sicher ist, den nächsten Verknüpfungszeiger aus seinem Verknüpfungsobjekt zu holen.)

6. Mit der Funktion compareAndSwap() wird versucht, curLink von dem erwarteten Übergangswert zu dem Wert zu ändern, der in Schritt 4 aus curPtr dereferenziert (oder in Schritt 5 aus dem Sperrobjekt erhalten) wurde. (Wenn der Verknüpfungszeiger jedoch gesperrt wurde (Markierungsbit ×01 blocked_link_pointer ist aktiv), dann ist der zu speichernde Wert ein Zeiger auf das Sperrobjekt, so dass alle Markierungsbits mit Ausnahme von ×01 inaktiv gesetzt werden, wenn sie in curLink gespeichert werden. Das Markierungsbit ×01 in curLink lautet committed_waiter; es gibt an, dass sich der durchlaufende Mutator nun verpflichtet hat, darauf zu warten, dass der sperrende Mutator seine Verknüpfungszeiger entsperrt.) Wenn der Versuch fehlschlägt (da ein Mutator curLink zwischenzeitlich zu dem speziellen Nichtübereinstimmungswert geändert hat), kehrt der Prozess zu Schritt 3 zurück und fährt von dort fort.

7. An dieser Stelle hat der Iterator seine alte aktuelle Verknüpfung erfolgreich verlassen. Wenn im alten Wert von curPtr (aus Schritt 2) das Markierungsbit ×01 committed_waiter nicht aktiv ist, gibt es keinen sperrenden Mutator, der darauf wartet, über das Vorrücken dieses Mutators unterrichtet zu werden. In diesem Fall wird mit Schritt 15, andernfalls wird mit Schritt 8 fortgefahren.

8. Ein sperrender Mutator wartet darauf, über das Vorrücken unterrichtet zu werden. Wenn das Markierungsbit ×04 remove_waiting aktiv ist und dieser Prozess als Teil einer Funktion getNextLink() ausgeführt wird, wird die Benachrichtigung über das Vorrücken bis zum nächsten Vorrücken verzögert. In diesem Fall wird mit Schritt 13, andernfalls wird mit Schritt 9 fortgefahren.

9. Wenn das Markierungsbit ×04 remove_waiting inaktiv ist (oder der Iterator keine Funktion getNextLink() verarbeitet), wird der wartende Mutator zu diesem Zeitpunkt unterrichtet. Der alte Wert von curPtr zeigt auf den Verknüpfungszeiger, den der Mutator gesperrt hat, so dass ein Zeiger auf das Sperrobjekt des Mutators von dem Verknüpfungszeiger erhalten und der Mutator über das Vorrücken unterrichtet wird.

10. Wenn das Markierungsbit ×02 priority_bumped im alten Wert von curPtr aktiv ist, hat der wartende Mutator angegeben, dass er die Priorität der Iterator-Aufgabe erhöht hat. In diesem Fall fährt der Prozess mit

Schritt 12 fort; andernfalls fährt er mit Schritt 11 fort.
11. Wenn das Markierungsbit ×02 priority_bumped im alten Wert von curPtr inaktiv ist, bedeutet dies, dass der Iterator curPtr geändert hat, bevor der wartende Mutator angeben konnte, dass er die Priorität der Iterator-Aufgabe erhöht hat. Um sicherzustellen, dass der Iterator die Prioritätserhöhung seiner eigenen Aufgabe nicht aufhebt, bevor der Mutator sie erhöhen kann, wird auf priority_bump_uncertainty_resolver (Sperrobjekt für eine einzige Aufgabe) innerhalb des Anzeigeobjekts gewartet. (Wenn der Mutator versucht, das Markierungsbit ×02 priority_bumped in curPtr aktiv zu setzen, und sieht, dass curPtr geändert wurde, entsperrt er priority_bump_resolver für das Anzeigeobjekt.) Wenn der wartende Iterator aufwacht, kann er sicher sein, dass die Priorität seiner Aufgabe erhöht wurde.
12. Die Prioritätserhöhung der Iterator-Aufgabe wird aufgehoben. Der Prozess fährt mit Schritt 15 fort.
13. An dieser Stelle gibt es einen warteten Mutator, der benachrichtigt werden muss, wobei die Benachrichtigung jedoch bis zum nächsten Vorrücken verzögert wird. (Die Markierungsbits wurden in Schritt 2 entsprechend gesetzt.) Wenn im alten Wert von curPtr auch das Markierungsbit ×02 priority_bumped aktiv ist, hat der wartende Mutator angegeben, dass er die Priorität der Iterator-Aufgabe erhöht hat. In diesem Fall wird mit Schritt 15 fortgefahren (dabei bemerkt der Iterator das Markierungsbit ×02 priority_bumped, wenn er das nächste Mal vorrückt); andernfalls wird mit Schritt 14 fortgefahren.
14. Wenn das Markierungsbit ×02 priority_bumped im alten Wert von curPtr inaktiv ist, bedeutet dies, dass der Iterator curPtr geändert hat, bevor der wartende Mutator angeben konnte, dass er die Priorität der Iterator-Aufgabe erhöht hat. Um sicherzustellen, dass der Iterator die Prioritätserhöhung seiner eigenen Aufgabe nicht aufhebt, bevor der Mutator sie erhöhen kann, wird auf priority_bump_resolver (Sperrobjekt für eine einzige Aufgabe) innerhalb des Anzeigeobjekts gewartet. (Wenn der Mutator versucht, das Markierungsbit ×02 priority_bumped in curPtr aktiv zu setzen, und sieht, dass curPtr geändert wurde, entsperrt er priority_bump_resolver für das Anzeigeobjekt.) Wenn der wartende Iterator aufwacht, kann er sicher sein, dass die Priorität seiner Aufgabe erhöht wurde. (Wenn er das nächste Mal vorrückt, sieht er das Markierungsbit ×02 priority_bumped in dem Wert, der zu diesem Zeitpunkt der alte (in Schritt 2 gesetzte) Wert von curPtr ist, und weiß, dass er die Prioritätserhöhung seiner Aufgabe aufheben soll.)
15. Wenn der in Schritt 6 in curLink gespeicherte Wert kein Zeiger auf ein Sperrobjekt war (Markierungsbit ×01 committed_waiter ist inaktiv), hat der Iterator sein Vorrücken abgeschlossen. In diesem Fall wird mit Schritt **17**, andernfalls wird mit Schritt 16 fortgefahren.
16. Der durchlaufende Mutator hat sich dazu verpflichtet, darauf zu warten, dass der Mutator, dessen Sperrobjekt-Adresse in curLink gespeichert ist, den Verknüpfungszeiger entsperrt, auf den curPtr zeigt. Die im Folgenden unter Warten und Wecken: Warten, bis ein Mutator seine Verknüpfungszeiger entsperrt beschriebenen Verarbeitungsschritte werden durchgeführt, um zu versuchen, die Priorität der Aufgabe des sperrenden Mutators zu erhöhen und darauf zu warten, dass er die Verknüpfungszeiger entsperrt. Wenn der sperrende Mutator die Verknüpfungszeiger entsperrt, setzt er curLink auf den gleichen Wert, den er in den (entsperrten) Vorwärtszeiger gesetzt hat, und weckt den durchlaufenden Mutator. Nachdem er geweckt wurde, hat der durchlaufende Mutator sein Vorrücken abgeschlossen.
17. Die Präemptiv-Funktion wird wieder aktiviert.

Zurücksetzen eines nichtatomaren Iterators (Inspektor oder Mutator) in den inaktiven Zustand

[0230] Ein Iterator wird in den inaktiven Zustand zurückversetzt (wobei auch sein Anzeigeobjekt inaktiv gesetzt und somit zur Wiederverwendung verfügbar gemacht wird), wenn dies von dem Client angefordert wird (bzw. wenn sein Destruktor ausgeführt wird). Außerdem wird ein Inspektor in den inaktiven Zustand zurückversetzt, wenn er zum Ende der Kette vorrückt. Schließlich wird das Anzeigeobjekt eines Mutators in den inaktiven Zustand zurückversetzt (wobei es nach wie vor dem Mutator zugehörig ist), wenn der Mutator eine Änderungsfunktion beginnt (das Anzeigeobjekt kann nach Abschluss der Änderungsfunktion wieder in den aktiven Zustand zurückkehren). Die Prozessschritte lauten wie folgt:
1. Die Präemptiv-Funktion wird deaktiviert.
2. Die Verarbeitungsschritte unter Vorrücken eines nichtatomaren Mutators zur nächsten Verknüpfung werden beginnend mit Schritt 2 durchgeführt, wobei in Schritt 2 curPtr jedoch auf die Adresse des Felds inactive_link_pointer innerhalb des SharedChain-Objekts (und nicht auf die Adresse des Vorwärtszeigers innerhalb der aktuellen Verknüpfung) gesetzt wird. Dadurch wird das Anzeigeobjekt „aus der Kette entfernt", so dass es danach für kein Sperrobjekt von Interesse ist. (Da das Feld inactive_link_pointer den Wert NULL enthält, wenn es in Schritt 4 dereferenziert wird, entsteht nie der Eindruck, dass es gesperrt ist. Somit entfällt das Warten auf die Entsperrung von Verknüpfungszeigern, obwohl ein sperrender Mutator unter Umständen benachrichtigt werden muss, wenn der Iterator seine aktuelle Verknüpfung verlässt.)
3. Das Feld active_inspector_view_count oder das Feld active_mutator_view_count im SharedChain-Objekt (je nachdem, was zutreffend ist) wird um den Wert 1 erniedrigt.
4. Diese Verarbeitung findet statt, wenn lediglich das Anzeigeobjekt zurückgesetzt wird (und der Iterator

selbst im aktiven oder Übergangszustand verbleiben soll); andernfalls wird Schritt 5 durchgeführt.
5. Das Anzeigeobjekt steht zur Wiederverwendung durch einen anderen Iterator zur Verfügung. Der Anzeigeobjekt-Teil des Felds activated_blocker_and_view_counts im SharedChain-Objekt wird um den Wert 1 erniedrigt. Wenn das gesamte Feld nun Null ist, bedeutet dies, dass keine ausstehenden Verweise auf Anzeige- oder Sperrobjekte vorhanden sind, so dass die Reklamationsfunktion ausgeführt wird, um alle nicht mehr benötigten Anzeige- oder Sperrobjekte zu löschen.

Erstellen oder Wiederherstellen einer neuen aktuellen Verknüpfung für einen nichtatomaren Mutator

[0231] Mit Ausnahme der Funktionen insertAfterCurrentLink() und removeAfterCurrentLink() hat ein Mutator, während er eine Verknüpfung einfügt oder entfernt, entweder kein Anzeigeobjekt, oder sein Anzeigeobjekt wurde in den inaktiven Zustand zurückversetzt. Wenn die Funktion annähernd abgeschlossen ist, die Verknüpfungszeiger jedoch noch nicht gesperrt wurden, kann die aktuelle Verknüpfung des Mutators folgendermaßen erstellt bzw. wiederhergestellt werden:

1. Wenn der Mutator bereits über ein Anzeigeobjekt verfügt, wird mit Schritt 2 fortgefahren. Andernfalls wird ein Anzeigeobjekt wie folgt zugewiesen: (a) Der Anzeigeobjekt-Teil des Felds activated_blocker_and_view_counts im SharedChain-Objekt wird um den Wert 1 erhöht; (b) In der Liste der Mutator-Anzeigeobjekte wird ein verfügbares Anzeigeobjekt ausfindig gemacht. Wenn kein Anzeigeobjekt verfügbar ist, wird der Anzeigeobjekt-Teil des Felds blocker_and_view_existence_counts um den Wert 1 erhöht, das neue Anzeigeobjekt wird erzeugt, wobei es (je nach Gültigkeit unter Verwendung des primären oder sekundären Listenkopfes) an den Kopf der Liste mit den Mutator-Anzeigeobjekten gestellt wird, und der Mutator-Teil des Felds primary_view_existence_counts oder secondary_view_existence_counts (je nach Gültigkeit) wird um den Wert 1 erhöht.
2. Das Feld active_mutator_view_count im SharedChain-Objekt wird um den Wert 1 erhöht.
3. An dieser Stelle ist das curLink-Feld des Anzeigeobjekts gleich NULL, während sein curPtr-Feld auf das Feld inactive_link_pointer innerhalb des SharedChain-Objekts zeigt. Dieses Anzeigeobjekt ist für keinen Mutator, der es sieht, von weiterem Interesse.
4. curLink wird so geändert, dass es auf die Verknüpfung zeigt, auf die der Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung zeigen wird, wenn er später entsperrt wird. Beim Einfügen einer Verknüpfung handelt es sich hierbei um die neue eingefügte Verknüpfung; wenn eine Verknüpfung entfernt wird, handelt es sich um die darauffolgende Verknüpfung. Da curPtr gegenüber dem vorhergegangenen Schritt unverändert bleibt, ist das Anzeigeobjekt immer noch nicht von Interesse für einen Mutator.
5. curPtr wird so geändert, dass es auf den Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung zeigt. An dieser Stelle verfügt das Anzeigeobjekt über eine neue aktuelle Verknüpfung und unterscheidet sich für einen durchlaufenden Mutator, der ansonsten nach der Entsperrung der Verknüpfungszeiger zu dieser Verknüpfung vorrücken würde, in nichts von einem beliebigen anderen Anzeigeobjekt.

Ändern der aktuellen Verknüpfung eines nichtatomaren Mutators

[0232] Wenn eine der Änderungsfunktionen, die ein nichtatomarer Mutator in Zusammenhang mit seiner aktuellen Verknüpfung ausführt, fehlschlägt, liegt dies daran, dass ein zweiter Mutator den betreffenden Verknüpfungszeiger bereits gesperrt hat. Der zweite Mutator weiß nichts von der Absicht des ersten Mutators und sieht ihn lediglich als einen durchlaufenden Mutator, auf dessen Vorrücken er wartet. Der erste Mutator muss sich als durchlaufender Mutator verhalten, dem zweiten Mutator Platz machen und anschließend darauf warten, dass dieser den Verknüpfungszeiger entsperrt. Abhängig von dem jeweiligen Szenario verfügt der Mutator nach dem Warten unter Umständen über eine neue aktuelle Verknüpfung. Wenn er über eine neue aktuelle Verknüpfung verfügt, schlägt die ursprüngliche Funktion fehl, da sie auf der ursprünglichen aktuellen Verknüpfung beruhte; andernfalls kann ein erneuter Versuch zur Ausführung der Funktion unternommen werden. Die ursprüngliche Entfernungs- oder Einfügefunktion schlägt zwangsläufig fehl, da sie auf der ursprünglichen aktuellen Verknüpfung beruhte. Tabelle 14 beschreibt die Funktionen, die Ursachen für den Fehlschlag und die Maßnahmen, mit denen Abhilfe geschaffen werden kann.

Tabelle 14: Ursachen und Gegenmaßnahmen beim Fehlschlag einer Funktion

Fehlgeschlagene Funktion	Ursache	Gegenmaßnahme
insertBeforeCurrentLink() oder removeCurrentLink()	Der Vorwärtszeiger innerhalb der vorherigen Verknüpfung (auf den curPtr zeigt) wird von einem Mutator gesperrt, der die vorherige Verknüpfung entfernen möchte.	Positionierung hinter dem Vorwärtszeiger in der Verknüpfung vor der vorherigen Verknüpfung. Der Mutator hat diesen Verknüpfungszeiger ebenfalls gesperrt und hat im Feld previous_link_pointer im Sperrobjekt einen Zeiger darauf gespeichert. Nach dem Wecken hat sich curPtr geändert, während curLink unverändert ist. Wiederholen der fehlgeschlagenen Funktion.
	Der Vorwärtszeiger innerhalb der vorherigen Verknüpfung (auf den curPtr zeigt) wird von einem Mutator gesperrt, der die aktuelle Verknüpfung entfernen oder eine Verknüpfung vor ihr einfügen möchte.	Positionierung hinter dem Vorwärtszeiger in der vorherigen Verknüpfung, auf die curPtr zeigt. Nach dem Wecken ist curPtr unverändert, während curLink geändert wurde. Rückgabe einer Fehlermeldung an den Client.
insertAfterCurrentLink() oder removeAfterCurrentLink()	Der Vorwärtszeiger innerhalb der aktuellen Verknüpfung (auf die curLink zeigt) wird von einem Mutator gesperrt, der die aktuelle Verknüpfung entfernen möchte.	Positionierung hinter dem Vorwärtszeiger in der vorherigen Verknüpfung, auf die curPtr zeigt. Nach dem Wecken ist curPtr unverändert, während curLink geändert wurde. Rückgabe einer Fehlermeldung an den Client.

Wechsel zur Position hinter einem gesperrten Verknüpfungszeiger

1. Wenn das Markierungsbit ×02 iterating_mutator_waiting im gesperrten Verknüpfungszeiger, hinter dem eine Position eingenommen werden soll, nicht aktiv ist, wird mit der Funktion compareAndSwap() versucht, das Markierungsbit aktiv zu setzen. Wenn dies fehlschlägt, wird der Versuch so lange wiederholt, bis das Markierungsbit aktiv ist. Da sich der erste Mutator im Weg des zweiten Mutators befindet, kann der zweite Mutator den Verknüpfungszeiger erst dann entsperren, wenn der erste Mutator ihm Platz macht, so dass die Funktion compareAndSwap() einfach so lange wiederholt werden kann, bis sie erfolgreich ist.
2. Das curLink-Feld innerhalb des Anzeigeobjekts des ersten Mutators wird so geändert, dass es auf das Sperrobjekt des zweiten Mutators zeigt (wobei das Markierungsbit ×01 committed_waiter aktiv ist). Dies gibt an, dass der erste Mutator sich dazu verpflichtet hat, darauf zu warten, dass das zweite Mutator seine Verknüpfungszeiger entsperrt.
3. Wenn das Markierungsbit ×01 committed_waiter im curPtr-Feld aktiv ist, wird mit compareAndSwap() versucht, alle Markierungsbits (×07) inaktiv zu setzen, während gleichzeitig bekannt ist, welche Markierungsbits aktiv waren. Wenn der Versuch nicht erfolgreich ist (da der zweite Mutator das Markierungsbit ×02 priority_bumped aktiv setzt), wird dieser Schritt so lange wiederholt, bis er erfolgreich ist. (Wenn das Markierungsbits ×01 committed_waiter nicht aktiv ist, wird es vom zweiten Mutator auch nicht aktiv gesetzt, da aufgrund der in Schritt 2 vorgenommenen Änderung der Eindruck entsteht, dass der erste Mutator sich dazu verpflichtet hat, auf den zweiten Mutator zu warten. In diesem Fall verpflichtet sich der zweite Mutator nicht dazu, auf den ersten Mutator zu warten. Selbst wenn der zweite Mutator gerade dabei wäre, sich zum Warten zu verpflichten, würde dies aufgrund des Setzens von curLink in Schritt 2 fehlschlagen, und bei einem erneuten Versuch wurde der zweite Mutator merken, dass sich stattdessen der erste Mutator dazu verpflichtet hat, auf ihn zu warten.)
4. Wenn in Schritt 3 festgestellt wurde, dass das Markierungsbit ×01 committed_waiter im alten Wert von curPtr aktiv gesetzt war, gibt dies an, dass sich der zweite Mutator dazu verpflichtet hat, auf das Vorrücken des ersten Mutators zu warten, so dass nun der Zeitpunkt gekommen ist, ihn über das Vorrücken zu unterrichten. (Der zweite Mutator kann auch das Markierungsbit ×02 priority_bumped und ×04 remove_waiting aktiv gesetzt haben; allerdings wird das Markierungsbit ×04 remove_waiting ignoriert, da die Benachrichtigung über das Vorrücken nicht verzögert werden sollte.)
5. Es wird darauf gewartet, dass der zweite Mutator seine Verknüpfungszeiger entsperrt. Die unter Warten, bis ein Mutator seine Verknüpfungszeiger entsperrt genannten Verarbeitungsschritte werden durchgeführt,

um zu versuchen, die Priorität der Aufgabe des zweiten Mutators zu erhöhen und darauf zu warten, dass er die Verknüpfungszeiger entsperrt. Wenn der zweite Mutator seine Verknüpfungszeiger entsperrt, sieht er das Anzeigeobjekt des ersten Mutators, setzt sein curLink-Feld auf den neuen Verknüpfungszeiger und weckt den ersten Mutator. Nach dem Wecken verfügt der erste Mutator über seine neue aktuelle Verknüpfung.

Sperren eines Verknüpfungszeigers Sperren eines Verknüpfungszeigers ohne Warten

[0233] Mit dieser Funktion wird versucht, den Vorwärtszeiger in der vorherigen Verknüpfung zu sperren, wobei die folgenden Funktionen zum Einsatz kommen:
insertBeforeCurrentLink()
removeCurrentLink()
insertBeforeArbitraryLink(), nachdem der Rückwärtszeiger in der frei wählbaren Verknüpfung vorläufig gesperrt wurde;
removeArbitryLink(), nachdem der Rückwärtszeiger in der frei wählbaren Verknüpfung vorläufig gesperrt wurde;
insertLast(), nachdem der letzte Verknüpfungszeiger innerhalb des SharedChain-Objekts vorläufig gesperrt wurde.

[0234] Sie wird außerdem intern von anderen Sperrfunktionen verwendet, die sie in Logik für das Warten bzw. erneute Versuchen einhüllen.

[0235] Im Folgenden werden die Schritte für das Sperren eines Verknüpfungszeigers ausführlich beschrieben. Im Wesentlichen wird der alte Wert des Verknüpfungszeigers ausgelagert und an das aufrufende Objekt zurückgegeben, während der neue Wert (der bei aktiv gesetzten Markierungsbits auf das Sperrobjekt zeigt) eingelagert wird, sofern der Verknüpfungszeiger nicht bereits gesperrt wurde.

1. Der Verknüpfungszeiger wird dereferenziert, um seinen aktuellen Inhalt zu erhalten. Wenn er von einem anderen Mutator gesperrt wurde, endet der Prozess, und es wird der aktuelle Inhalt des Verknüpfungszeigers (ein Zeiger auf das Sperrobjekt des sperrenden Mutators) zurückgegeben, um einen Fehler anzuzeigen; andernfalls fährt der Prozess fort.
2. Wenn ein Vorwärts-Verknüpfungszeiger gesperrt wird (jedoch nicht innerhalb einer gerade entfernten Verknüpfung), wird der (in Schritt 1 erfasste) aktuelle Inhalt des Verknüpfungszeigers im Feld primary_next_link gespeichert. Wenn der Vorwärtszeiger innerhalb einer gerade entfernten Verknüpfung gesperrt wird, wird der aktuelle Inhalt des Verknüpfungszeigers im Feld secondary_next_link gespeichert. (Wenn dagegen ein Rückwärts-Verknüpfungszeiger gesperrt wird, gibt es keinen Inhalt, der im Sperrobjekt gespeichert werden müsste.)
3. Mit compareAndSwap() wird versucht, den Verknüpfungszeiger so zu ändern, dass er auf das Sperrobjekt zeigt (wobei die betreffenden Markierungsbits aktiv gesetzt sind), falls dies möglich ist, während gleichzeitig der aktuelle Inhalt des Verknüpfungszeigers bekannt ist. Wenn die Operation fehlschlägt, kehrt der Prozess zu Schritt 1 zurück und beginnt vor vorn.
4. Wenn der gerade gesperrte Verknüpfungszeiger der Vorwärts-Verknüpfungszeiger innerhalb einer gerade entfernten Verknüpfung war, wird das Feld secondary_next_link in das Feld primary_next_link kopiert.
5. Der Prozess endet und gibt den (in Schritt 1 erfassten) ursprünglichen Inhalt des Verknüpfungszeigers zurück. Die Tatsache, dass dieser auf eine Verknüpfung anstelle auf ein Sperrobjekt zeigt, gibt an, dass der Versuch einer Sperroperation erfolgreich war.

Sperren eines Verknüpfungszeigers mit Warten

[0236] Diese Funktion wird verwendet, wenn Rückwärtszeiger unter Verwendung der folgenden Funktionen vorläufig gesperrt werden:
insertLast()
insertBeforeArbitraryLink()
removeArbitraryLink()

[0237] Sie wird auch verwendet, um die Vorwärtszeiger in der vorherigen Verknüpfung unter Verwendung der folgenden Funktionen zu sperren:
insertFirst()
insertAfterCurrentLink()
insertAfterArbitraryLink()
removeFirst()

removeAfterArbitraryLink()

[0238] Zunächst wird das Feld waiting_for_blocker innerhalb des Sperrobjekts mit dem Wert NULL initialisiert, um anzugeben, dass der Mutator nicht wartet.

1. Es wird die weiter oben unter Sperren eines Verknüpfungszeigers ohne Warten beschriebene Prozedur aufgerufen, um zu versuchen, den Verknüpfungszeiger zu sperren. Wenn dies gelingt, endet der Prozess und gibt den ursprünglichen (entsperrten) Inhalt des Verknüpfungszeigers zurück; andernfalls wird mit Schritt 2 fortgefahren.
2. Der Sperrversuch ist fehlgeschlagen, da der Verknüpfungszeiger bereits von einem anderen Mutator gesperrt wurde, und der fehlgeschlagene Versuch hat einen Zeiger auf das Sperrobjekt des anderen Mutators zurückgegeben. Im Feld waiting_for_blocker innerhalb des ersten Sperrobjekts wird ein Zeiger auf das zweite Sperrobjekt gespeichert (wobei alle Markierungsbits inaktiv gesetzt sind). Die Tatsache, dass keine Markierungsbits aktiv gesetzt sind, zeigt an, dass der erste Mutator beabsichtigt, auf den zweiten Mutator zu warten, ohne sich dazu bereits verpflichtet zu haben.
3. Der zu sperrende Verknüpfungszeiger wird erneut dereferenziert. Wenn er nicht mehr von demselben Mutator gesperrt ist, kehrt der Prozess zu Schritt 1 zurück und beginnt vor vorn. (Durch die Rückkehr zu Schritt 1 bleibt das Feld waiting_for_blocker in diesem Übergangszustand, ohne sich nachteilig auszuwirken. Der Mutator, der das Sperrobjekt verwendet, auf den es zeigt, kann das Feld später auf NULL setzen, wenn er Verknüpfungszeiger entsperrt; wenn er es auf NULL setzt, führt er allerdings keine weiteren Aktionen durch, da keine Markierungsbits gesetzt sind.)
4. Wenn sich das Markierungsbit ×04 blocking_mutator_waiting nicht im gesperrten Verknüpfungszeiger befindet, wird mit compareAndSwap() versucht, es aktiv zu setzen. Wenn die Operation fehlschlägt, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn.
5. Mit compareAndSwap() wird versucht, das Markierungsbit ×01 committed_waiter im Feld waiting_for_blocker zu setzen, um so anzugeben, dass der erste Mutator sich nun verpflichtet hat, darauf zu warten, dass der zweite Mutator seine Verknüpfungszeiger entsperrt. Wenn der Versuch fehlschlägt, liegt dies daran, dass der zweite Mutator den Verknüpfungszeiger bereits entsperrt und das Feld waiting_for_blocker auf NULL zurückgesetzt hat. In diesem Fall kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn.
6. An dieser Stelle hat sich der erste Mutator dazu verpflichtet, auf den zweiten Mutator zu warten. Die unter Warten, bis ein Mutator seine Verknüpfungszeiger entsperrt beschriebenen Verarbeitungsschritte werden durchgeführt, um zu versuchen, die Priorität der Aufgabe des zweiten Mutators zu erhöhen und darauf zu warten, dass dieser die Verknüpfungszeiger entsperrt. Nachdem er geweckt wurde, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn.

Bedingtes Sperren eines Vorwärts-Verknüpfungszeigers

[0239] Mit dieser Funktion wird versucht, beim Einfügen oder Entfernen nach der aktuellen Verknüpfung den Vorwärtszeiger in der aktuellen Verknüpfung zu sperren.

1. Die oben beschriebene Prozedur Sperren eines Verknüpfungszeigers ohne Warten wird aufgerufen, um zu versuchen, den Verknüpfungszeiger zu sperren. Wenn dies erfolgreich ist, endet der Prozess und gibt den ursprünglichen (entsperrten) Inhalt des Verknüpfungszeigers zurück, um den Erfolg kenntlich zu machen; andernfalls fährt er mit Schritt 2 fort.
2. Der Sperrversuch ist fehlgeschlagen und der gesperrte Inhalt des Verknüpfungszeigers wurde zurückgegeben. Wenn der Verknüpfungszeiger von einem Mutator gesperrt wird, der die Verknüpfung mit dem gesperrten Zeiger entfernt (wobei das Markierungsbit ×08 remove_pending aktiv und das Markierungsbit ×04 blocking_mutator_waiting inaktiv gesetzt ist), endet der Prozess und gibt den ursprünglichen (ungesperrten) Inhalt des Verknüpfungszeigers zurück, um einen Fehlschlag anzugeben; andernfalls fährt er mit Schritt 3 fort.
3. Der Verknüpfungszeiger wird von einem zweiten Mutator gesperrt, der nicht beabsichtigt, die ihn enthaltende Verknüpfung zu entfernen. Die unter Sperren eines Verknüpfungszeigers mit Warten beschriebenen Schritte werden durchgeführt, um darauf zu warten, dass der Verknüpfungszeiger entsperrt wird. Wenn einer dieser Schritte fehlschlägt oder wenn der Mutator aus dem Wartezustand geweckt wird, kehrt der Prozess zum obigen Schritt 1 zurück und fährt von dort fort.

Sperren eines Vorwärts-Verknüpfungszeigers nach der vorläufigen Sperre eines Rückwärts-Verknüpfungszeigers

[0240] Mit dieser Funktion wird der Vorwärtszeiger in der vorherigen Verknüpfung gesperrt, wenn eine frei wählbare Verknüpfung entfernt wird oder eine Verknüpfung als letzte Verknüpfung bzw. vor einer frei wählba-

ren Verknüpfung eingefügt wird. Dabei wurde der Rückwärtszeiger bereits gesperrt, wobei die Markierung für die vorläufige Sperrung gesetzt wurde.

1. Die weiter oben unter Sperren eines Verknüpfungszeigers ohne Warten beschriebene Prozedur wird aufgerufen um zu versuchen, den Verknüpfungszeiger zu sperren. Wenn dies erfolgreich ist, fährt der Prozess mit Schritt 4 fort; andernfalls fährt er mit Schritt 2 fort.
2. Der Sperrversuch ist fehlgeschlagen und hat einen Zeiger auf das Sperrobjekt für den Mutator zurückgegeben, der den Verknüpfungszeiger bereits gesperrt hat. Der zweite Mutator sperrt denselben Verknüpfungszeiger, der zuvor vom ersten Mutator bereits vorläufig gesperrt wurde, und um eine gegenseitige Blockierung zu verhindern, wird die vorläufige Sperre ausgesetzt, so dass der zweite Mutator die Sperre verwenden kann. Für die Aussetzung der vorläufigen Sperre wird der Mechanismus für die Verknüpfungsübertragung mit dem Objekt suspended_link_control im Sperrobjekt des zweitens Mutators verwendet, um einen Zeiger auf die Verknüpfung, auf die der vorläufig gesperrte Rückwärtszeiger gezeigt hat, bevor er gesperrt war, an den zweiten Mutator weiterzuleiten.
3. Der Mechanismus für die Verknüpfungsübertragung wird mit dem Objekt suspended_link_control in dem Sperrobjekt des Mutators verwendet, dessen Sperrversuch fehlgeschlagen ist, um darauf zu warten, dass der zweite Mutator einen Zeiger auf die neue vorherige Verknüpfung zurückgibt. Danach kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn, indem er versucht, den Vorwärtszeiger innerhalb dieser neuen vorherigen Verknüpfung zu sperren.
4. An dieser Stelle wurde der Vorwärts-Verknüpfungszeiger erfolgreich gesperrt, und der Rückwärts-Verknüpfungszeiger, der vorläufig gesperrt war, muss dies nicht mehr sein. Mit compareAndSwap() wird versucht, das darin enthaltene Markierungsbit ×08 für die vorläufige Sperrung inaktiv zu setzen. Wenn die Funktion fehlschlägt, wird sie so lange wiederholt, bis sie erfolgreich ist. (Dabei könnte hier auch ein anderer Mutator das Markierungsbit ×04 blocking_mutator_waiting aktiv setzen.)
5. Die Prozedur wird beendet und der ursprüngliche (entsperrte) Inhalt des Verknüpfungszeigers zurückgegeben.

Sperren des Vorwärts-Verknüpfungszeigers innerhalb einer Verknüpfung, die gerade entfernt wird

1. Die oben unter Sperren eines Verknüpfungszeigers ohne Warten beschriebene Prozedur wird aufgerufen, um zu versuchen, den Verknüpfungszeiger zu sperren. Wenn sie erfolgreich ist, endet die Prozedur und gibt den ursprünglichen (entsperrten) Inhalt des Verknüpfungszeigers zurück; andernfalls fährt diese Prozedur mit Schritt 2 fort.
2. Der Sperrversuch ist fehlgeschlagen, da der Verknüpfungszeiger bereits von einem zweiten Mutator gesperrt wurde. Mit compareAndSwap() wird versucht, sowohl das Markierungsbit ×08 remove_pending als auch das Markierungsbit ×04 blocking_mutator in dem gesperrten Verknüpfungszeiger aktiv zu setzen. Wenn dies nicht erfolgreich ist, kehrt der Prozess zu Schritt 1 zurück, um von vorn zu beginnen; andernfalls fährt er mit Schritt 3 fort.
3. An dieser Stelle wird der Verknüpfungszeiger von einem zweiten Mutator gesperrt, und die Markierungen innerhalb des Verknüpfungszeigers wurden gesetzt, um anzugeben, dass ein Mutator, der die Verknüpfung mit dem gesperrten Verknüpfungszeiger entfernen möchte, darauf wartet, dass der Verknüpfungszeiger entsperrt wird. Diese Bedingung wird erkannt, wenn der zweite Mutator den Verknüpfungszeiger entsperrt. Der Mechanismus für die Verknüpfungsübertragung mit dem Objekt blocked_remove_control wird verwendet, um darauf zu warten, dass der zweite Mutator den Verknüpfungszeiger entsperrt. In diesem Fall hebt der zweite Mutator die Sperre jedoch nicht auf, sondern sperrt den Verknüpfungszeiger erneut für den wartenden Mutator (wobei die Sperrobjektadresse des wartenden Mutators mit den aktiv gesetzten Markierungsbits ×01 blocked_link_pointer und ×08 remove_pending darin gespeichert wird). Danach verwendet er den Mechanismus für die Verknüpfungsübertragung, um den wartenden Mutator zu wecken und den Wert an ihn weiterzuleiten, den er andernfalls in den Verknüpfungszeiger eingesetzt hätte, wenn er ihn tatsächlich gesperrt hätte.
4. Die Prozedur endet und gibt den Wert des entsperrten Verknüpfungszeigers zurück, der mit dem Mechanismus für die Verknüpfungsübertragung in Schritt 3 erhalten wurde.

Sperren des letzten Verknüpfungszeigers

[0241] Wenn eine Verknüpfung entfernt wird oder wenn eine Verknüpfung nach der aktuellen Verknüpfung oder nach einer frei wählbaren Verknüpfung eingefügt wird, wird mit dieser Funktion der Rückwärtszeiger innerhalb der nächsten Verknüpfung gesperrt. Dabei handelt es sich in diesen Fällen um den letzten Verknüpfungszeiger, der gesperrt wird.

1. Die oben unter Sperren eines Verknüpfungszeigers ohne Warten beschriebene Prozedur wird aufgerufen, um zu versuchen, den Verknüpfungszeiger zu sperren. Wenn sie erfolgreich ist, endet die Prozedur

und gibt den ursprünglichen (entsperrten) Inhalt des Verknüpfungszeigers zurück; andernfalls fährt sie mit Schritt 2 fort.
2. Der Versuch, den Verknüpfungszeiger zu sperren, ist fehlgeschlagen, da er bereits von einem zweiten Mutator gesperrt wurde. Wenn der Verknüpfungszeiger nicht vorläufig gesperrt wurde, fährt der Prozess mit Schritt 3 fort. Andernfalls wurde der Verknüpfungszeiger von dem zweiten Mutator vorläufig gesperrt, und der Mutator setzt die Sperre aus und verwendet den Mechanismus für die Verknüpfungsübertragung mit dem Objekt suspended_link_control innerhalb des Sperrobjekts des ersten Mutators, um einen Zeiger auf die vorherige Verknüpfung an ihn weiterzuleiten. Somit wird der Mechanismus für die Verknüpfungsübertragung mit dem Objekt suspended_link_control innerhalb des Sperrobjekts des ersten Mutators verwendet, um auf den Verknüpfungszeiger zu warten. Nach dem Wecken endet der Prozess und gibt einen Zeiger auf die vorherige Verknüpfung zurück.
3. An dieser Stelle wird der Verknüpfungszeiger durch den zweiten Mutator gesperrt, wobei dies allerdings nicht vorläufig ist. Mit den oben unter Sperren eines Verknüpfungszeigers mit Warten beschriebenen Schritten wird darauf gewartet, dass der Verknüpfungszeiger entsperrt wird. Wenn einer der Schritte fehlschlägt oder wenn der Mutator aus dem Wartezustand geweckt wird, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn.

Entsperren von Verknüpfungszeigern

[0242] Beim Entsperren von Verknüpfungszeigern werden zunächst die Vorwärtszeiger innerhalb der vorherigen Verknüpfung entsperrt, danach können die Rückwärtszeiger innerhalb der nächsten Verknüpfung entsperrt werden.

Entsperren des Vorwärtszeigers innerhalb der vorherigen Verknüpfung

1. Wenn die Markierungsbits ×04 blocking_mutator_waiting und ×08 remove_pending in dem zu entsperrenden Verknüpfungszeiger beide aktiv gesetzt sind, gibt dies an, dass ein Mutator, der die Verknüpfung mit dem gesperrten Vorwärtszeiger entfernen möchte, wartet. In diesem Fall fährt der Prozess mit Schritt 2 fort. Andernfalls wird mit compareAndSwap() versucht, den gesperrten Verknüpfungszeiger durch einen Zeiger auf seine neue nächste Verknüpfung zu ersetzen. Wenn der Versuch fehlschlägt, liegt dies daran, dass andere Mutatoren die Markierungsbits innerhalb des gesperrten Verknüpfungszeigers setzen, so dass der Prozess zum Anfang von Schritt 1 zurückkehrt und noch einmal von vorn beginnt. Wenn der Versuch erfolgreich ist, endet der Prozess und gibt den alten gesperrten Wert des Verknüpfungszeigers zurück, bei dem das Markierungsbit ×02 iterating_mutator_waiting und/oder das Markierungsbit ×04 blocking_mutator_waiting aktiv gesetzt sein können/kann, wodurch angegeben wird, ob nach möglicherweise vorhandenen wartenden Objekten gesucht werden muss.
2. An dieser Stelle wartet ein Mutator, der die Verknüpfung mit dem gesperrten Verknüpfungszeiger (anhand des Mechanismus für die Verknüpfungsübertragung) entfernen möchte, auf die Erlaubnis zum Fortfahren. Der Mutator hat außerdem den Rückwärtszeiger innerhalb der betreffenden Verknüpfung gesperrt. Der Rückwärtszeiger wird ausfindig gemacht und dient dazu, das von dem Mutator verwendete Sperrobjekt ausfindig zu machen.
3. Mit compareAndSwap() wird versucht, die Adresse des Sperrobjekts (mit aktiv gesetzten Markierungsbits ×01 blocked_link_pointer und ×08 remove_pending) in dem gesperrten Verknüpfungszeiger zu sperren. Wenn der Versuch fehlschlägt, liegt dies daran, dass andere durchlaufende Mutatoren das Markierungsbit ×02 iterating_mutator_waiting innerhalb des gesperrten Verknüpfungszeigers aktiv setzen, woraufhin dieser Schritt so lange wiederholt wird, bis er erfolgreich ist. Nachdem er erfolgreich durchgeführt wurde, scheint der Verknüpfungszeiger nun von dem wartenden Mutator gesperrt zu sein, der die Verknüpfung mit dem Verknüpfungszeiger entfernen möchte.
4. Anhand des Mechanismus für die Verknüpfungsübertragung mit dem Objekt blocked_remove_control innerhalb des Sperrobjekts des wartenden Mutators wird der Verknüpfungszeiger an den wartenden Mutator weitergeleitet, und dieser wird geweckt.
5. Der Prozess endet und gibt den alten gesperrten Wert des Verknüpfungszeigers zurück. Dabei ist sowohl das Markierungsbit ×02 iterating_mutator_waiting als auch das Markierungsbit ×04 blocking_mutator_waiting relevant, da hierdurch angegeben wird, dass nach möglicherweise vorhandenen wartenden Objekten gesucht werden muss.

Entsperren des Rückwärtszeigers innerhalb der nächsten Verknüpfung

1. Wenn der Verknüpfungszeiger vorläufig gesperrt wird, erfolgt dies tatsächlich durch einen anderen Mutator unter Verwendung eines anderen Sperrobjekts. In diesem Schritt fährt der Prozess mit Schritt 3 fort.

2. Mit compareAndSwap() wird versucht, den gesperrten Verknüpfungszeiger durch einen Zeiger auf seine neue vorherige Verknüpfung zu ersetzen. Wenn der Versuch fehlschlägt, liegt dies daran, dass ein anderer sperrender Mutator das Markierungsbit ×04 blocking_mutator_waiting innerhalb des gesperrten Verknüpfungszeigers setzt, woraufhin dieser Schritt so lange wiederholt wird, bis er erfolgreich ist. Nachdem er erfolgreich durchgeführt wurde, endet der Prozess und gibt den alten (gesperrten) Wert des Verknüpfungszeigers zurück, bei dem das Markierungsbit ×04 blocking_mutator_waiting aktiv gesetzt sein kann, um damit anzugeben, ob nach möglicherweise vorhandenen wartenden Objekten gesucht werden muss.
3. An dieser Stelle wird der Verknüpfungszeiger durch einen anderen Mutator vorläufig gesperrt. Der Mutator hat den gesperrten Verknüpfungszeiger ausgesetzt und verwendet den Mechanismus für die Verknüpfungsübertragung mit dem Objekt suspended_link_control innerhalb deines Sperrobjekts, um auf die Erlaubnis zum Fortfahren zu warten. Anhand des Mechanismus für die Verknüpfungsübertragung mit dem Objekt suspended_link_control wird der Zeiger auf die neue vorherige Verknüpfung an ihn weitergeleitet (wobei dieser denselben Wert aufweist, den andernfalls der Verknüpfungszeiger in Schritt 2 erhalten hätte). Dies weckt den wartenden Mutator und übergibt ihm einen neuen vorherigen Verknüpfungszeiger.
4. Der Prozess endet und gibt nichts zurück. Der Verknüpfungszeiger wurde nicht geändert. (Wenn das Markierungsbit ×04 blocking_mutator_waiting aktiv gesetzt ist, geschieht dies mit Blick auf den anderen Mutator, der den Verknüpfungszeiger vorläufig gesperrt hat und das Markierungsbit sieht, wenn er den Verknüpfungszeiger entsperrt.)

Interaktionen zwischen Sperr- und Anzeigeobjekten Suchen von Iteratoren, auf die gewartet werden muss, wenn eine Verknüpfung eingefügt wird

[0243] Nachdem die Verknüpfungszeiger, die für das Einfügen einer Verknüpfung benötigt werden, gesperrt wurden, wartet der Mutator, dass etwaige Iteratoren, die dem Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung zugewiesen sind, vorrücken. Die Inspektor-Anzeigeobjekte müssen überprüft werden, falls das Feld active_inspector_view_count im SharedChain-Objekt nicht gleich Null ist, während die Mutator-Anzeigeobjekte überprüft werden müssen, wenn das Feld active_mutator_view_count im SharedChain-Objekt nicht gleich Null ist. Wenn Anzeigeobjekte zu überprüfen sind, gibt die Überprüfung eines jeden Objekts einen booleschen Wert zurück, der angibt, ob der sperrende Mutator darauf warten muss, dass der Iterator vorrückt. Dabei steht die Anzahl der WAHR-Ergebnisse für die Anzahl der „Benachrichtigung bei Vorrücken", auf die der sperrende Mutator warten muss.

[0244] Für jedes zu überprüfende Anzeigeobjekt werden die folgenden Schritte durchgeführt:
1. Wenn das curPtr-Feld des Anzeigeobjekts nicht auf den Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung zeigt, ist das Anzeigeobjekt nicht weiter von Belang, so dass der Prozess endet und FALSCH zurückgibt.
2. Wenn das curLink-Feld des Anzeigeobjekts auf das Sperrobjekt dieses Mutators zeigt, hat sich der Iterator bereits dazu verpflichtet, darauf zu warten, dass der Mutator den Verknüpfungszeiger entsperrt. In diesem Fall ist das Anzeigeobjekt nicht weiter von Belang, so dass FALSCH zurückgegeben wird. (Dabei ist zu beachten, dass dieser Fall nie eintritt, wenn ein Anzeigeobjekt für einen Inspektor überprüft wird, da Inspektoren nicht auf sperrende Mutatoren warten.)
3. Wenn das curLink-Feld des Anzeigeobjekts (1) auf dieselbe Verknüpfung zeigt, auf die der Verknüpfungszeiger gezeigt hat, bevor er von dem Mutator gesperrt wurde, oder wenn es (2) auf ein anderes Sperrobjekt zeigt, ist das Anzeigeobjekt in beiden Fällen für den Mutator von Interesse. (Wenn ein Anzeigeobjekt für einen Inspektor überprüft wird, zeigt curLink nie auf ein Sperrobjekt, da Inspektoren nicht warten. Wenn dagegen curLink bei einem Mutator-Anzeigeobjekt auf ein anderes Sperrobjekt zeigt, bedeutet dies lediglich, dass der Mutator, der dieses Sperrobjekt verwendet, den Verknüpfungszeiger kürzlich entsperrt hat, jedoch noch nicht dazu kommen ist, curLink zu aktualisieren.) In beiden Fällen wird mit compareAndSwapQualified() versucht, das Markierungsbit ×01 committed_waiter im curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn dies fehlschlägt (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, hat sich der Mutator nun dazu verpflichtet, auf das Vorrücken des Iterators zu warten, so dass die Priorität der Iterator-Aufgabe erhöht wird. Danach wird mit compareAndSwap() versucht, das Markierungsbit ×02 priority_bumped im curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn der Versuch fehlschlägt (da der Iterator bereits vorrückt und curPtr geändert hat), wird priority_bump_uncertainty_resolver (Sperrobjekt für eine einzige Aufgabe) in dem Anzeigeobjekt entsperrt und anschließend als Ergebnis FALSCH zurückgegeben. (Der vorrückende Iterator sieht, dass das Markierungsbit ×01 committed_waiter aktiv, das Markierungsbit ×02 priority_bumped im alten Wert von curPtr jedoch inaktiv gesetzt ist, und weiß so, dass der Mutator nicht über sein Vorrücken unterrichtet werden muss, wartet jedoch darauf, dass sein Objekt priority_bump_uncertainty_resolver entsperrt wird, um sichergehen zu können, dass die Priorität seiner

Aufgabe erhöht wurde, bevor er die Erhöhung aufhebt.) Wenn der Versuch, das Markierungsbit ×02 priority_bumped aktiv zu setzen, erfolgreich ist, wird WAHR zurückgegeben. Dies ist ein Anzeigeobjekt für einen Iterator, auf den der Mutator wartet. (Wenn der Iterator vorrückt und sieht, dass die Markierungsbits ×01 committed_waiter und ×02 priority_bumped im alten Wert von curPtr aktiv gesetzt sind, benachrichtigt er den Mutator über sein Vorrücken und hebt die Prioritätserhöhung seiner eigenen Aufgabe auf.)

4. Wenn curLink den speziellen Übergangswert enthält, gibt dies in allen anderen Fällen an, dass curPtr zwar gesetzt wurde, curLink jedoch noch nicht den Wert erhalten hat, der aus curPtr dereferenziert wird. Der Mutator weiß nicht, ob der in curLink zu speichernde Wert vor oder nach seiner Sperrung des Verknüpfungszeigers erhalten wurde, und weiß so auch nicht, ob er auf das Vorrücken des Iterators warten soll. Um hier sicherzugehen, wird mit compareAndSwapQualified() versucht, curLink so zu ändern, dass es den speziellen Nichtübereinstimmungswert enthält. Wenn dies nicht erfolgreich ist (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, macht er den Versuch des Iterators zunichte, curLink auf den wie auch immer gearteten Wert zu setzen, den er aus curPtr dereferenziert hat, und zwingt ihn dazu, von vorn zu beginnen und curPtr erneut zu dereferenzieren, wobei er dieses Mal sieht, dass der Verknüpfungszeiger gesperrt ist. Auf diese Weise kann der Mutator sicher sein, dass er nicht auf das Vorrücken des Iterators warten muss, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 5 fort.

5. Das Anzeigeobjekt ist allem Anschein nach nicht von Interesse für den Mutator, sollte jedoch daraufhin überprüft werden, ob der curLink-Wert aktuell ist. Mit compareAndSwap() wird versucht, den zuvor erhaltenen Wert wieder in curLink zu speichern. Wenn dies nicht erfolgreich ist, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Andernfalls ist das Anzeigeobjekt für den Mutator nicht relevant, und es wird FALSCH zurückgegeben.

[0245] Nachdem alle Anzeigeobjekte überprüft wurden, steht die Anzahl der WAHR-Ergebnisse für die Anzahl der Iteratoren, die vorrücken müssen, bevor der Mutator fortfahren kann. Diese Anzahl wird dem Objekt status_control innerhalb des Mutator-Sperrobjekts hinzugefügt. Dabei sind die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in curPtr für jeden Iterator, der den Mutator von seinem Vorrücken unterrichten muss (indem das Objekt status_control innerhalb des Sperrobjekts dekrementiert wird), aktiv gesetzt. Der Mutator wartet auf die betreffende Anzahl von Benachrichtigungen.

Suchen von Iteratoren, auf die gewartet werden muss, wenn eine Verknüpfung entfernt wird

[0246] Nachdem die Verknüpfungszeiger, die für die Entfernung einer Verknüpfung benötigt werden, gesperrt wurden, muss der Mutator darauf warten, dass alle Iteratoren, die mit dem Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung bzw. dem Vorwärtszeiger innerhalb der gerade entfernten Verknüpfung verbunden sind, vorrücken. Die Inspektor-Anzeigeobjekte werden überprüft, wenn das Feld active_inspector_view_count im SharedChain-Objekt nicht Null ist, und die Mutator-Anzeigeobjekte werden überprüft, wenn das Feld active_mutator_view_count im SharedChain-Objekt nicht Null ist. Wenn Anzeigeobjekte überprüft werden müssen, gibt die Überprüfung eines jeden Objekts einen booleschen Wert zurück, der angibt, ob der sperrende Mutator darauf warten muss, dass der Iterator vorrückt. Dabei steht die Anzahl der WAHR-Ergebnisse für die Anzahl der „Benachrichtigungen bei Vorrücken", auf die der sperrende Mutator warten muss. Für jedes zu überprüfende Anzeigeobjekt werden die folgenden Schritte durchgeführt:

1. Wenn das curPtr-Feld des Anzeigeobjekts nicht auf den Vorwärts-Verknüpfungszeiger innerhalb der gerade entfernten Verknüpfung zeigt, fährt der Prozess mit Schritt 2 fort. Andernfalls wird mit compareAndSwapQualified() versucht, das Markierungsbit ×02 committed_waiter in dem curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn dies fehlschlägt (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zum Anfang dieses Schritts zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, hat sich der Mutator nun dazu verpflichtet, auf das Vorrücken des Iterators zu warten, woraufhin die Priorität der Iterator-Aufgabe erhöht wird. Danach wird mit compareAndSwap() versucht, das Markierungsbit ×02 priority_bumped im curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn der Versuch fehlschlägt (da der Iterator bereits vorrückt und curPtr geändert hat), wird priority_bum_uncertainty_resolver (Sperrobjekt für eine einzige Aufgabe) in dem Anzeigeobjekt entsperrt und als Ergebnis FALSCH zurückgegeben. (Der vorrückende Iterator sieht, dass das Markierungsbit ×01 committed_waiter aktiv, das Markierungsbit ×02 priority_bumped im alten Wert von curPtr jedoch inaktiv gesetzt ist, und weiß so, dass der Mutator nicht über sein Vorrückten unterrichtet werden muss, wartet jedoch darauf, dass sein Objekt priority_bump_uncertainty_resolver entsperrt wird, um sichergehen zu können, dass die Priorität seiner Aufgabe erhöht wurde, bevor er die Erhöhung aufhebt.) Wenn der Versuch, das Markierungsbit ×02 priority_bumped aktiv zu setzen, erfolgreich ist, wird WAHR zurückgegeben. Dies ist ein Anzeigeobjekt für einen Iterator, auf den der Mutator wartet. (Wenn der Iterator vorrückt und sieht, dass die Markierungsbits ×01 committed_waiter und ×02 priority_bumped im alten Wert von curPtr aktiv gesetzt sind, benachrichtigt

er den Mutator über sein Vorrücken und hebt die Prioritätserhöhung seiner eigenen Aufgabe auf.) Wenn in Schritt 1 nicht WAHR zurückgegeben wurde, fährt der Prozess mit Schritt 2 fort.
2. Wenn das curPtr-Feld des Anzeigeobjekts nicht auf den Vorwärts-Verknüpfungszeiger innerhalb der vorherigen Verknüpfung zeigt, ist das Anzeigeobjekt nicht weiter von Belang, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 3 fort.
3. Wenn das curLink-Feld des Anzeigeobjekts auf das Sperrobjekt dieses Mutators zeigt, hat sich der Iterator bereits dazu verpflichtet, auf den Mutator zu warten. In diesem Fall ist das Anzeigeobjekt nicht weiter von Belang, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 4 fort.
4. Wenn das curLink-Feld des Anzeigeobjekts (1) auf dieselbe Verknüpfung zeigt, auf die der Verknüpfungszeiger gezeigt hat, bevor er von dem Mutator gesperrt wurde, oder wenn es (2) auf ein anderes Sperrobjekt zeigt, ist das Anzeigeobjekt in beiden Fällen für den Mutator relevant. (Wenn ein Anzeigeobjekt für einen Inspektor überprüft wird, zeigt curLink nie auf ein Sperrobjekt, da Inspektoren nicht warten. Wenn dagegen curLink bei einem Mutator-Anzeigeobjekt auf ein anderes Sperrobjekt zeigt, bedeutet dies lediglich, dass der Mutator, der dieses Sperrobjekt verwendet, den Verknüpfungszeiger kürzlich entsperrt hat, jedoch noch nicht dazu kommen ist, curLink zu aktualisieren.) In beiden Fallen wird mit compareAndSwapQualified() versucht, sowohl das Markierungsbit ×01 committed_waiter als auch das Markierungsbit ×04 remove_waiting im curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn dies fehlschlägt (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, hat sich der Mutator nun dazu verpflichtet, auf das (zweimalige) Vorrücken des Iterators zu warten, so dass die Priorität der Iterator-Aufgabe erhöht wird. Danach wird mit compareAndSwap() versucht, das Markierungsbit ×02 priority_bumped im curPtr-Feld des Anzeigeobjekts aktiv zu setzen. Wenn der Versuch fehlschlägt (da der Iterator bereits vorrückt und curPtr geändert hat), wird priority_bump_uncertainty_resolver (Sperrobjekt für eine einzige Aufgabe) in dem Anzeigeobjekt entsperrt und als Ergebnis WAHR zurückgegeben. (Der vorrückende Iterator sieht, dass die Markierungsbits ×01 committed_waiter und ×04 remove_waiting aktiv gesetzt sind, das Markierungsbit ×02 priority_bumped im alten Wert von curPtr jedoch inaktiv gesetzt ist. Er setzt die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in curPtr aktiv, so dass er weiß, dass er den Mutator benachrichtigen und die Priorität seiner eigenen Aufgabe herabsetzen muss, wenn er das nächste Mal vorrückt. In der Zwischenzeit wartet er darauf, dass sein Objekt priority_bump_uncertainty_resolved entsperrt wird, um sichergehen zu können, dass die Priorität seiner Aufgabe erhöht wurde, bevor er die Erhöhung schließlich aufhebt.) Wenn der Versuch, das Markierungsbit ×02 priority_bumped aktiv zu setzen, erfolgreich ist, wird WAHR zurückgegeben; andernfalls fährt der Prozess mit Schritt 5 fort. Im ersteren Fall handelt es sich dabei um ein Anzeigeobjekt für einen Iterator, auf den der Mutator wartet. (Wenn der Iterator vorrückt und sieht, dass alle drei Markierungsbits im alten Wert von curPtr aktiv gesetzt sind, setzt er die Markierungsbits ×01 committed_waiter und ×02 priority_bumped in curPtr aktiv, so dass er weiß, dass er den Mutator benachrichtigen und die Priorität seiner eigenen Aufgabe erhöhen muss, wenn er das nächste Mal vorrückt.)
5. Wenn curLink den speziellen Übergangswert enthält, gibt dies an, dass zwar curPtr gesetzt wurde, curLink jedoch noch nicht auf den Wert gesetzt wurde, der bei der Dereferenzierung von curPtr erhalten wird. Der Mutator weiß nicht, ob der in curLink zu speichernde Wert vor oder nach seiner Sperrung des Verknüpfungszeigers erhalten wurde, und er weiß somit auch nicht, ob er auf das Vorrücken des Iterators warten muss. Um sicherzugehen, wird mit compareAndSwapQualified() versucht, curLink so zu ändern, dass es den speziellen Nichtübereinstimmungswert enthält. Wenn dies nicht erfolgreich ist (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, macht dies den Versuch des Iterators zunichte, curLink auf den wie auch immer gearteten Wert zu setzen, den er curPtr dereferenziert hat, und zwingt ihn dazu, von vorn zu beginnen und curPtr erneut zu dereferenzieren, wobei er dieses Mal sieht, dass der Verknüpfungszeiger gesperrt ist. Auf diese Weise kann der Mutator sicher sein, dass er nicht auf das Vorrücken des Iterators warten muss, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 6 fort.
6. Wenn curLink nicht den speziellen Übergangswert enthält, ist das Anzeigeobjekt allem Anschein nach nicht von Interesse für den Mutator, sollte jedoch daraufhin überprüft werden, ob der curLink-Wert aktuell ist. Mit compareAndSwap() wird versucht, den zuvor erhaltenen Wert wieder in curLink zu speichern. Wenn dies nicht erfolgreich ist, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Andernfalls ist das Anzeigeobjekt für den Mutator nicht relevant, und es wird FALSCH zurückgegeben.

[0247] Nachdem alle Anzeigeobjekte überprüft wurden, steht die Anzahl der WAHR-Ergebnisse für die Anzahl der Iteratoren, die vorrücken müssen, bevor der Mutator fortfahren kann. Diese Anzahl wird dem Objekt status_control innerhalb des Mutator-Sperrobjekts hinzugefügt. Danach wird gewartet, bis die betreffende Anzahl von Benachrichtigungen (d.h. Dekrementierungen des Objekts status_control) eingegangen ist.

Suchen von durchlaufenden Mutatoren, die geweckt werden müssen, wenn Verknüpfungszeiger entsperrt werden

[0248] Wenn ein Mutator seine Verknüpfungszeiger entsperrt und falls im alten (gesperrten) Wert des Vorwärts-Verknüpfungszeigers das Markierungsbit ×02 iterating_mutator_waiting aktiv gesetzt ist, bedeutet dies, dass sich mindestens ein durchlaufender Mutator eventuell gerade dazu verpflichtet oder sich bereits dazu verpflichtet hat, darauf zu warten, dass der Mutator seine Verknüpfungszeiger entsperrt. In diesem Fall wird jedes einzelne Anzeigeobjekt für Mutatoren überprüft. Die Überprüfung eines jeden Objekts ergibt einen booleschen Wert, der angibt, ob sich der durchlaufende Mutator dazu verpflichtet hat, auf die Entsperrung der Verknüpfungszeiger zu warten. Dabei steht die Anzahl der WAHR-Ergebnisse für die Anzahl der durchlaufenden Mutatoren, die geweckt werden und die nach dem Wecken den sperrenden Mutator darüber unterrichten, dass sie nun nicht mehr warten. (Wenn dies von allen angegeben wurde, kann das Sperrobjekt wiederverwendet werden.) Für jedes der überprüften Anzeigeobjekte werden die folgenden Schritte durchgeführt:

1. Wenn das curPtr-Feld des Anzeigeobjekts nicht auf den Vorwärtszeiger innerhalb der vorherigen Verknüpfung zeigt, ist das Anzeigeobjekt nicht weiter von Belang für den entsperrenden Mutator, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 2 fort.
2. Wenn das curLink-Feld des Anzeigeobjekts auf das Sperrobjekt des entsperrenden Mutators zeigt (wobei das Markierungsbit ×01 committed_waiter aktiv gesetzt ist), hat sich der durchlaufende Iterator dazu verpflichtet, auf die Entsperrung der Verknüpfungszeiger zu warten. In diesem Fall wird mit compareAndSwap() versucht, curLink auf den neuen (entsperrten) Wert des Verknüpfungszeigers zu setzen. Wenn der Versuch fehlschlägt (da der durchlaufende Mutator das Markierungsbit ×02 priority_bumped oder ×04 priority_not_bumped in curLink aktiv setzt), kehrt der Prozess an den Anfang dieses Schritts zurück und beginnt von vorn. Sobald der Versuch erfolgreich ist, verfügt der durchlaufende Mutator über eine aktuelle Verknüpfung, und der entsperrende Mutator weiß aufgrund des alten Werts von curLink, ob der durchlaufende Mutator die Priorität seiner Aufgabe erhöht hat bzw. ob diese Information ungewiss ist. Wenn im alten Wert von curLink weder das Markierungsbit ×02 priority_bumped noch das Markierungsbit ×04 priority_not_bumped aktiv gesetzt ist, weiß der entsperrende Mutator nicht, ob der durchlaufende Mutator die Priorität seiner Aufgabe erhöht hat. In diesem Fall wird das Objekt status_control innerhalb des Sperrobjekts des entsperrenden Mutators inkrementiert, und es wird FALSCH zurückgegeben. (Der Versuch des durchlaufenden Mutators, entweder das Markierungsbit ×02 priority_bumped oder das Markierungsbit ×04 priority_not_bumped aktiv zu setzen schlägt fehl (da der entsperrende Mutator curLink geändert hat), und wenn er später den Verknüpfungszeiger erneut dereferenziert, sieht er, dass dieser entsperrt ist.) Wenn der durchlaufende Mutator in der Zwischenzeit die Priorität der Aufgabe des sperrenden Mutators erhöht hatte, inkrementiert er das Feld priority_bump_count innerhalb des Sperrobjekts des entsperrenden Mutators. Danach dekrementiert er in beiden Fällen das Objekt status_control des entsperrenden Mutators, um so anzugeben, dass die Unsicherheit bezüglich der Prioritätserhöhung ausgeräumt wurde. Andernfalls ist entweder das Markierungsbit ×02 priority_bumped oder das Markierungsbit ×04 priority_not_bumped aktiv gesetzt. Wenn das Markierungsbit ×02 priority_bumped aktiv gesetzt ist, wird das Feld priority_bump_count innerhalb des Sperrobjekts des entsperrenden Mutators inkrementiert. In beiden Fällen wird WAHR zurückgegeben; andernfalls fährt der Prozess mit Schritt 3 fort.
3. Wenn curLink den speziellen Übergangswert enthält, gibt dies an, dass der durchlaufende Mutator curPtr zwar gesetzt hat, dass er jedoch noch nicht curLink auf den Wert gesetzt hat, den er durch die Dereferenzierung von curPtr erhält. Der entsperrende Mutator weiß nicht, ob er den durchlaufenden Mutator wecken muss. Um sicherzugehen, wird mit compareAndSwapQualified() versucht, curLink, so zu ändern, dass es den speziellen Nichtübereinstimmungswert enthält. Wenn dies fehlschlägt (da entweder curLink oder curPtr zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, schlägt der Versuch des durchlaufenden Mutators, curLink zu setzen, zwangsläufig fehl, und wenn er curPtr erneut dereferenziert, sieht er, dass der Verknüpfungszeiger entsperrt ist. Auf diese Weise kann der entsperrende Mutator sicher sein, dass der durchlaufende Mutator nicht auf ihn wartet, so dass FALSCH zurückgegeben wird.
4. Wenn curLink nicht den speziellen Übergangswert enthält, ist das Anzeigeobjekt allem Anschein nach nicht von Belang für den Mutator, wobei diese Prozedur jedoch überprüft, ob der curLink-Wert aktuell ist. Mit compareAndSwap() wird versucht, den zuvor erhaltenen Wert wieder in curLink zu speichern. Wenn dies nicht erfolgreich ist, kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Andernfalls ist das Anzeigeobjekt für den Mutator nicht von Interesse, und es wird FALSCH zurückgegeben.

[0249] Nachdem alle Anzeigeobjekte für Mutatoren überprüft wurden, verfügt der entsperrende Mutator über die folgenden Zählwerte: (a) die Anzahl der wartenden Mutatoren, die er wecken und die er darüber unterrichten muss, dass sie geweckt wurden und nicht mehr warten; (b) die Anzahl der verbleibenden Unsicherheiten bezüglich der Prioritätserhöhung (in status_control), auf deren Benachrichtigung über ihre Behebung der sper-

rende Mutator warten muss; und (c) die Anzahl der Prioritätserhöhungen, die der sperrende Mutator von seinen wartenden Mutatoren empfangen hat (in priority_bump_count).

Suchen von sperrenden Mutatoren, die geweckt werden müssen, wenn Verknüpfungszeiger entsperrt werden

[0250] Wenn ein Mutator seine Verknüpfungszeiger entsperrt und falls im alten (gesperrten) Wert eines Verknüpfungszeigers das Markierungsbit ×04 blocking_mutator aktiv gesetzt ist, bedeutet dies, dass sich mindestens ein sperrender Mutator eventuell gerade dazu verpflichtet oder sich bereits dazu verpflichtet hat, darauf zu warten, dass der entsperrende Mutator seine Verknüpfungszeiger entsperrt. In diesem Fall muss jedes einzelne Sperrobjekt überprüft werden. Die Überprüfung eines jeden Objekts gibt einen booleschen Wert zurück, der angibt, ob sich der sperrende Mutator dazu verpflichtet hat, auf die Entsperrung der Verknüpfungszeiger zu warten. Dabei steht die Anzahl der WAHR-Ergebnisse für die Anzahl der sperrenden Mutatoren, die geweckt werden müssen und die nach dem Wecken den entsperrenden Mutator darüber unterrichten, dass sie nun nicht mehr warten. (Wenn dies von allen angegeben wurde, kann das Sperrobjekt wiederverwendet werden.) Bei der Überprüfung eines jeden Sperrobjekts werden die folgenden Schritte durchgeführt:

1. Wenn das Feld waiting_for_blocker innerhalb des Sperrobjekts nicht auf das Sperrobjekt des entsperrenden Mutators zeigt (wobei alle Markierungsbits ignoriert werden), ist dieses Sperrobjekt nicht von Interesse, so dass FALSCH zurückgegeben wird; andernfalls fährt der Prozess mit Schritt 2 fort.
2. Mit compareAndSwap() wird versucht, das Feld waiting_for_blocker des Sperrobjekts auf NULL zu setzen. Wenn der Versuch fehlschlägt (da der sperrende Mutator die darin enthaltenen Markierungsbits setzt), kehrt der Prozess zum Anfang dieses Schritts zurück und unternimmt einen erneuten Versuch. Wenn der Versuch schließlich erfolgreich ist, ist das Feld waiting_for_blocker NULL und gibt so an, dass der sperrende Mutator nicht mehr wartet.
3. Wenn das Markierungsbit ×01 committed_waiter im alten Wert des Felds waiting_for_blocker inaktiv gesetzt ist, gibt dies an, dass der sperrende Mutator nicht dazu gekommen war, sich zum Warten zu verpflichten, als der entsperrende Mutator die Verknüpfungszeiger entsperrt hat. In diesem Fall wird FALSCH zurückgegeben; andernfalls fährt der Prozess mit Schritt 4 fort.
4. Der sperrende Mutator hat sich dazu verpflichtet, darauf zu warten, dass der entsperrende Mutator seine Verknüpfungszeiger entsperrt. Der sperrende Mutator hat versucht, die Priorität der Aufgabe des entsperrenden Mutators zu erhöhen, und kann, muss jedoch nicht, dazu gekommen sein, dies zu tun und die Ergebnisse anzugeben. Wenn im alten Wert von curLink weder das Markierungsbit ×02 priority_bumped noch das Markierungsbit ×04 priority_not_bumped aktiv gesetzt ist, besteht eine Unsicherheit bezüglich der Prioritätserhöhung. In diesem wird das Objekt status_control innerhalb des Sperrobjekts des entsperrenden Mutators inkrementiert, und es wird das Ergebnis FALSCH zurückgegeben; andernfalls fährt der Prozess mit Schritt 5 fort. (Der Versuch des sperrenden Mutators, das Markierungsbit ×02 priority_bumped oder ×04 priority_not_bumped zu setzen, ist fehlgeschlagen (da der entsperrende Mutator das Feld waiting_for_blocker auf NULL gesetzt hat), und wenn er später den Verknüpfungszeiger erneut dereferenziert, sieht er, dass dieser entsperrt ist.) Wenn der sperrende Mutator zwischenzeitlich die Priorität der Aufgabe des entsperrenden Mutators erhöht hat, inkrementiert er das Feld priority_bump_count innerhalb des Sperrobjekts des entsperrenden Mutators. Danach dekrementiert er in beiden Fällen das Objekt status_control des entsperrenden Mutators, um so anzugeben, dass die Unsicherheit bezüglich der Prioritätserhöhung behoben wurde.
5. Eines der Markierungsbits ×02 priority_bumped oder ×04 priority_not_bumped ist auf aktiv gesetzt. Wenn das Markierungsbit ×02 priority_bumped aktiv gesetzt ist, wird das Feld priority_bump_count innerhalb des Sperrobjekts des entsperrenden Mutators inkrementiert. In beiden Fällen wird WAHR zurückgegeben.

[0251] Nachdem alle Sperrobjekte überprüft wurden, verfügt der entsperrende Mutator über die folgenden Zählwerte: (a) die Anzahl der wartenden Mutatoren, die er wecken und die er darüber unterrichten muss, dass sie geweckt wurden und nicht mehr warten; (b) die Anzahl der Unsicherheiten bezüglich der Prioritätserhöhung (im Objekt status_control), auf deren Benachrichtigung über ihre Behebung der sperrende Mutator warten muss; und (c) die Anzahl der Prioritätserhöhungen, die der sperrende Mutator von seinen wartenden Mutatoren empfangen hat (Feld priority_bump_count). Diese Informationen legen fest, wann das Sperrobjekt für die Wiederverwendung bei einer anderen Änderungsfunktion zur Verfügung steht.

Warten und Wecken

Warten, dass ein Mutator seine Verknüpfungszeiger entsperrt

[0252] Diese Prozedur wird entweder von einem durchlaufenden oder einem sperrenden Mutator aufgerufen, der darauf warten muss, dass ein zweiter Mutator seine Verknüpfungszeiger entsperrt. Dabei wird entweder

ein Verweis auf das curLink-Feld innerhalb eines durchlaufenden Mutators oder auf das Feld waiting_for_blocker im Sperrobjekt eines Mutators als Parameter weitergegeben. Dies wird durchgeführt, damit das Feld mit den Ergebnissen des Versuchs, die Priorität der Aufgabe des zweiten Mutators zu erhöhen, aktualisiert werden kann. (Das Markierungsbit ×01 committed_waiter wurde hier bereits aktiv gesetzt.) Diese Funktion gibt nur ein Ergebnis zurück, nachdem auf die Entsperrung der Verknüpfungszeiger gewartet wurde oder nachdem ermittelt wurde, dass ein Warten nicht notwendig ist, da beim Setzen der Werte für die Prioritätserhöhung festgestellt wurde, dass die Verknüpfungszeiger bereits entsperrt sind.

1. Die Priorität der Aufgabe des zweiten Mutators wird erhöht, und es wird das Markierungsbit ×02 priority_bumped in Schritt 2 als Markierungsbit verwendet. Wenn es allerdings nicht notwendig sein sollte, die Priorität der Aufgabe des zweiten Mutators zu erhöhen, wird das Markierungsbit ×04 priority_not_bumped in Schritt 2 als Markierungsbit verwendet.
2. Mit compareAndSwap() wird versucht, das in Schritt 1 ermittelte Markierungsbit aktiv zu setzen. Wenn dies erfolgreich ist, fährt der Prozess mit Schritt 3 fort. Andernfalls ist der Versuch fehlgeschlagen, da der zweite Mutator die Verknüpfungszeiger bereits entsperrt und das relevante Feld aktualisiert hat: Wenn es sich bei dem Feld um curLink innerhalb eines Anzeigeobjekts handelt, zeigt es nun auf die neue aktuelle Verknüpfung für den Iterator; wenn es sich um das Feld waiting_for_blocker innerhalb eines Sperrobjekts handelt, ist es nun NULL. In beiden Fällen ist es nicht sinnvoll zu warten, da der Verknüpfungszeiger nun entsperrt ist, wobei der zweite Mutator, der weder das Markierungsbit ×02 priority_bumped noch das Markierungsbit ×04 priority_not_bumped im alten Wert des Felds sieht, jedoch nicht weiß, ob die Priorität seiner Aufgabe erhöht wurde. Wenn die Priorität also in Schritt 1 erhöht wurde, wird der Zählwert für die Prioritätserhöhung im Sperrobjekt des zweiten Mutators inkrementiert. Danach wird in beiden Fällen das Objekt status_control im Sperrobjekt des zweiten Mutators dekrementiert, um ihn so darüber zu unterrichten, dass die Unsicherheit bezüglich der Prioritätserhöhung behoben wurde, und der Prozess endet. (Der zweite Mutator wartet, bis schließlich alle Unsicherheiten bezüglich der Prioritätserhöhung behoben sind.)
3. An dieser Stelle wurde das Markierungsbit gesetzt um anzugeben, ob die Priorität der Aufgabe des zweiten Mutators erhöht wurde, so dass darauf gewartet wird, dass der zweite Mutator seine Verknüpfungszeiger entsperrt. (Wenn der Mutator seine Verknüpfungszeiger entsperrt und die Sperr- oder Anzeigeobjekte des wartenden Mutators überprüft, weiß er, ob die Priorität seiner Aufgabe erhöht wurde.) Nachdem er aus dem Wartezustand geweckt wurde, wird das Objekt status_control im Sperrobjekt des zweiten Mutators dekrementiert, um so anzugeben, dass das Warten beendet ist, und der Prozess endet. (Hiermit wird angegeben, dass der aktuelle Mutator über keinen Verweis auf das Sperrobjekt des zweiten Mutators mehr verfügt, und nachdem alle derartigen Verweise abgelaufen sind, kann das Sperrobjekt bei einer anderen Änderungsfunktion verwendet werden.)

Primäre Warte- und Weckmechanismen

[0253] Das Zustandszähler-Objekt status_control ist das primäre Objekt, wenn es darum geht, dass Iteratoren und Mutatoren aufeinander warten und einander wecken. Dieses Objekt befindet sich im Sperrobjekt und steuert die folgenden Ereignisse: (a) Ermitteln, ob das Sperrobjekt aktiv oder inaktiv ist, sowie Ermitteln der atomaren Mittel für den Übergang zwischen diesen Zuständen; (b) Veranlassen, dass der Mutator, der das Sperrobjekt verwendet, auf das Vorrücken von Iteratoren wartet; (c) Veranlassen, dass durchlaufende und sperrende Mutatoren darauf warten, dass der das Sperrobjekt verwendende Mutator seine Verknüpfungszeiger entsperrt; und (d) Veranlassen, dass der das Sperrobjekt verwendende Mutator darauf wartet, dass alle Unsicherheiten bezüglich der Prioritätserhöhung behoben sind. Tabelle 15 beschreibt die verschiedenen Werte des Zustandszählers sowie ihre Bedeutung:

Tabelle 15: Werte des Zustandszählers

Wert	Bedeutung
0	Dieser Wert gibt an, dass sich das Sperrobjekt im inaktiven Zustand befindet. Um das Sperrobjekt in den aktiven Zustand zu versetzen (wenn es einem Mutator zugewiesen wird), wird der Zählwert atomar von Null auf einen Wert ungleich Null gesetzt. (Zuvor wird der Sperrobjekt-Teil des Felds activated_blocker_and_ view_counts im SharedChain-Objekt atomar um den Wert 1 erhöht.)
×8000 ...	Das höchstwertige Bit ist 1, alle anderen Bits sind Null. Dieser Wert wird dem Zählwert zugewiesen, um seinen Zustand von inaktiv in aktiv zu ändern. Während der Zählwert diesen Wert aufweist, befindet sich der das Sperrobjekt verwendende Mutator in einem Gleichgewichtszustand und wartet nicht auf wie auch immer geartete Benachrichtigungen von durchlaufenden oder sperrenden Mutatoren. Wenn der sperrende Mutator etwaige Verknüpfungszeiger gesperrt hat, warten alle wartenden Mutatoren darauf, dass der Zählwert zu einem Wert unter ×4000... wechselt, so dass dieser Wert sie veranlasst, zu warten.
×8000 ... +/– x	Ein Wert in diesem Bereich gibt an, dass der sperrende Mutator auf das Vorrücken von Iteratoren wartet. Der Wert x ist die Anzahl der beteiligten Iteratoren, wobei diese Anzahl innerhalb eines bestimmten Wertebereichs liegt. Wenn ein sperrender Mutator ein Anzeigeobjekt überprüft, setzt er die betreffenden Markierungsbits in curPtr und zählt die Anzahl der Iteratoren, auf die er warten muss. Wenn die Iteratoren vorrücken, benachrichtigen sie den sperrenden Mutator über ihr Vorrücken, indem sie den Zustandzählwert jeweils um den Wert 1 erniedrigen. Nachdem er alle Anzeigeobjekte überprüft hat, addiert der sperrende Mutator die Anzahl, auf die er warten muss, atomar zu seinem Zustandszähler. Danach wartet er, bis der Zustandszähler bis zum Wert für den Gleichgewichtszustand ×8000 ... dekrementiert wurde, während etwaige verbleibende Iteratoren ihn über ihr Vorrücken unterrichten. Nachdem alle Benachrichtigungen eingegangen sind, kann der sperrende Mutator seine Verknüpfungszeiger entsperren und weiß dabei, dass keine anderen Iteratoren mehr unterwegs sind. Während dieses gesamten Zeitraums ist der Zählwert ausreichend groß, um alle Mutatoren, die auf die Entsperrung der Verknüpfungszeiger warten müssen, zum Warten zu veranlassen.
×8000 ... +/– y	Ein Wert in diesem Bereich gibt an, dass der sperrende Mutator darauf wartet, dass Unsicherheiten bezüglich der Prioritätserhöhung behoben werden. Dabei ist der Wert y die Anzahl der verbleibenden Unsicherheiten und weist denselben Wertebereich auf wie x oben. Anhand des Zählwerts lässt sich dieser Zustand vom obigen Zustand unterscheiden, bei dem der sperrende Mutator auf das Vorrücken von Iteratoren wartet. In diesem Fall hat der sperrende Mutator jedoch seine Verknüpfungszeiger entsperrt, und während er die Anzahl der verpflichteten wartenden Mutatoren gezählt hat, hat er auch die Anzahl der Unsicherheiten bezüglich der Prioritätserhöhung gezählt. Indem wartende Mutatoren die Unsicherheit, auf die sie stoßen, beheben, dekrementieren sie diesen Zählwert. Zwischenzeitlich addiert der sperrende Mutator die Gesamtzahl der Unsicherheiten zu diesem Zähler und wartet darauf, dass dieser den Wert für den Gleichgewichtszustand von ×8000 ... erreicht. Wenn dieser Wert erreicht wird, kann der sperrende Mutator sicher sein, dass er weiß, wie oft seine Priorität von wartenden Objekten erhöht wurde, und er kann die Erhöhung seiner Priorität um die entsprechende Zahl rückgängig machen.
z	Nachdem ein sperrender Mutator seine Verknüpfungszeiger entsperrt und darauf gewartet hat, dass alle Unsicherheiten bezüglich der Prioritätserhöhung behoben wurden, setzt er den Zählwert auf die Anzahl der verpflichteten wartenden Mutatoren, die er zuvor gezählt hat. Dadurch werden alle wartenden Objekte geweckt (wobei diejenigen, die sich zum Warten verpflichtet, jedoch noch nicht damit begonnen haben, den Wartezustand sofort verlassen, wenn sie diesen Punkt erreichen). Wenn jeder einzelne Mutator geweckt wird, benachrichtigt er den sperrenden Mutator, dass er nicht mehr wartet, indem er seinen Zähler um den Wert 1 erniedrigt. Somit steht der Wert z für die Anzahl der noch ausstehenden wartenden Objekte, die den Mutator noch nicht darüber unterrichtet haben, dass sie nicht mehr warten. Wenn der Zähler den Wert Null erreicht, kehrt das Sperrobjekt in den inaktiven Zustand zurück und kann von einem anderen Mutator verwendet werden (da keine weiteren ausstehenden Verweise darauf vorhanden sind), und der Sperrobjekt-Teil des Felds activated_blocker_view_counts im SharedChain-Objekt wird um den Wert 1 erniedrigt. Wenn das gesamte Feld Null ist, können mit der Reklamationsfunktion nicht mehr benötigte Sperr- oder Anzeigeobjekte gelöscht werden.

[0254] Der Mechanismus für die Verknüpfungsübertragung ist das Mittel, mit dem ein sperrender Mutator die Verwendung eines Verknüpfungszeigers auf einen sperrenden Mutatoren übertragen kann. Dieser Mechanismus kommt in zwei verschiedenen Situationen zum Tragen, die unterschiedliche Zustandszähler-Objekte innerhalb des Sperrobjekts verwenden:

1. Wenn Mutator 1 (unter Verwendung von Sperrobjekt 1) den Rückwärtszeiger innerhalb einer Verknüpfung vorläufig gesperrt hat und danach feststellt, dass der Vorwärtszeiger innerhalb der vorherigen Verknüpfung (unter Verwendung von Sperrobjekt 2) von Mutator 2 gesperrt wird, setzt Mutator 1 seine vorläufige Sperre aus und ermöglicht Mutator 2 so die Verwendung des Verknüpfungszeigers. Mutator 1 setzt mit dem Mechanismus für die Verknüpfungsübertragung einen Zeiger auf die vorherige Verknüpfung in das Zustandszähler-Objekt suspended_link_control innerhalb von Sperrobjekt 2. Zwischenzeitlich hat Mutator 2 auf dieses Objekt suspended_link_control gewartet und wird geweckt, wenn Mutator 1 den Verknüpfungszeiger dort setzt. Nachdem der Verknüpfungszeiger auf Mutator 2 gesetzt wurde, wartet Mutator 1 auf das Objekt suspended_link_control innerhalb seines eigenen Sperrobjekts. Wenn Mutator 2 fertig ist, überträgt er den Verknüpfungszeiger zurück zu Mutator 1 und setzt einen Zeiger auf die neue vorherige Verknüpfung in das Objekt suspended_link_control innerhalb von Sperrobjekt 1, so dass Mutator 1 geweckt wird.
2. Wenn Mutator 1 (unter Verwendung von Sperrobjekt 1) eine Verknüpfung entfernen möchte, aber feststellt, dass der Vorwärtszeiger innerhalb der Verknüpfung (unter Verwendung von Sperrobjekt 2) bereits von Mutator 2 gesperrt wird, wartet Mutator 1 auf das Zustandszähler-Objekt blocked_remove_control innerhalb seines Sperrobjekts. Wenn Mutator 2 den Verknüpfungszeiger entsperren möchte und dabei feststellt, dass ein Mutator, der die darin enthaltene Verknüpfung entfernen möchte, darauf wartet, entsperrt er den Verknüpfungszeiger nicht, sondern sperrt ihn für Mutator 1 erneut. Danach gibt Mutator 2 mit dem Mechanismus für die Verknüpfungsübertragung einen Zeiger auf die nächste Verknüpfung an Mutator 1 weiter, indem er den Zeiger im Zustandszähler-Objekt blocked_remove_control innerhalb von Sperrobjekt 1 speichert. Dies weckt Mutator 1, der den Zeiger auf die nächste Verknüpfung aus seinem Objekt blocked_remove_control erhält und fortfährt, da er nun den Vorwärts-Verknüpfungszeiger innerhalb der Verknüpfung, die er entfernen wird, erfolgreich gesperrt hat.

[0255] Der Mechanismus für die Verknüpfungsübertragung funktioniert unabhängig von der Situation, in der er verwendet wird, immer gleich. Ein Zustandszähler-Objekt enthält entweder einen Zeiger auf eine Verknüpfung oder einen Wert, der den Status des Mutators, zu dem der Zustandszähler gehört, widerspiegelt. Tabelle 16 zeigt die Statuswerte, die in dem Zähler auftreten können. Sie werden so ausgewählt, dass sie einerseits konsistent mit den Markierungsbits sind, die in Verknüpfungszeigern auftreten, und andererseits nicht mit einem Verknüpfungszeiger verwechselt werden können (wobei sich alle Null-Werte als Verknüpfungszeiger entweder auf die erste oder die letzte Verknüpfung in der Kette beziehen).

Tabelle 16: Statuswerte

Wert	Bedeutung	Beschreibung
×00 ... 08	Inaktiv (inactive)	Dies ist der Ausgangswert des Zählers. Er gibt an, dass der Mutator noch nicht damit begonnen hat, darauf zu warten, dass er einen Verknüpfungszeiger empfängt.
×00 ... 01	Objekt mit Warteverpflichtung (committed_waiter)	Dies ist der Wert, den ein Mutator speichert, wenn er mit dem Warten beginnt. Er gibt an, dass Unsicherheit darüber besteht, ob die Priorität der Aufgabe für den Mutator, auf den gewartet wird, erhöht wurde.
×00 ... 03	Priorität erhöht (priority_bumped)	Dieser Wert gibt an, dass der Mutator auf den Empfang eines Verknüpfungszeigers wartet und die Priorität der Aufgabe des Mutators, von dem er den Verknüpfungszeiger empfangen wird, erhöht hat.
×00 ... 05	Priorität nicht erhöht (priority_not_bumped)	Dieser Wert gibt an, dass der Mutator auf den Empfang eines Verknüpfungszeigers wartet und festgestellt hat, dass es nicht notwendig ist, die Priorität der Aufgabe des Mutators, von dem er den Verknüpfungszeiger empfangen wird, zu erhöhen.

[0256] Es ist vorab nicht bekannt, ob der Mutator, der den Verknüpfungszeiger überträgt, dies tut, bevor, währenddessen oder nachdem sich der Mutator, der den Verknüpfungszeiger empfangen soll, darauf vorbereitet bzw. vorbereitet hat.

Übertragen eines Verknüpfungszeigers an einen anderen Mutator

[0257] Wenn ein Mutator einen Verknüpfungszeiger an einen zweiten Mutator übertragen möchte, kennt er das Sperrobjekt dieses zweiten Mutators und verwendet den betreffenden Zustandszähler innerhalb des Sperrobjekts. Dabei wird der Verknüpfungszeiger mit den folgenden Schritten übertragen:

1. Aus dem Zustandszähler-Objekt wird der aktuelle Zählerwert erhalten.
2. Mit compareAndSwap() wird versucht, den Zähler von seinem aktuellen Wert zum Wert des Verknüpfungszeigers zu ändern. Wenn der Versuch fehlschlägt (da der empfangende Mutator Markierungsbits in dem Zähler setzt), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn; andernfalls fährt er mit Schritt 3 fort.
3. Wenn der alte Zählerwert ×08 inaktiv lautete, gibt dies an, dass der zweite Mutator noch nicht mit dem Warten begonnen hat und genau genommen weder warten noch die Priorität der Aufgabe des ersten Mutators erhöhen wird. In diesem Fall ist die Übertragungsfunktion abgeschlossen, andernfalls fährt der Prozess mit Schritt 4 fort.
4. Wenn der alte Zählerwert ×01 committed_waiter lautete, hat der zweite Mutator mit dem Warten begonnen und versucht zunächst, bevor er mit dem eigentlichen Warten beginnt, die Priorität der Aufgabe des ersten Mutators zu erhöhen, wobei der erste Mutator jedoch nicht wissen kann, ob die Priorität seiner Aufgabe erhöht wurde. Das Objekt status_control in dem Sperrobjekt des ersten Mutators wird inkrementiert, um eine Unsicherheit bezüglich der Prioritätserhöhung anzuzeigen, die noch behoben werden muss, und die Übertragungsfunktion ist abgeschlossen. (Der erste Mutator wartet darauf, dass alle Unsicherheiten bezüglich einer Prioritätserhöhung behoben sind, und der zweite Mutator dekrementiert das Objekt status_control des ersten Mutators, um dies anzuzeigen. Dabei wartet der zweite Mutator nicht auf die Verknüpfungsübertragung.) Wenn der alte Zählerwert nicht ×01 committed_waiter lautete, fährt der Prozess dagegen mit Schritt 5 fort.
5. Wenn der alte Zählerwert ×03 priority_bumped lautete, hat der zweite Mutator die Priorität der Aufgabe des ersten Mutators erhöht. In diesem Fall wird der Wert von priority_bump_count im Sperrobjekt des ersten Mutators inkrementiert, und die Übertragungsfunktion ist abgeschlossen. (Der erste Mutator hebt schließlich die Prioritätserhöhung seiner Aufgabe auf. Zwischenzeitlich wurde der zweite Mutator aus seinem Wartezustand geweckt.)
6. Andernfalls lautete der alte Zählerwert ×05 priority_not_bumped und gibt somit an, dass der zweite Mu-

tator festgestellt hat, dass die Priorität der Aufgabe des ersten Mutators nicht erhöht werden muss. In diesem Fall ist die Übertragungsfunktion abgeschlossen. (Der zweite Mutator wird aus seinem Wartezustand geweckt.)

Warten auf den Empfang eines übertragenen Verknüpfungszeigers

[0258] Wenn ein Mutator einen Verknüpfungszeiger von einem zweiten Mutator empfangen möchte, weiß er, dass der zweite Mutator den Verknüpfungszeiger an ein bestimmtes Zustandszähler-Objekt innerhalb des Sperrobjekts des ersten Mutators weiterleitet. Die folgenden Schritte beschreiben den Prozess, mit dem auf den Empfang des Verknüpfungszeigers gewartet wird.

1. Mit compareAndSwap() wird versucht, den Zähler des Zustandszähler-Objekts von seinem erwarteten Wert ×08 inactive zu ×01 committed_waiter zu ändern. Wenn der Versuch fehlschlägt, liegt dies daran, dass der zweite Mutator den Verknüpfungszeiger bereits übertragen hat. In diesem Fall fährt der Prozess mit Schritt 6 fort; andernfalls fährt er mit Schritt 2 fort.
2. Der Versuch war erfolgreich, und der neue Wert des Zählers gibt an, dass, wenn der zweite Mutator den Verknüpfungszeiger nun übertragen würde, er nicht wissen würde, ob die Priorität seiner Aufgabe erhöht wurde. tryBump() wird aufgerufen, um zu versuchen, die Priorität der Aufgabe des zweiten Mutators zu erhöhen und den Wert ×03 priority_bumped in den folgenden Schritten zu verwenden; wenn es allerdings nicht notwendig ist, die Priorität zu erhöhen, wird stattdessen der Wert ×05 priority_not_bumped verwendet.
3. Mit compareAndSwap() wird versucht, den Zählwert von dem erwarteten Wert ×01 committed_waiter zu dem in Schritt 2 ermittelten Wert zu ändern. Wenn der Versuch fehlschlägt, liegt dies daran, dass der zweite Mutator den Verknüpfungszeiger soeben übertragen hat. In diesem Fall fährt der Prozess mit Schritt 5 fort; andernfalls fährt er mit Schritt 4 fort.
4. Der Zähler gibt nun an, ob die Priorität des zweiten Mutators erhöht wurde, so dass darauf gewartet wird, dass der zweite Mutator den Verknüpfungszeiger erhöht. Dies bedeutet, dass darauf gewartet wird, dass der Zustandszähler einen anderen Wert als den in Schritt 3 gesetzten erhält. Wenn der zweite Mutator den Verknüpfungszeiger überträgt, weckt er den ersten Mutator. Nach dem Wecken fährt der Prozess mit Schritt 6 fort.
5. An dieser Stelle hat der zweite Mutator den Verknüpfungszeiger übertragen, weiß jedoch nicht, ob seine Priorität erhöht wurde. Wenn die Priorität in Schritt 2 erhöht wurde, wird das Feld priority_bump_count innerhalb des Sperrobjekts des zweiten Mutators inkrementiert. In beiden Fällen wird das Objekt status_control innerhalb des Sperrobjekts des zweiten Mutators dekrementiert, um diesen darüber zu unterrichten, dass die Unsicherheit bezüglich der Prioritätserhöhung behoben wurde.
6. An dieser Stelle hat der zweite Mutator den Verknüpfungszeiger übertragen. Der Verknüpfungszeiger wird aus dem Zustandszähler abgerufen.
7. Der Zustandszähler wird wieder auf den Wert ×08 inactive gesetzt. Dies ist gefahrlos möglich, da kein anderer Mutator versuchen wird, einen Verknüpfungszeiger mit diesem Zustandszähler-Objekt an den ersten Mutator zu übertragen, bis der erste Mutator anderweitige Aktionen durchgeführt hat, die einen anderen Mutator dazu veranlassen.

Rückfordern nicht mehr benötigter Sperr- oder Anzeigeobjekte

[0259] Die Markierungsbits ×02 iterating_mutator_waiting und ×04 blocking_mutator_waiting innerhalb von Verknüpfungszeigern sowie die Felder active_inspector_view_count und active_mutator_view_count innerhalb des SharedChain-Objekts sind dazu gedacht anzugeben, wann ein Mutator die Liste der Anzeige- und/oder Sperrobjekte durchsuchen muss. Ein unnötiges Durchsuchen stellt ein nicht erwünschtes Übermaß an Leistung dar. Darüber sollten die Listen der Anzeige- und Sperrobjekte möglichst kurz gehalten werden, so dass keine übermäßig große Zahl inaktiver Objekte durchsucht werden muss, wenn ein Durchsuchen erforderlich ist. Vorzugsweise speichert das SharedChain-Objekt in jeder Liste eine Mindestzahl von Objekten (derzeit vier), und wenn es die Gelegenheit erhält, nicht mehr benötigte Objekte zurückzufordern, fordert es die Hälfte der zu diesem Zeitpunkt vorhandenen nicht mehr benötigten Objekte zurück. Die Absicht dahinter besteht darin, zusätzliche Objekte zwar zu löschen, dies jedoch nicht zu schnell zu tun.

[0260] Sperr- und Anzeigeobjekte werden in einfach verknüpften Listen verwaltet, so dass neue Objekte atomar an den Anfang der Liste aufgenommen werden können, ohne dass hierfür eine Sperre erforderlich ist. Objekte können entfernt werden, sobald keine ausstehenden Verweise mehr für sie vorhanden sind. Dabei ist bekannt, dass dieser Fall dann gegeben ist, wenn das Feld activated_blocker_view_counts innerhalb des SharedChain-Objekts bis auf Null dekrementiert wurde.

[0261] Wenn ein nichtatomarer Iterator die Verarbeitung seines (nichtatomaren) Anzeigeobjekts beendet hat

oder wenn ein Zustandszähler status_control eines Sperrobjekts auf Null gesetzt wird, so dass keine (weiteren) wartenden Objekte mehr einen Verweis darauf haben, wird der betreffende Teile des Felds activated_blocker_and_view_counts um den Wert 1 atomar erniedrigt. Wenn als Ergebnis der Dekrementierung der Wert des gesamten Felds jedoch Null lautet, wird anstelle der einfachen Dekrementierung zusätzlich die Markierung reclaim_in_progress (d.h. das höherwertige Bit des Felds) aktiv gesetzt, wenn es einen oder mehrere zusätzliche Objekte gibt.

[0262] Die folgenden Schritte werden verwendet, wenn das Feld activated_blocker_view_counts dekrementiert wird um zu versuchen, die Markierung reclaim_in_progress aktiv zu setzen und – wenn dies erfolgreich ist – die Reklamationsfunktion auszuführen:

1. Es wird eine vorübergehende Kopie des Felds activated_blocker_and_view_counts gespeichert, und die Dekrementierung wird für dieses Feld durchgeführt. Wenn das Ergebnis Null ist, fährt der Prozess mit Schritt 2 fort. Andernfalls wird mit compareAndSwap() versucht, die dekrementierte Kopie in das Feld zurückzuspeichern. Wenn der Versuch fehlschlägt, gibt dies an, dass das Feld zwischenzeitlich geändert wurde, so dass der Prozess zu Schritt 1 zurückkehrt und noch einmal von vorn beginnt. Wenn der Versuch erfolgreich ist, ist die Dekrementierung abgeschlossen, da jedoch noch ausstehende Verweise auf Anzeige- und/oder Sperrobjekte vorhanden ist (oder die Rückforderung bereits läuft), gibt es zu diesem Zeitpunkt nichts zurückzufordern, so dass der Prozess einfach beendet wird.
2. An dieser Stelle sind keine ausstehenden Verweise auf Sperr- und Anzeigeobjekte vorhanden, so dass eine Rückforderung erfolgen kann, falls es nicht mehr benötigte Objekte gibt. Hierfür wird (je nach Gültigkeit) eine vorübergehende Kopie des Felds primary_view_existence_count oder secondary_view_existence_count aus dem SharedChain-Objekt gespeichert. Dieses Feld enthält getrennt voneinander die Anzahl der vorhandenen Anzeigeobjekte für Inspektoren und für Mutatoren.
3. Es wird eine vorübergehende Kopie des Felds blocker_and_view_existence_counts aus dem SharedChain-Objekt gespeichert.
4. Anhand der in den Schritten 2 und 3 erhaltenen Felder wird ermittelt, ob nicht mehr benötigte Sperrobjekte, Anzeigeobjekte für Inspektoren oder Anzeigeobjekte für Mutatoren vorhanden sind. Wenn es wie auch immer geartete nicht mehr benötigte Objekte gibt, wird die Markierung reclaim_in_progress in der in Schritt 1 erhaltenen vorübergehenden Kopie des Felds activated_blocker_and_view_counts aktiv gesetzt.
5. Mit compareAndSwapQualified() wird versucht, die vorübergehende Kopie in das Feld activated_blocker_and_view_counts zurückzuspeichern. Wenn dies fehlschlägt (da entweder das Feld activated_blocker_and_view_counts oder das Feld blocker_and_view_existence_counts zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 1 zurück und beginnt von vorn. Wenn der Versuch erfolgreich ist, findet je nach den Ergebnissen aus Schritt 4 möglicherweise gerade eine Rückforderung statt. Wenn dies nicht der Fall ist, gibt es nichts zu tun, so dass der Prozess endet.
6. An dieser Stelle ist eine Rückforderung im Gange, und der Inhalt mehrerer Felder wurde zur späteren Verwendung aus dem SharedChain-Objekt erhalten. Zwischenzeitlich können die Felder selbst geändert werden, während die Rückforderungsverarbeitung versucht wird. Wenn es keine nicht mehr benötigten Sperrobjekte gibt, fährt der Prozess mit Schritt 10 fort.
7. Ausgehend von dem Zählwert, der in Schritt 3 erhalten wurde, sind nicht mehr benötigte Sperrobjekte vorhanden. Der Sperrobjekt-Teil des erhaltenen Felds blocker_and_view_existence_counts wird um die Hälfte der nicht mehr benötigten Sperrobjekte dekrementiert. Diese Änderung wird erst dann wirksam, wenn die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist.
8. Ausgehend vom ersten Sperrobjekt (je nach Gültigkeit aus dem Feld primary_blocker_list_head oder secondary_blocker_list_head) wird das nächste Sperrobjekt ausfindig gemacht, und der Prozess geht die Liste schrittweise so lange durch, bis die Anzahl der zurückzufordernden Sperrobjekte übersprungen wurde. Dabei handelt es sich bei den übersprungenen Sperrobjekten um diejenigen Sperrobjekte, die verworfen werden sollen, während das aktuelle Sperrobjekt dasjenige ist, das zum neuen ersten Sperrobjekt in der Liste werden soll. Die Liste kann mehr, jedoch nie weniger Sperrobjekte als erwartet enthalten. Der Zeiger auf das neue erste Sperrobjekt wird, je nachdem, welches Feld ungültig ist, im Feld primary_blocker_list_head oder secondary_blocker_list_head gespeichert. Diese Änderung bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist.
9. Das höherwertige Bit im Sperrobjekt-Teil des erhaltenen Felds blocker_and_view_existence_counts wird umgeschaltet. Dadurch wird die Gültigkeit der Felder primary_blocker_list_head und secondary_blocker_list_head getauscht. Dies bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist. Wenn sie jedoch erfolgreich ist, sind die zu löschenden Sperrobjekte in der Liste nicht mehr sichtbar und können nach Belieben verworfen werden.
10. Wenn keine nicht mehr benötigten Anzeigeobjekte für Inspektoren vorhanden sind, fährt der Prozess mit Schritt 14 fort.
11. Ausgehend von dem in Schritt 2 erhaltenen Zählwert sind nicht mehr benötigte Anzeigeobjekte für In-

spektoren vorhanden. Der Inspektor-Teil des erhaltenen Felds primary_view_existence_counts oder secondary_view_existence_counts wird um die Hälfte der nicht mehr benötigten Anzeigeobjekte dekrementiert. Darüber hinaus wird auch der Anzeigeobjekt-Teil des erhaltenen Felds extracted_blocker_and_view_existence um dieselbe Anzahl dekrementiert. Diese Änderungen bleiben so lange ohne Wirkung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist.

12. Ähnlich der für Sperrobjekte beschriebenen Verfahrensweise wird die Anzahl der zurückzufordernden Anzeigeobjekte übersprungen, um das Anzeigeobjekt zu finden, das zum neuen ersten Anzeigeobjekt in der Liste werden soll. Auch hier kann die Liste mehr, jedoch nie weniger Inspektor-Anzeigeobjekte enthalten als erwartet. Der Zeiger auf das neue erste Inspektor-Anzeigeobjekt wird im Feld primary_inspector_view_chain_head oder secondary_inspector_view_chain_head gespeichert, je nachdem, welches Feld ungültig ist. Diese Änderung bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist.

13. Das höherwertige Bit im Inspektor-Teil des erhaltenen Felds primary_view_existence_counts oder secondary_view_existence_counts wird umgeschaltet. Dadurch wird die Gültigkeit der Felder primary_inspector_view_list_head und secondary_inspector_view_list_head getauscht. Dies bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist. Wenn sie jedoch erfolgreich ist, sind die zu löschenden Anzeigeobjekte in der Liste nicht mehr sichtbar und können nach Belieben verworfen werden.

14. Wenn keine nicht mehr benötigten Anzeigeobjekte für Mutatoren vorhanden sind, fährt der Prozess mit Schritt 15 fort. Andernfalls werden die Mutator-Anzeigeobjekte entsprechend der in den Schritten 11 bis 13 für Inspektor-Anzeigeobjekte beschriebenen Verfahrensweise verarbeitet.

15. Wenn nicht mehr benötigte Anzeigeobjekte für Inspektoren oder Mutatoren vorhanden waren, wird je nachdem, welches Feld ungültig ist, das erhaltene/aktualisierte Feld primary_view_existence_counts oder secondary_view_existence_counts in das Feld primary_view_existence_counts oder secondary_view_existence_counts innerhalb des SharedChain-Objekts kopiert. Dieses aktualisierte Feld enthält aktualisierte Zählwerte für Inspektor-Anzeigeobjekte sowie den Anzeiger, der angibt, ob primary_mutator_list_head oder secondary_mutator_list_head gültig ist. Dies bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist. Wenn nicht mehr benötigte Anzeigeobjekte für Inspektoren oder Mutatoren vorhanden waren, wird zusätzlich das höherwertige Bit im Anzeigeobjekt-Teil des erhaltenen Felds blocker_and_view_existence umgeschaltet. Dadurch wird die Gültigkeit der Felder primary_view_existence_counts und secondary_view_existence_counts getauscht. Dies bleibt so lange ohne Bedeutung, bis die Operation compareAndSwapQualified() in Schritt 16 erfolgreich ist.

16. Dieser Schritt ist der atomare Bestandteil der gesamten Rückforderungsoperation. Mit compareAndSwapQualified() wird versucht, das Feld blocker_and_view_existence_counts durch die Aktualisierungen, die an dem erhaltenen Wert vorgenommen wurden, atomar zu ersetzen, während gleichzeitig sichergestellt wird, dass weder dieses noch das Feld activated_blocker_and_view_counts zwischenzeitlich geändert wurde. Wenn die Operation erfolgreich ist, fährt der Prozess mit Schritt 20 fort. Andernfalls ist der Versuch, die Änderungen zu übergeben, fehlgeschlagen, da es zum Zeitpunkt der Operation aktive Sperrobjekte und/oder Anzeigeobjekte gab bzw. es nun mehr Sperr- und/oder Anzeigeobjekt gibt als zu Beginn der Rückforderungsverarbeitung.

17. Wie in Schritt 2 wird das Feld primary_view_existence_counts oder secondary_view_existence_counts (je nach Gültigkeit) aus dem SharedChain-Objekt erhalten. Danach wird wie in Schritt 3 das Feld blocker_and_view_existence_counts erhalten.

18. Das Feld activated_blocker_and_view_counts wird aus dem SharedChain-Objekt erhalten. Wenn es mit Ausnahme der Markierung reclaim_in_progress nicht nur Null-Werte enthält, fährt der Prozess mit Schritt 19 fort. Andernfalls enthält es mit Ausnahme der Markierung reclaim_in_progress ausschließlich Null-Werte, so dass die Reklamationsverarbeitung erneut versucht wird. Mit compareAndSwapQualified() wird das Feld activated_blocker_and_viewcounts zurückgespeichert, wobei überprüft wird, ob dieses und das Feld blocker_and_view_existence_counts noch gültig ist. Wenn die Operation fehlschlägt, fährt der Prozess mit Schritt 17 fort und unternimmt einen erneuten Versuch. Wenn der Versuch dagegen erfolgreich ist, fährt der Prozess mit Schritt 6 fort und versucht die Rückforderungsverarbeitung erneut.

19. Da nun aktive Sperr- oder Anzeigeobjekte vorhanden sind, muss die Rückforderungsfunktion so lange warten, bis sie alle inaktiv sind, so dass mit compareAndSwapQualified() versucht wird, die Markierung reclaim_in_progress inaktiv zu setzen. Wenn diese Operation fehlschlägt (da entweder das Feld activated_blocker_and_view_counts oder das Feld blocker_and_view_existence_counts zwischenzeitlich geändert wurde), kehrt der Prozess zu Schritt 17 zurück und unternimmt einen erneuten Versuch. Wenn der Versuch erfolgreich ist, endet der Prozess; die Rückforderungsoperation wird erneut versucht, sobald das Feld activated_blocker_and_view_existence_counts ausschließlich aus Null-Werten besteht.

20. An dieser Stelle wurde die Rückforderung erfolgreich durchgeführt. Wenn Sperrobjekte zurückgefordert wurden, hat das umgeschaltete höherwertige Bit im Sperrobjekt-Teil des Felds

blocker_and_view_existence_counts dazu geführt, dass die Felder primary_blocker_list_head und secondary_blocker_list_head ihre Gültigkeit getauscht haben, und der neue gültige Listenkopf hat die Sperrobjekte, die gelöscht werden sollten, übersprungen. Wenn Anzeigeobjekte zurückgefordert wurden, hat das umgeschaltete höherwertige Bit im Anzeigeobjekt-Teil des Felds blocker_and_view_existence_counts entsprechend dazu geführt, dass die Felder primary_view_existence_counts und secondary_view_existence_counts ihre Gültigkeit getauscht haben, und das umgeschaltete höherwertige Bit in jedem Teil des neuen gültigen Felds gibt unabhängig davon an, ob das Feld primary_chain_head oder secondary_chain_head für Inspektor- bzw. Mutator-Anzeigeobjekte nun gültig ist. Ein neues gültiges Listenkopffeld für Inspektor- bzw. Mutator-Anzeigeobjekte hat die zu löschenden Anzeigeobjekte übersprungen.

21. Mit compareAndSwap() wird versucht, die Markierung reclaim_in_progress im Feld activated_blocker_and_view_counts inaktiv zu setzen. Wenn die Operation fehlschlägt, wird sie so lange wiederholt, bis sie erfolgreich ist.

22. Die nicht mehr benötigten Sperrobjekte, Inspektor- und/oder Mutator-Anzeigeobjekte, die in den obigen Schritten zurückgefordert wurden, werden gelöscht.

Patentansprüche

1. Verfahren für die Verwaltung einer verknüpften Liste, das die folgenden Schritte umfasst:
Erstellen einer Hilfsdatenstruktur für die verknüpfte Liste, wobei die Hilfsdatenstruktur eine Liste von Umgehungselementen umfasst und jedes Umgehungselement einen Zeiger auf ein erstes Element eines Teils der Liste beinhaltet;
Veranlassen einer ersten Aktualisierungsaufgabe für das Aktualisieren der verknüpften Liste über die Hilfsdatenstruktur, indem eine Aktion aus der Gruppe von Aktionen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen eines Elements innerhalb des ersten Teils der verknüpften Liste aus der verknüpften Liste und (b) Einfügen eines Elements in die verknüpfte Liste innerhalb des ersten Teils der verknüpften Liste;
Aufzeichnen von Sperrdaten innerhalb der Umgehungselemente der Hilfsdatenstruktur, um zu verhindern, dass eine zweite Aktualisierungsaufgabe, die über die Hilfsdatenstruktur auf die verknüpfte Liste zugreift, einen ersten Teil der verknüpften Liste ändert, wobei der erste Teil nicht alle Elemente der verknüpften Liste umfasst;
wobei der Schritt des Aufzeichnens von Sperrdaten das Ändern des Vorwärtszeigers von dem vorhergehenden Listenelement in ein erstes Listenelement des ersten Teils der verknüpften Liste umfasst, um so stattdessen auf ein Umgehungselement zu verweisen, wobei eine Aufgabe, welche die verknüpfte Liste in einer Vorwärtsrichtung durchläuft, ohne die verknüpfte Liste jedoch zu ändern, die verknüpfte Liste über das Umgehungselement zu dem ersten Teil der verknüpften Liste durchläuft; und
wobei eine zweite Aktualisierungsaufgabe daran gehindert wird, die verknüpfte Liste in einer Vorwärtsrichtung zu aktualisieren, und abwartet, bis die erste Aktualisierungsaufgabe die Sperrdaten freigibt.

2. Verfahren nach Anspruch 1, wobei die verknüpfte Liste eine doppelt verknüpfte Liste ist, wobei ein Rückwärtszeiger von einem folgenden Listenelement zu einem letzten Listenelement des ersten Teils der verknüpften Liste so geändert wird, dass er stattdessen auf das Umgehungselement verweist, und wobei eine Aufgabe, welche die verknüpfte Liste in einer Rückwärtsrichtung durchläuft, ohne die verknüpfte Liste jedoch zu ändern, die verknüpfte Liste über das Umgehungselement zu dem ersten Teil der verknüpften Liste durchläuft; und wobei die zweite Aktualisierungsaufgabe daran gehindert wird, die erste verknüpfte Liste in einer Rückwärtsrichtung zu aktualisieren, und abwartet, bis die erste Aktualisierungsaufgabe die Sperrdaten freigibt.

3. Verfahren nach Anspruch 1 oder 2, das weiter die folgenden Schritte umfasst:
Veranlassen einer weiteren Aktualisierungsaufgabe für das Aktualisieren eines zweiten Teils der verknüpften Liste über die Hilfsdatenstruktur;
Aufzeichnen von Sperrdaten innerhalb der Hilfsdatenstruktur, um zu verhindern, dass Aufgaben mit Ausnahme der weiteren Aktualisierungsaufgabe, die über die Hilfsdatenstruktur auf die verknüpfte Liste zugreifen, den zweiten Teil der verknüpften Liste ändern, wobei der zweite Teil nicht alle Elemente der verknüpften Liste umfasst; und
Aktualisieren der verknüpften Liste innerhalb des zweiten Teils der verknüpften Liste, indem eine Aktion aus einer Gruppe von Aktionen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen eines Elements innerhalb des zweiten Teils der verknüpften Liste aus der verknüpften Liste und (b) Einfügen eines Elements in die verknüpfte Liste innerhalb des zweiten Teils der verknüpften Liste, wobei der Schritt des Aktualisierens der verknüpften Liste innerhalb des zweiten Teils gleichzeitig mit dem Schritt des Aktualisierens der verknüpften Liste innerhalb des ersten Teils ausgeführt wird.

4. Verfahren nach einem beliebigen vorangegangenen Anspruch, das weiter den folgenden Schritt um-

fasst:
Weitergeben eines Zeigers auf eine frei wählbare Verknüpfung innerhalb der verknüpften Liste, um einen Punkt für das Ändern der verknüpften Liste zu finden, wobei der erste Teil, der durch den Schritt des Aufzeichnens von Sperrdaten gesperrt wird, zumindest einen Teil der frei wählbaren Verknüpfung enthält.

5. Verfahren nach einem beliebigen vorangegangenen Anspruch, wobei die Hilfsdatenstruktur ein Objekt in einer Klassenhierarchie einer objektorientierten Programmierstruktur ist und wobei die Klassenhierarchie eine erste Klasse für das Durchlaufen der verknüpften Liste, ohne sie zu ändern, und eine zweite Klasse für Objekte enthält, welche die verknüpfte Liste durchlaufen und sie ändern.

6. Verfahren nach Anspruch 5, wobei die Klassenhierarchie weiter eine dritte Klasse für Objekte, welche die verknüpfte Liste durchlaufen, ohne sie zu ändern, und über einen Sperrzugriff auf Listenebene verfügen, eine vierte Klasse für Objekte, welche die verknüpfte Liste durchlaufen, ohne sie zu ändern, und über einen Sperrzugriff auf Verknüpfungsebene verfügen, eine fünfte Klasse für Objekte, welche die verknüpfte Liste durchlaufen und sie ändern und über einen Sperrzugriff auf Listenebene verfügen, sowie eine sechste Klasse für Objekte, welche die verknüpfte Liste durchlaufen und sie ändern und über einen Sperrzugriff auf Verknüpfungsebene verfügen, enthält und wobei die dritten und vierten Klassen von der ersten Klasse erben und die fünften und sechsten Klassen von der zweiten Klasse erben.

7. Verfahren nach einem beliebigen der vorangegangenen Ansprüche 2 bis 6, das die folgenden Schritte umfasst:
Kenntlichmachen einer ersten Gruppe von aufeinanderfolgenden Verknüpfungen in einem relevanten Teil der doppelt verknüpften Liste, wobei die erste Gruppe von aufeinanderfolgenden Verknüpfungen nicht alle Verknüpfungen der doppelt verknüpften Liste umfasst;
Verhindern, dass Aufgaben mit Ausnahme einer ersten Aufgabe zumindest einen Teil von Verknüpfungen der ersten Gruppe von aufeinanderfolgenden Verknüpfungen ändern, und gleichzeitig Zulassen, dass Aufgaben mit Ausnahme der ersten Aufgabe Verknüpfungen der doppelt verknüpften Liste, die nicht in der ersten Gruppe enthalten sind, ändern;
Ändern der doppelt verknüpften Liste innerhalb der ersten Gruppe, indem eine Operation aus der Gruppe von Operationen ausgeführt wird, die aus Folgendem besteht: (a) Entfernen einer Verknüpfung der ersten Gruppe aus der doppelt verknüpften Liste und (b) Einfügen einer Verknüpfung in die doppelt verknüpfte Liste zwischen zwei Verknüpfungen der ersten Gruppe, wobei die Operation von der ersten Ausgabe ausgeführt wird; und
Aufheben der Sperre, die Aufgaben mit Ausnahme der ersten Aufgabe daran hindert, zumindest einen Teil der Verknüpfungen der ersten Gruppe zu ändern, nachdem der Änderungsschritt ausgeführt wurde.

8. Verfahren nach Anspruch 7, das weiter die folgenden Schritte umfasst:
Kenntlichmachen einer zweiten Gruppe von aufeinanderfolgenden Verknüpfungen in einem relevanten Teil der doppelt verknüpften Liste, wobei die zweite Gruppe von aufeinanderfolgenden Verknüpfungen nicht alle Verknüpfungen der doppelt verknüpften Liste umfasst;
Verhindern, dass Aufgaben mit Ausnahme einer zweiten Aufgabe zumindest einen Teil von Verknüpfungen der zweiten Gruppe von aufeinanderfolgenden Verknüpfungen ändern, und gleichzeitig Zulassen, dass Aufgaben mit Ausnahme der zweiten Aufgabe Verknüpfungen der doppelt verknüpften Liste, die nicht in der zweiten Gruppe enthalten sind, ändern;
Ändern der doppelt verknüpften Liste innerhalb der zweiten Gruppe, indem eine Operation aus der Gruppe von Operationen ausgeführt wird, die Folgendes umfasst: (a) Entfernen einer Verknüpfung der zweiten Gruppe aus der doppelt verknüpften Liste und (b) Einfügen einer Verknüpfung in die doppelt verknüpfte Liste zwischen zwei Verknüpfungen der zweiten Gruppe, wobei die Operation durch die zweite Aufgabe ausgeführt wird; und
Aufheben der Sperre, die Aufgaben mit Ausnahme der zweiten Aufgabe daran hindert, zumindest einen Teil der Verknüpfungen der zweiten Gruppe zu ändern, nachdem der Schritt des Änderns der doppelt verknüpften Liste innerhalb der zweiten Gruppe ausgeführt wurde;
wobei der Schritt des Änderns der doppelt verknüpften Liste innerhalb der zweiten Gruppe nach dem Schritt, mit dem Aufgaben mit Ausnahme der ersten Aufgabe daran gehindert werden, Verknüpfungen der ersten Gruppe zu ändern, und vor dem Schritt ausgeführt wird, mit dem die Sperre, die Aufgaben mit Ausnahme der ersten Aufgabe daran gehindert hat, Verknüpfungen der ersten Gruppe zu ändern, aufgehoben wird.

9. Computerprogramm, das, wenn es in einen Computer geladen und ausgeführt wird, die Schritte eines Verfahrens gemäß einem beliebigen vorangegangenen Anspruch ausführt.

Es folgen 16 Blatt Zeichnungen

Anhängende Zeichnungen

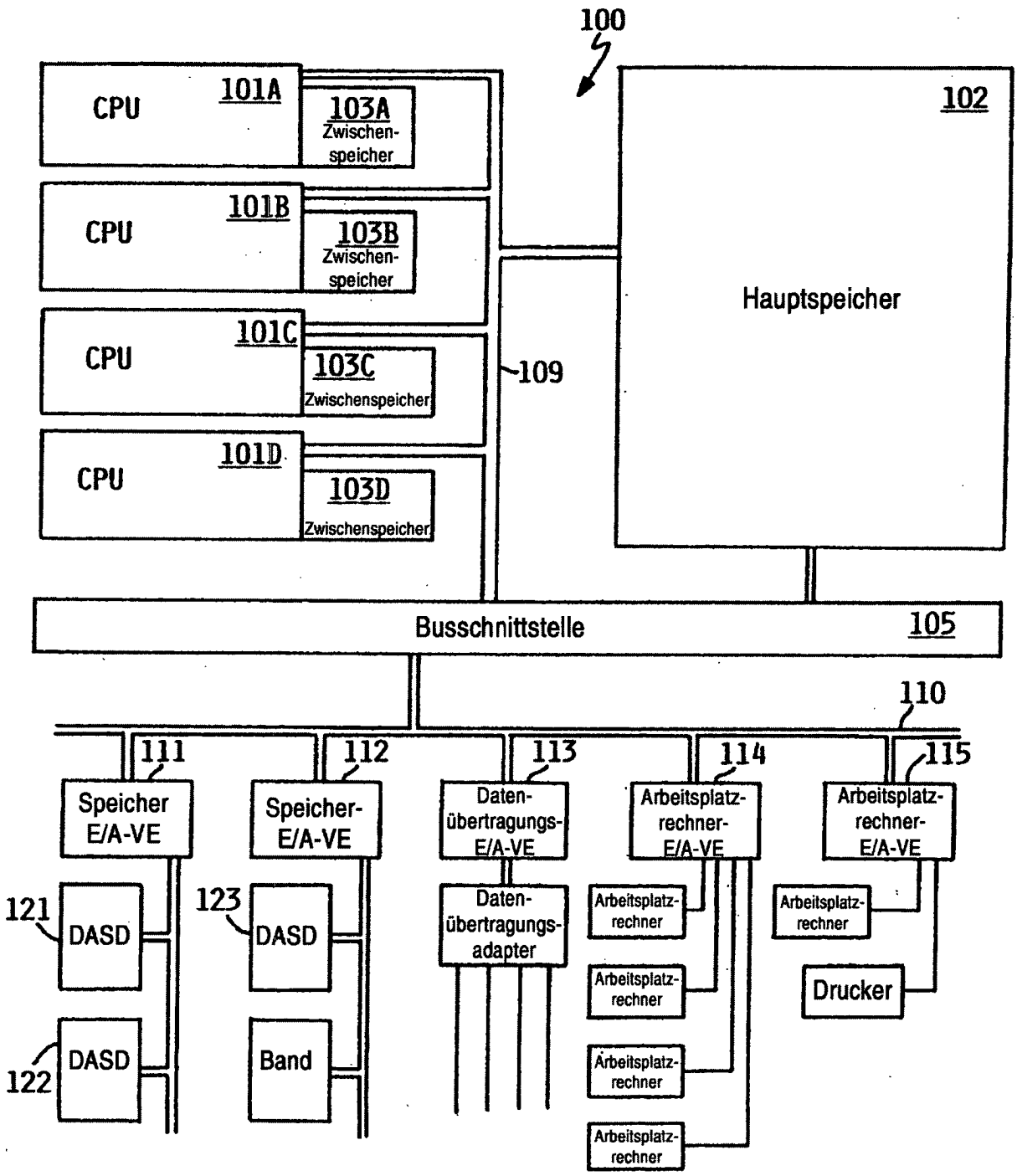

FIG. 1

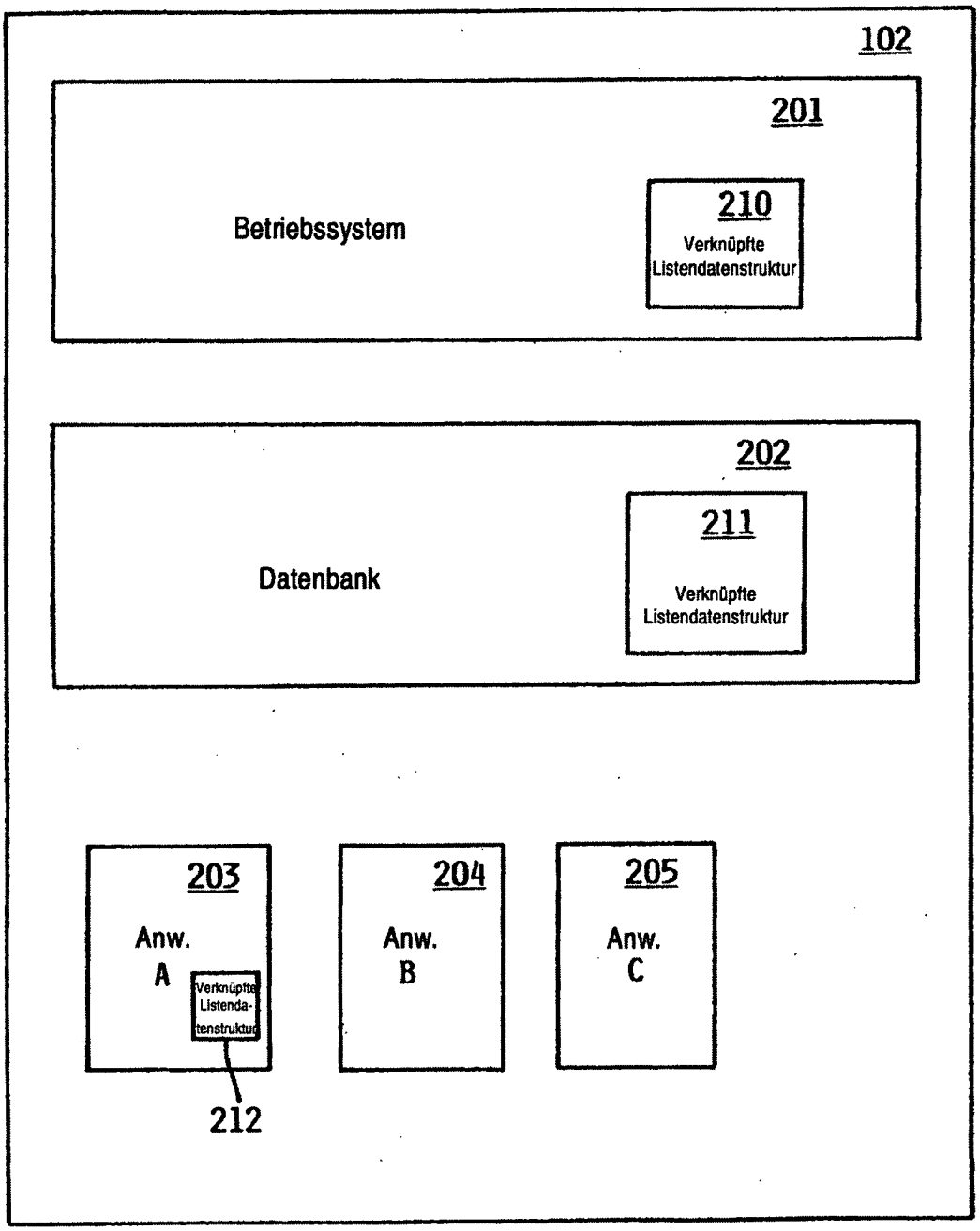
102
201
Betriebssystem
210
Verknüpfte Listendatenstruktur
202
Datenbank
211
Verknüpfte Listendatenstruktur
203
Anw. A
Verknüpfte Listenda-tenstruktur
212
204
Anw. B
205
Anw. C

FIG. 2

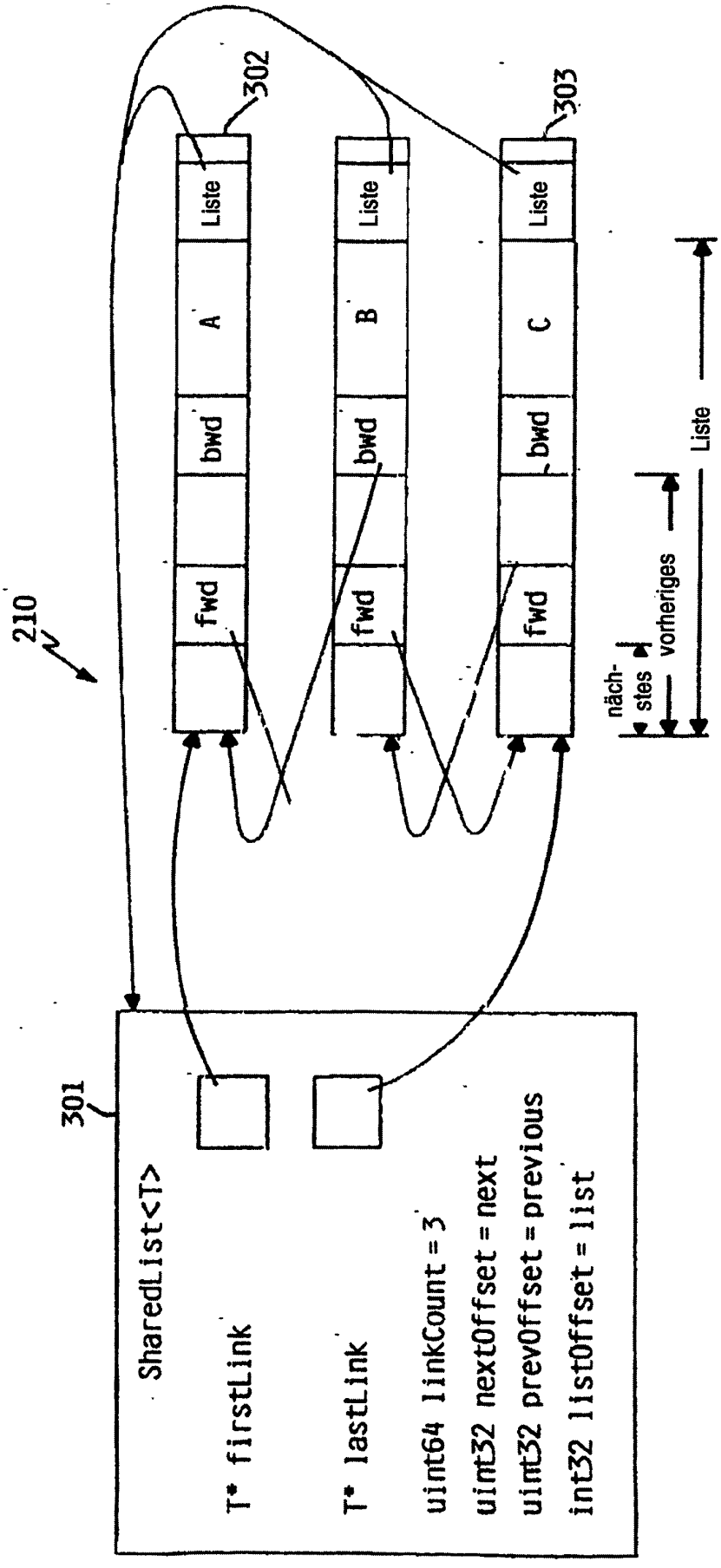
210
301
302
303
SharedList<T>
T* firstLink
T* lastLink
uint64 linkCount = 3
uint32 nextOffset = next
uint32 prevOffset = previous
int32 listOffset = list
Liste
A
B
C
bwd
fwd
nächstes
vorheriges
Liste

FIG. 3

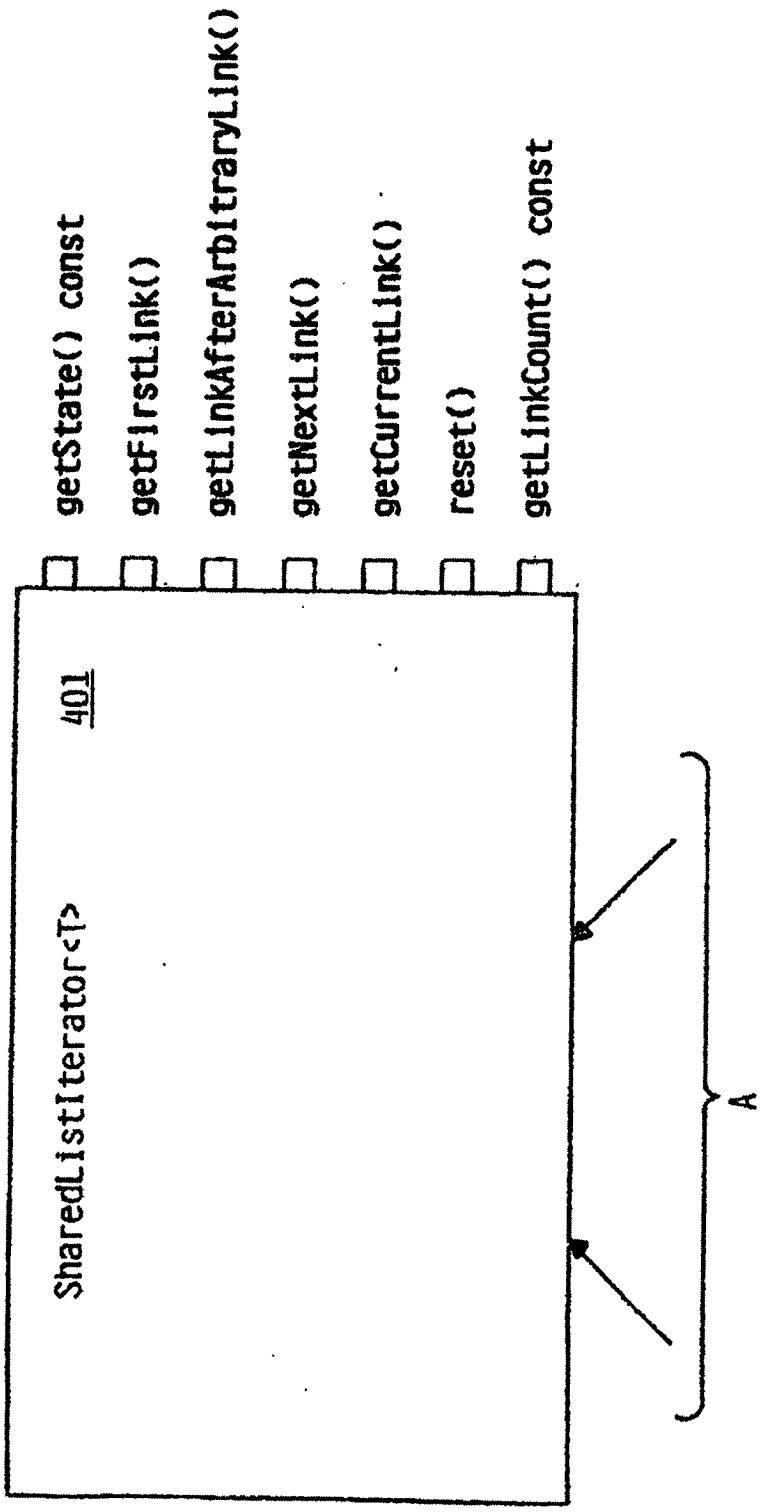
getState() const
getFirstLink()
getLinkAfterArbitraryLink()
getNextLink()
getCurrentLink()
reset()
getLinkCount() const
SharedListIterator<T>
401
A

FIG. 4A

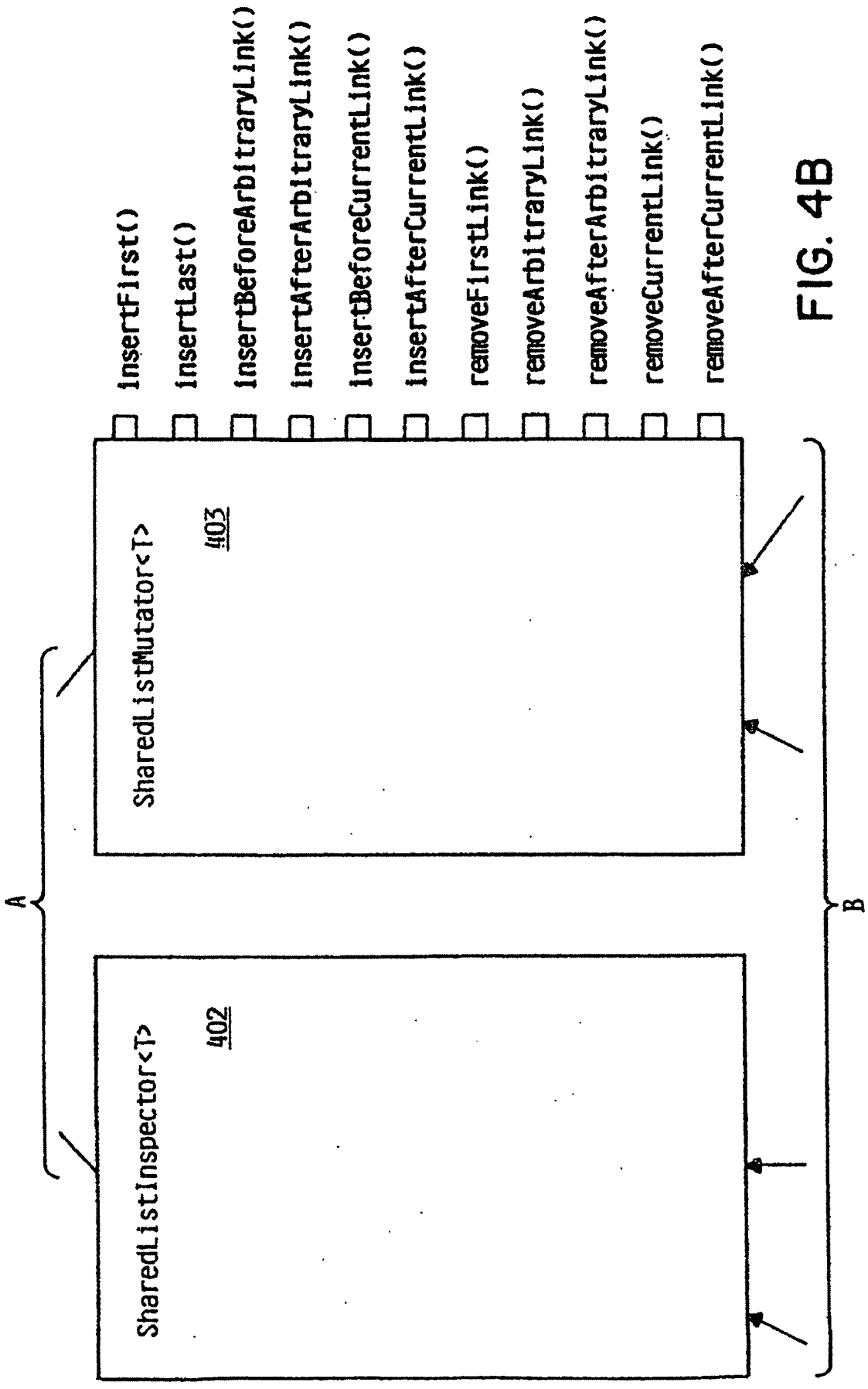
SharedListInspector<T>
402
SharedListMutator<T>
403
insertFirst()
insertLast()
insertBeforeArbitraryLink()
insertAfterArbitraryLink()
insertBeforeCurrentLink()
insertAfterCurrentLink()
removeFirstLink()
removeArbitraryLink()
removeAfterArbitraryLink()
removeCurrentLink()
removeAfterCurrentLink()
A
B

FIG. 4B

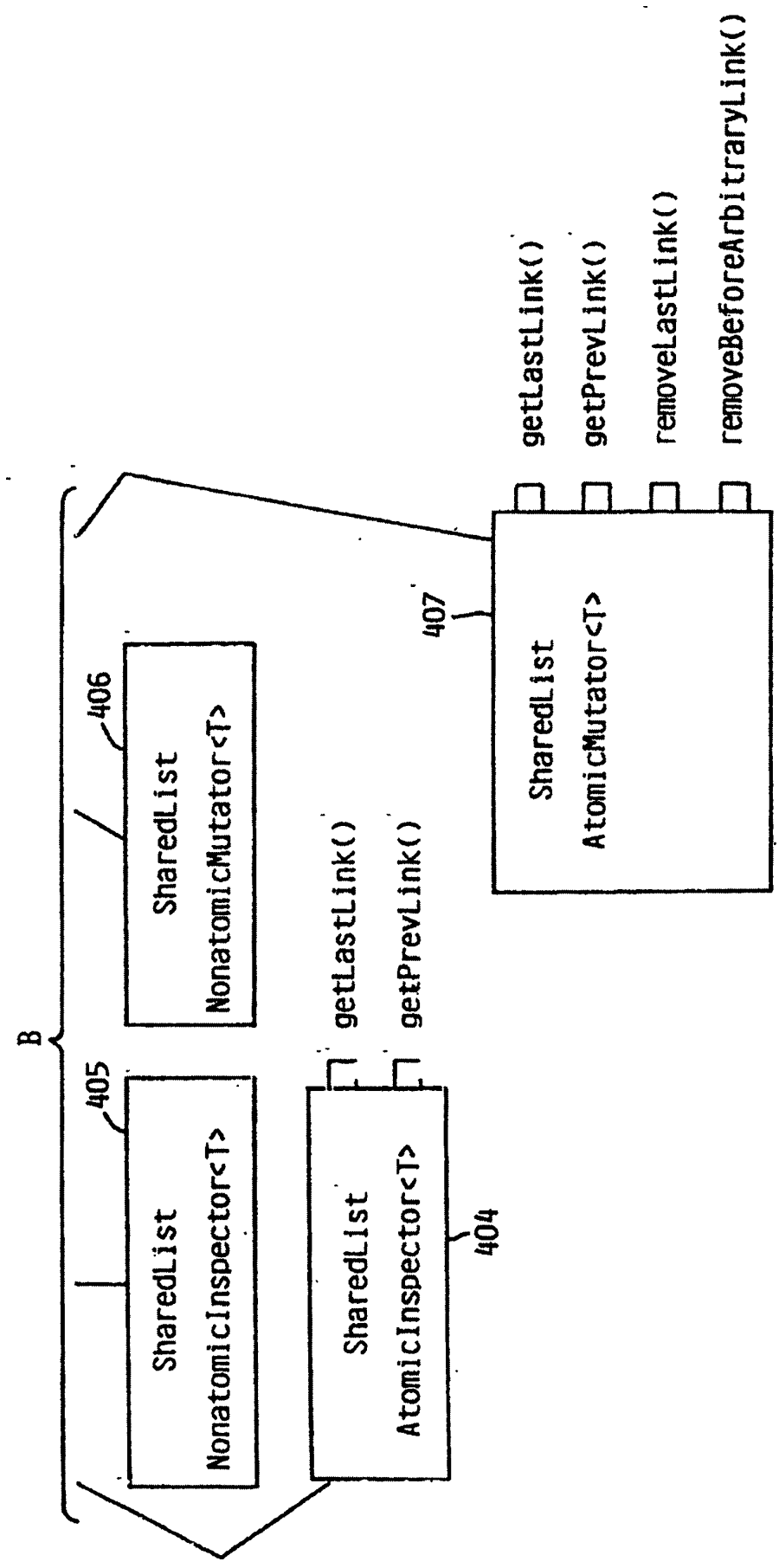
B
405
SharedList
NonatomicInspector<T>
406
SharedList
NonatomicMutator<T>
404
SharedList
AtomicInspector<T>
getLastLink()
getPrevLink()
407
SharedList
AtomicMutator<T>
getLastLink()
getPrevLink()
removeLastLink()
removeBeforeArbitraryLink()

FIG. 4C

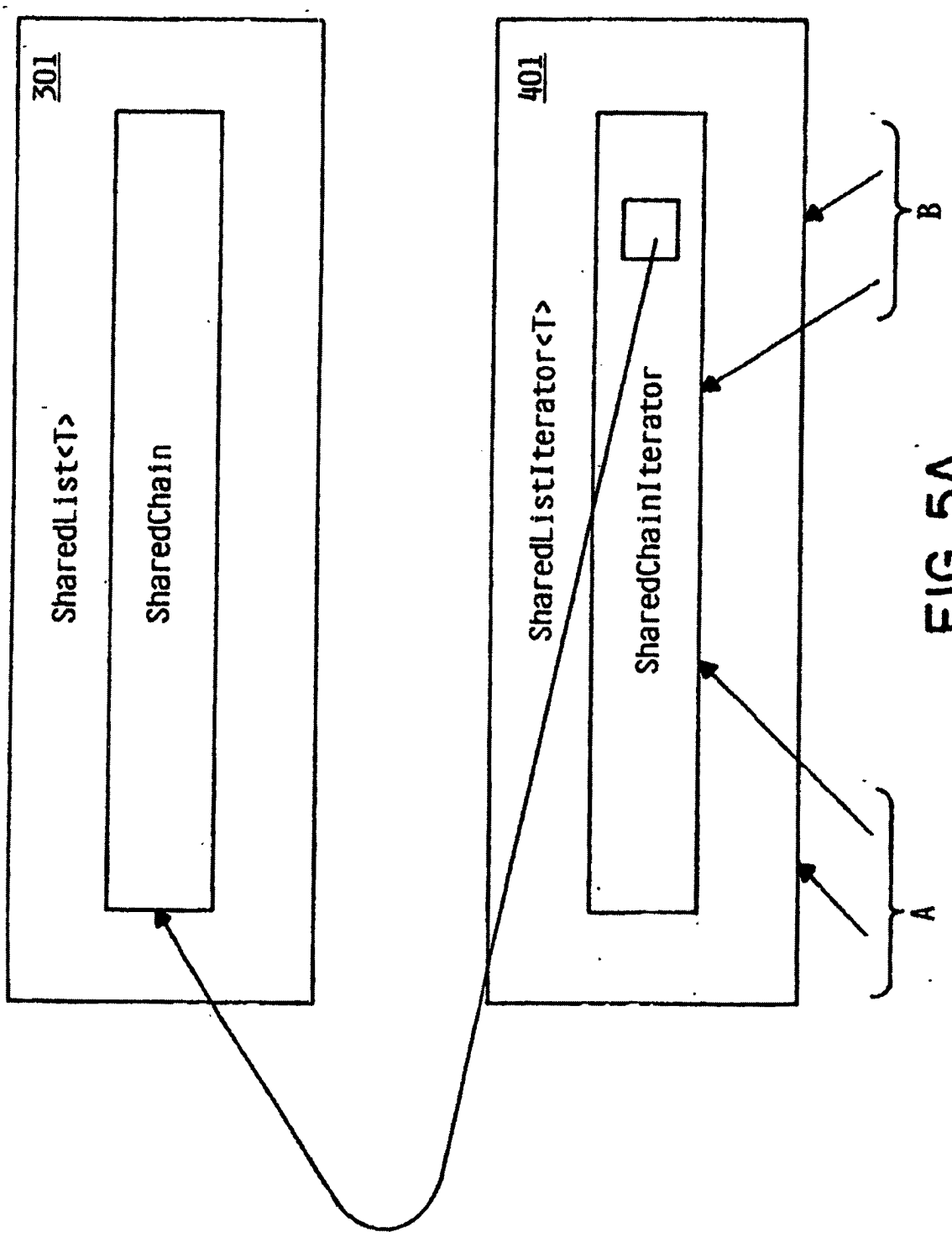
301
SharedList<T>
SharedChain
401
SharedListIterator<T>
SharedChainIterator
A
B

FIG. 5A

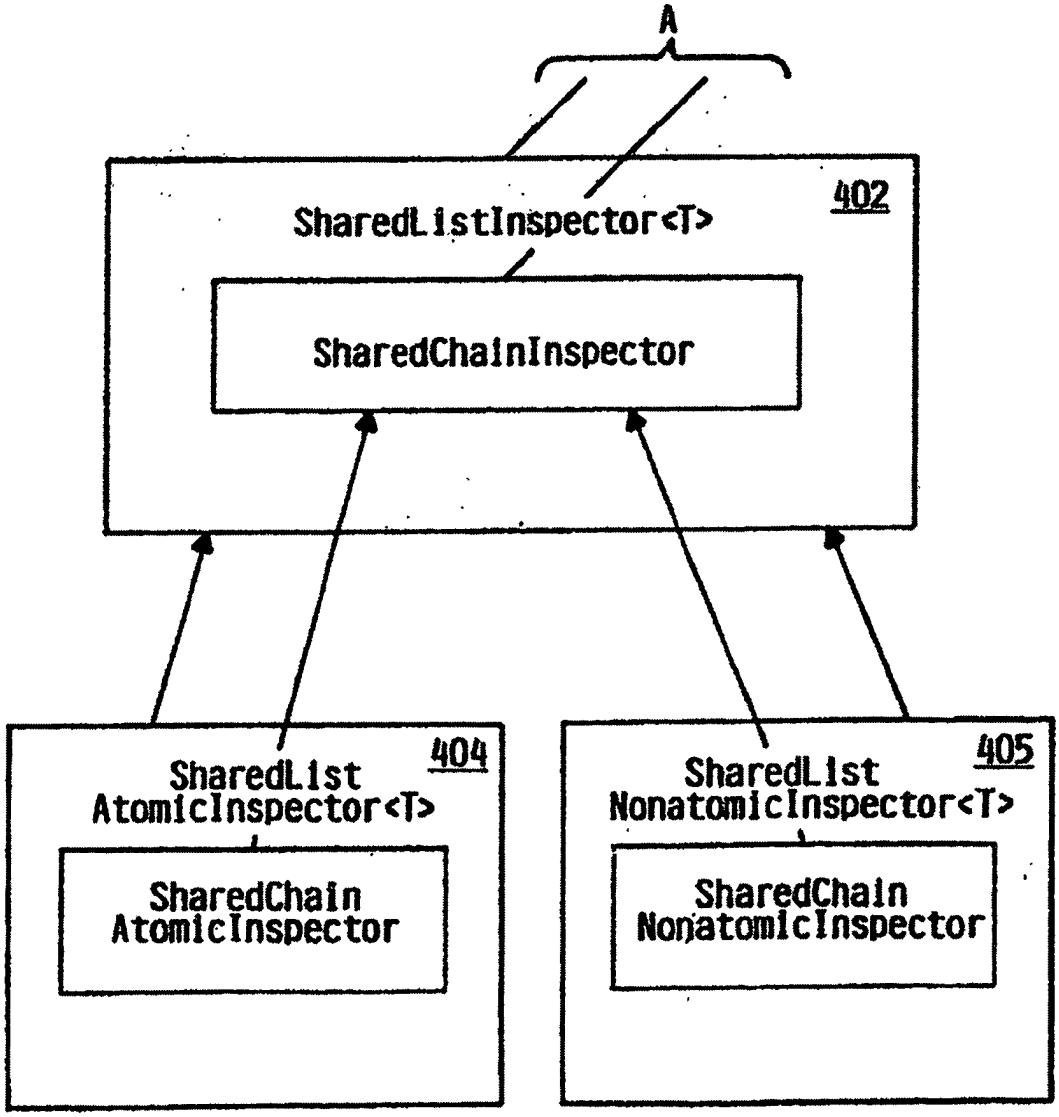
A
SharedListInspector<T>
402
SharedChainInspector
SharedList
AtomicInspector<T>
404
SharedChain
AtomicInspector
SharedList
NonatomicInspector<T>
405
SharedChain
NonatomicInspector

FIG. 5B

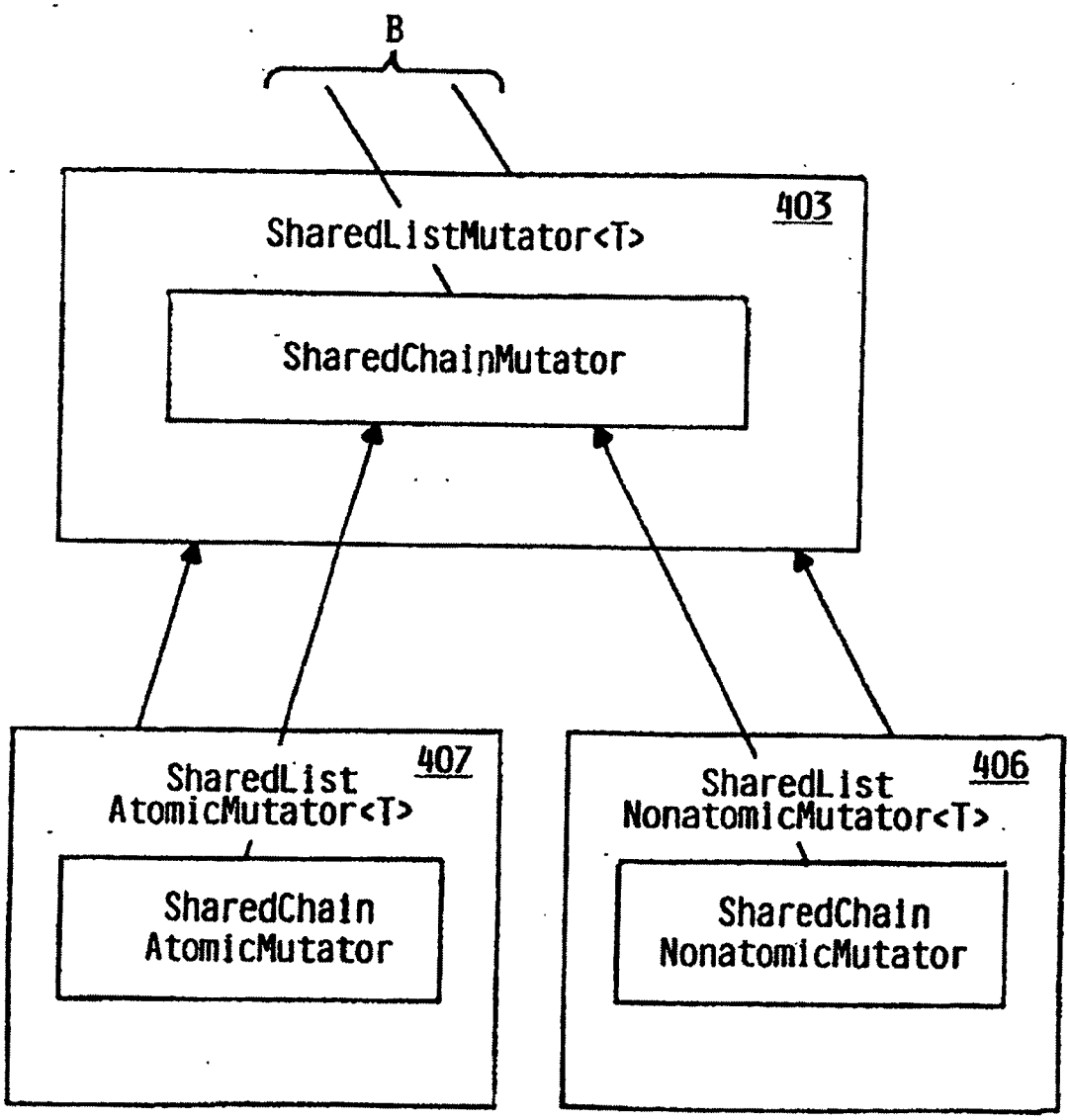
B
403
SharedListMutator<T>
SharedChainMutator
407
SharedList
AtomicMutator<T>
SharedChain
AtomicMutator
406
SharedList
NonatomicMutator<T>
SharedChain
NonatomicMutator

FIG. 5C

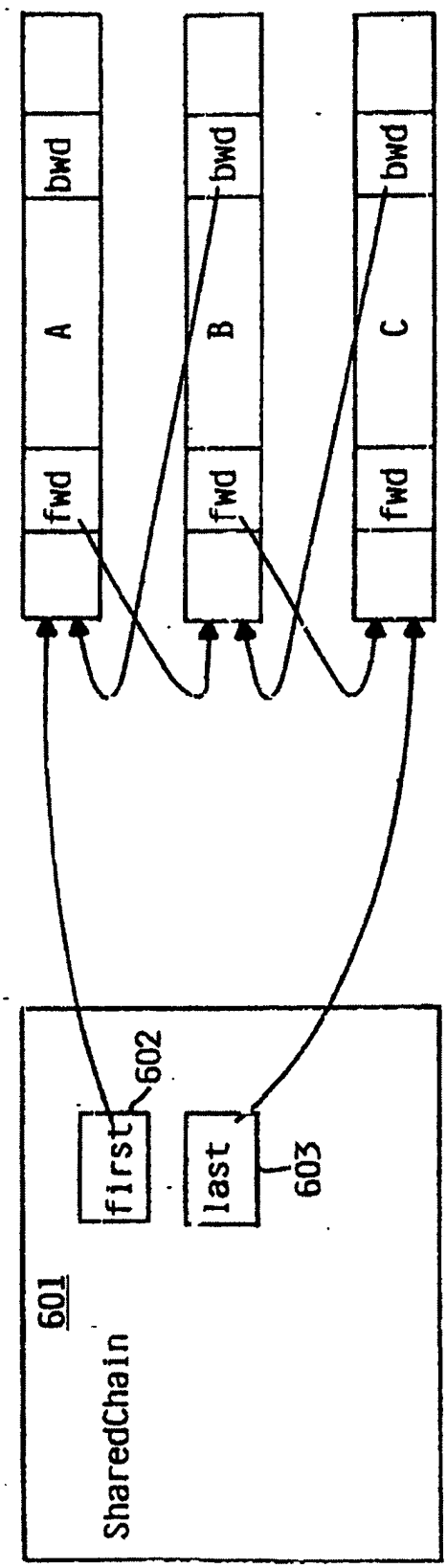
bwd
A
fwd
bwd
B
fwd
bwd
C
fwd
602
first
last
603
601
SharedChain

FIG. 6

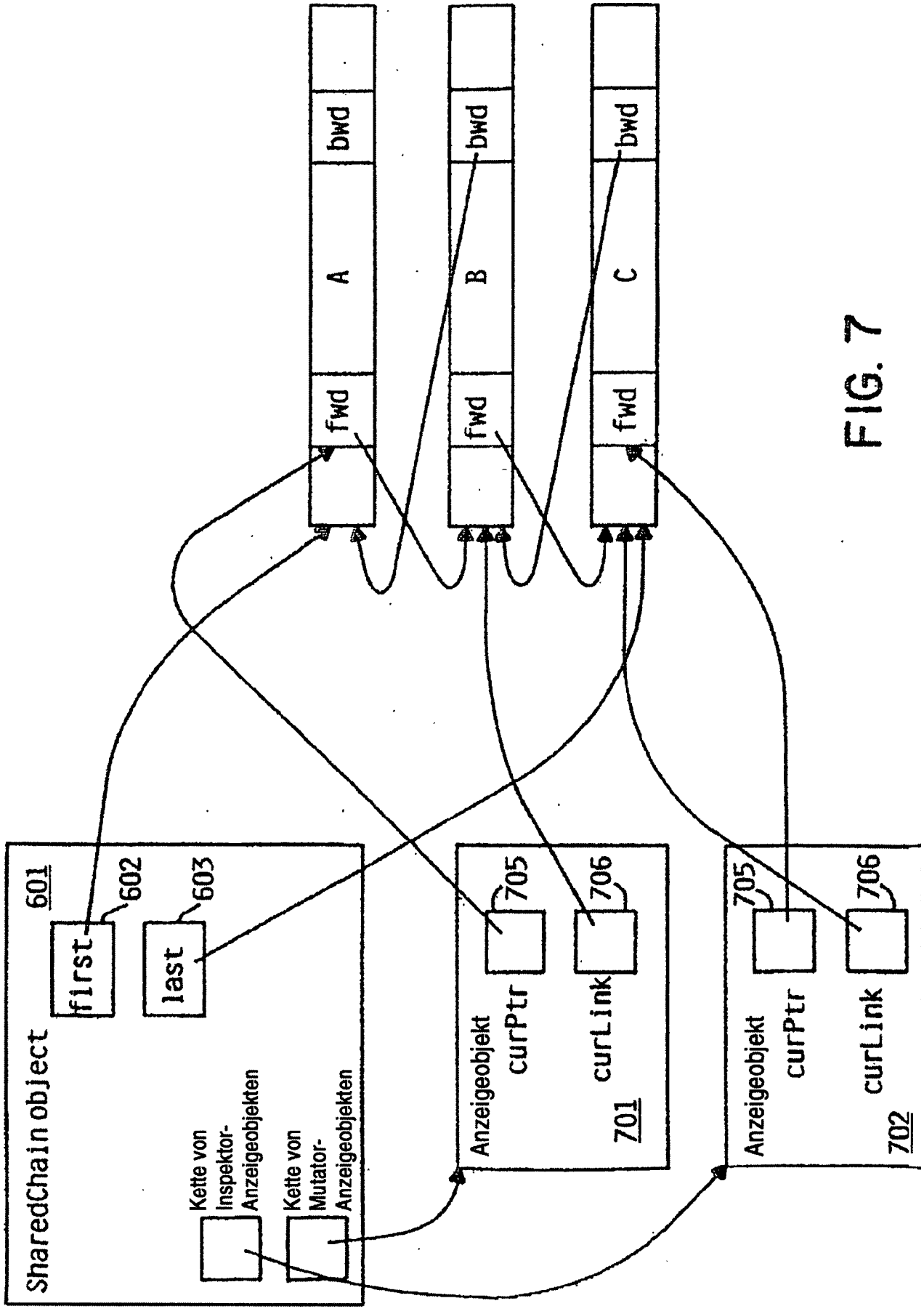
SharedChain object
601
first
602
last
603
Kette von Inspektor-Anzeigeobjekten
Kette von Mutator-Anzeigeobjekten
fwd
bwd
A
B
C
Anzeigeobjekt
curPtr
curLink
705
706
701
702

FIG. 7

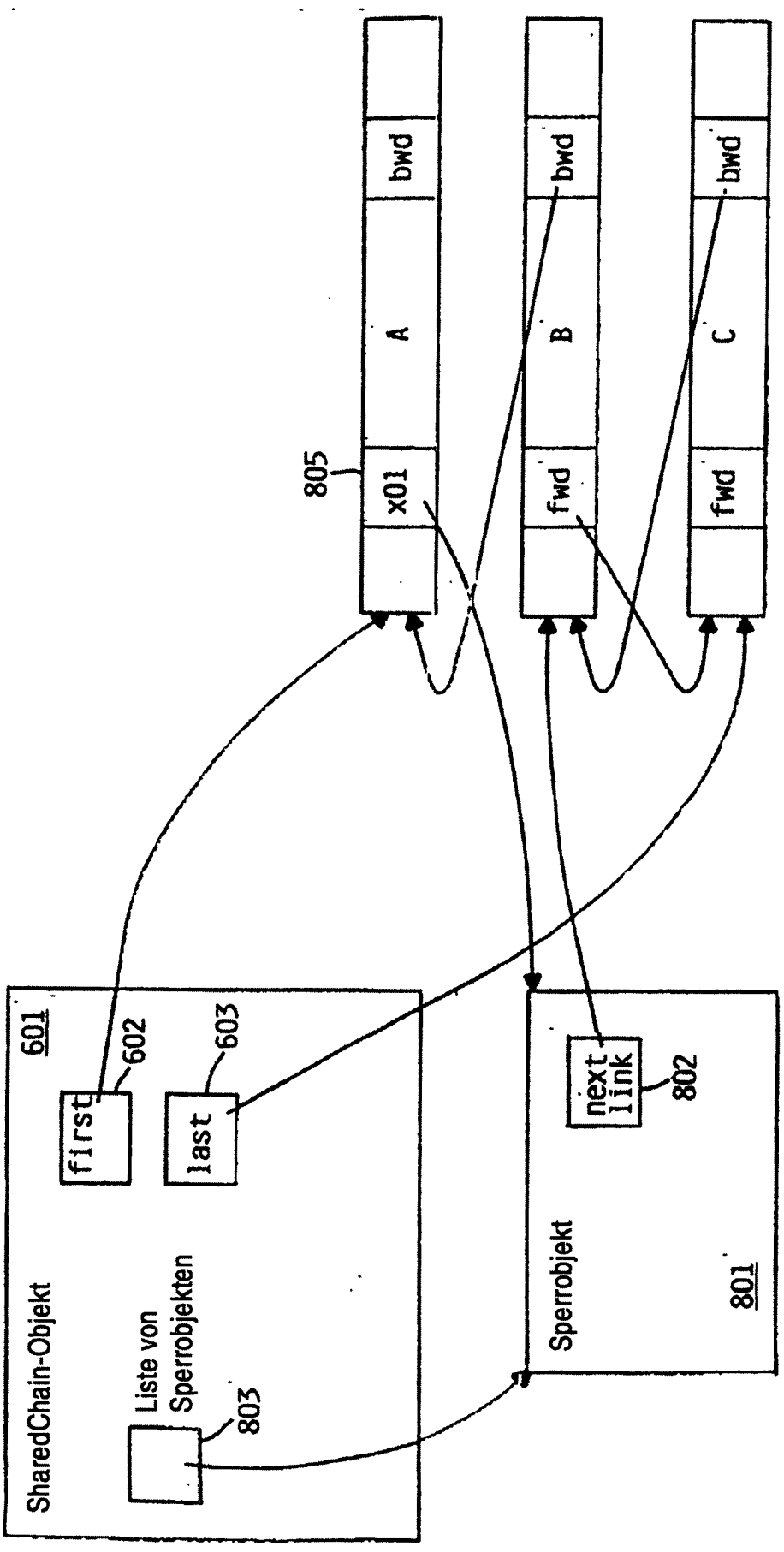
805
x01
fwd
bwd
A
B
C
601
602
603
first
last
SharedChain-Objekt
Liste von Sperrobjekten
803
next link
802
Sperrobjekt
801

FIG. 8

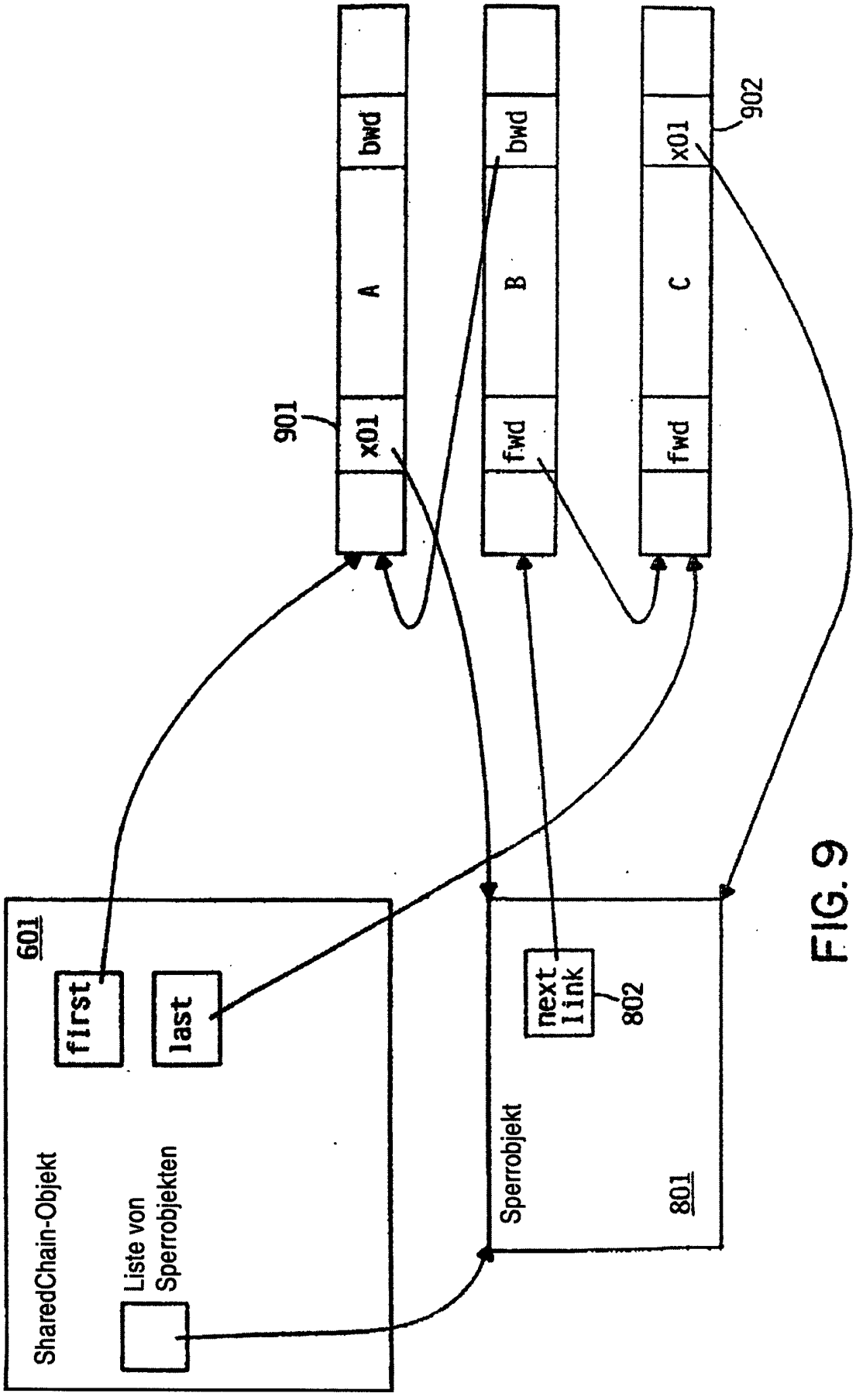
601
first
last
SharedChain-Objekt
Liste von Sperrobjekten
Sperrobjekt
next link
802
801
901
902
A
B
C
bwd
fwd
x01

FIG. 9

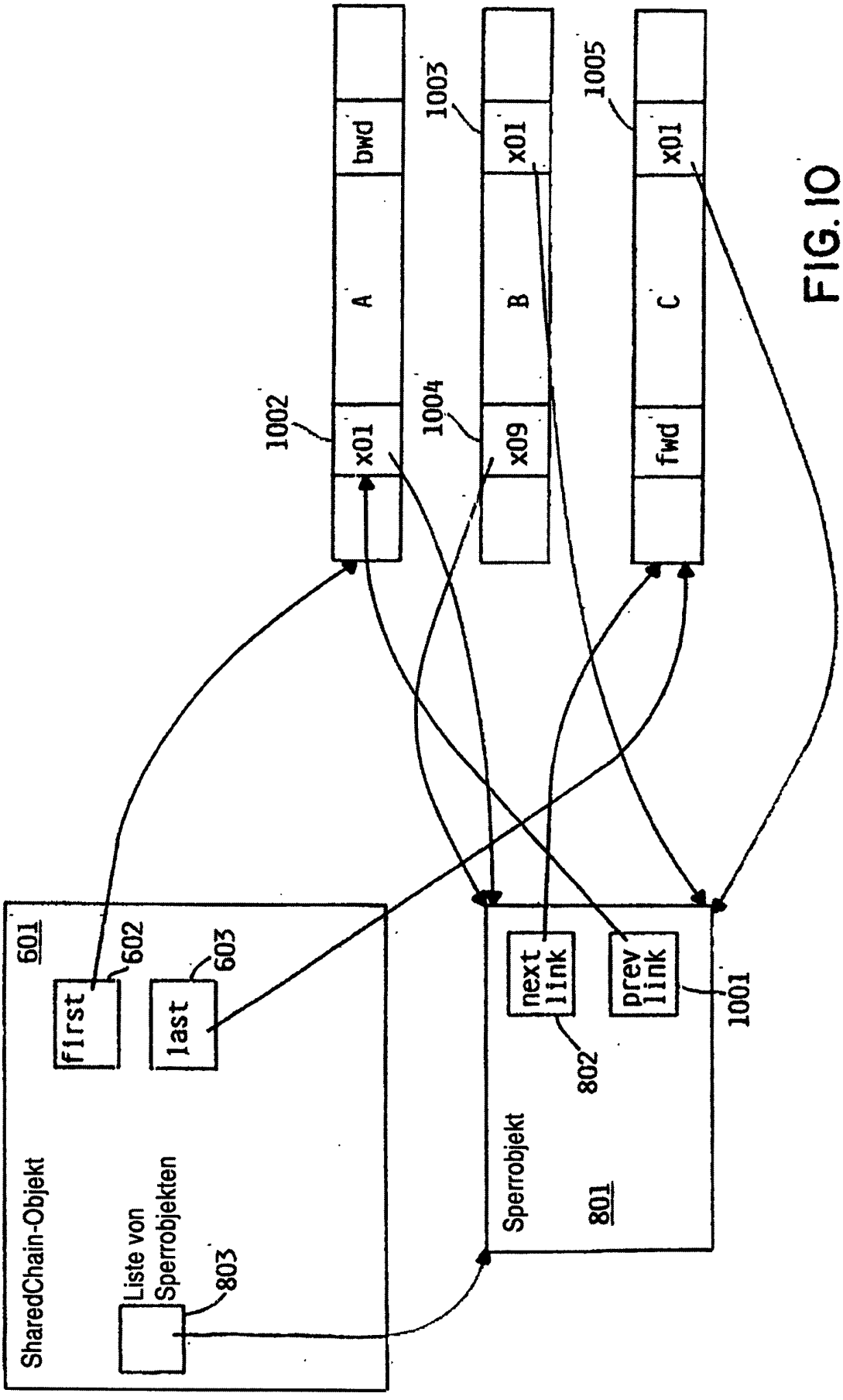
SharedChain-Objekt
601
first
602
last
603
Liste von Sperrobjekten
803
Sperrobjekt
801
next link
802
prev link
1001
1002
x01
A
bwd
1003
x01
B
1004
x09
1005
x01
C
fwd

FIG. 10

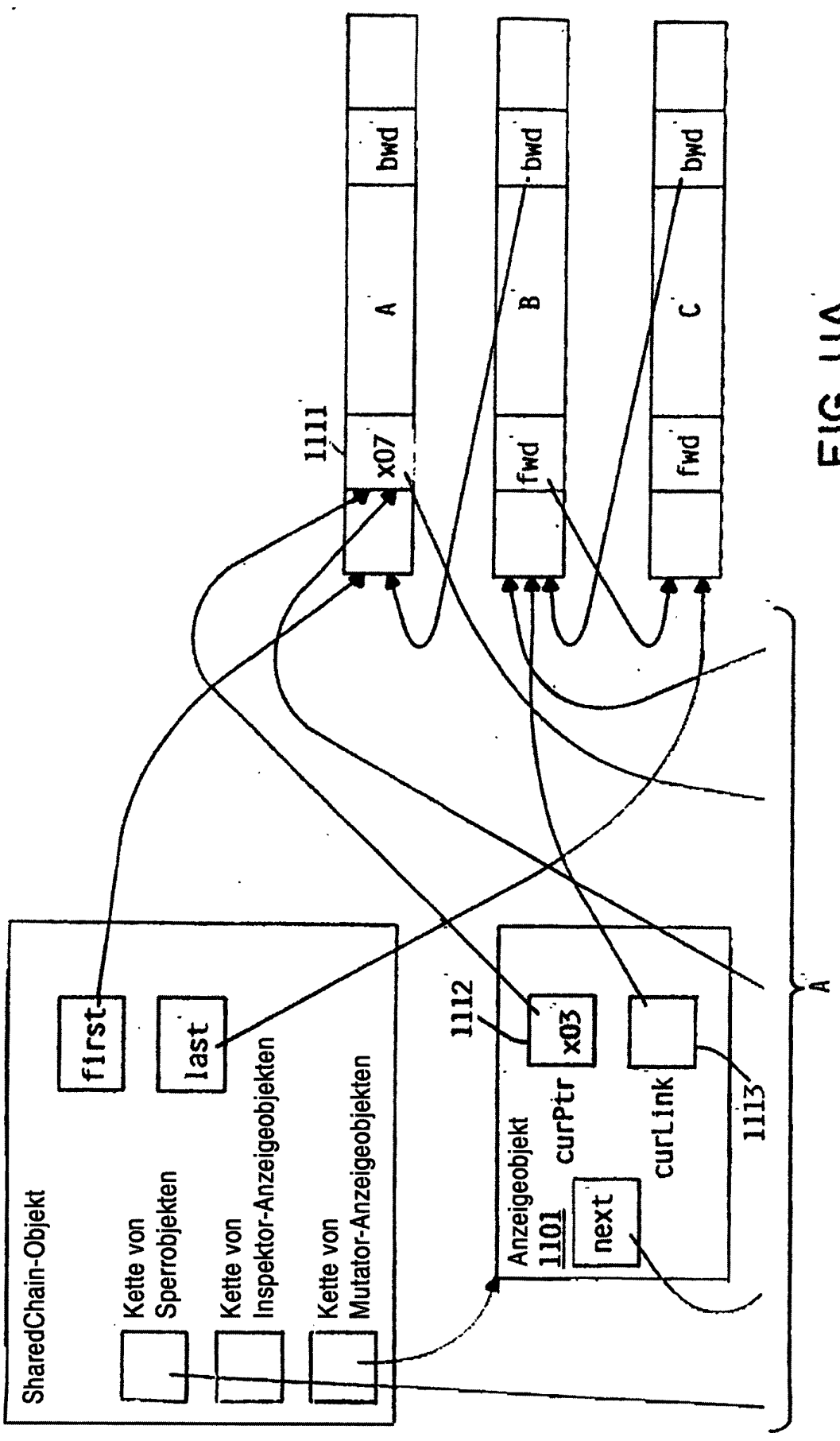
SharedChain-Objekt
first
last
Kette von Sperrobjekten
Kette von Inspektor-Anzeigeobjekten
Kette von Mutator-Anzeigeobjekten
1111
x07
A
bwd
fwd
B
C
1112
x03
curPtr
curLink
1113
Anzeigeobjekt
1101
next
A

FIG. 11A

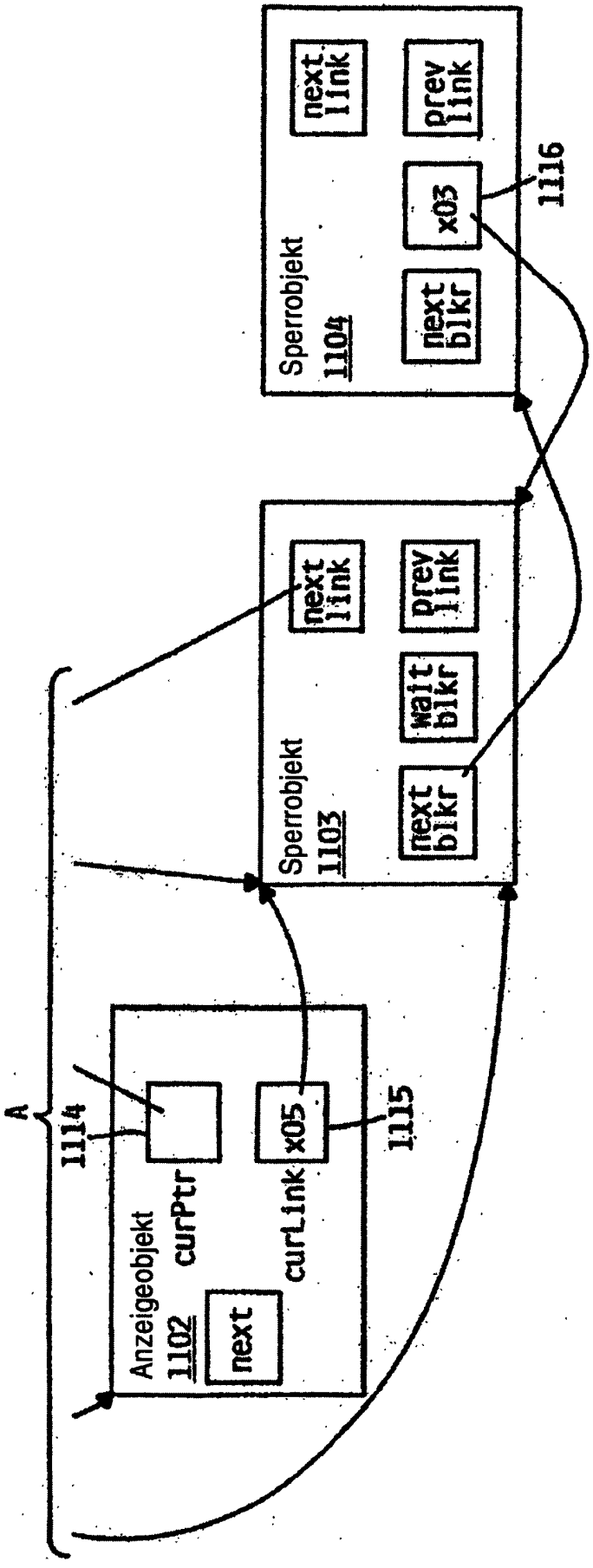
A
1114
Anzeigeobjekt
1102
next
curPtr
curLink
x05
1115
Sperrobjekt
1103
next link
prev link
wait blkr
next blkr
Sperrobjekt
1104
next link
prev link
x03
1116
next blkr

FIG. 11B