(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 July 2001 (26.07.2001)

PCT

(10) International Publication Number
WO 01/53942 A2

(51) International Patent Classification[7]: G06F 9/46

(21) International Application Number: PCT/US01/00042

(22) International Filing Date: 2 January 2001 (02.01.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/177,089 20 January 2000 (20.01.2000) US
09/547,288 11 April 2000 (11.04.2000) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, Palo Alto, CA 94303 (US).

(72) Inventors: SHAVIT, Nir, N.; 153 Upland Road, Cambridge, MA 02140 (US). AGESEN, Ole; 153 Laurel Drive, Needham, MA 02492 (US). DETLEFS, David, L.; 94 Depot Street, Westford, MA 01886 (US). FLOOD, Christine, H.; 13 Main Street, Westford, MA 01886 (US). GARTHWAITE, Alexander, T.; 2 Burton Avenue, Beverly, MA 01915 (US). MARTIN, Paul, A.; 70 Ronald Road, Arlington, MA 02474 (US). STEELE, Guy, L., Jr.; 9 Lantern Lane, Lexington, MA 02421 (US).

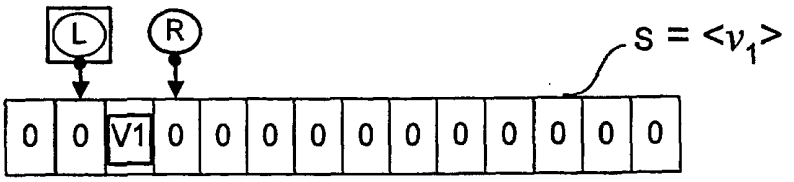
(74) Agent: O'BRIEN, David, W.; Zagorin, O'Brien & Graham, L.L.P., Suite 870, 401 West 15th Street, Austin, TX 78701 (US).

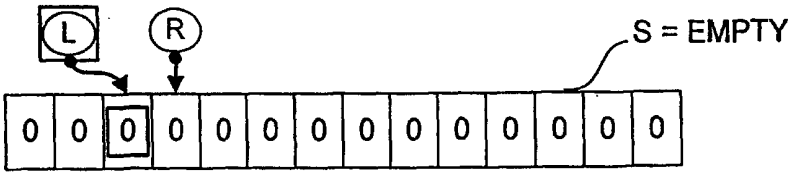
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE,

[Continued on next page]

(54) Title: DOUBLE-ENDED QUEUE IN A CONTIGUOUS ARRAY WITH CONCURRENT NON-BLOCKING INSERT AND REMOVE OPERATIONS





(57) Abstract: An array-based concurrent shared object implementation has been developed that provides non-blocking and linearizable access to the concurrent shared object. In an application of the underlying techniques to a deque, the array-based algorithm allows uninterrupted concurrent access to both ends of the deque, while returning appropriate exceptions in the boundary cases when the deque is empty or full. An interesting characteristic of the concurrent deque implementation is that a processor can detect these boundary cases, e.g., determine whether the array is empty or full, without checking the relative locations of the two end pointers in an atomic operation.

WO 01/53942 A2

IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

DOUBLE-ENDED QUEUE IN A CONTIGUOUS ARRAY WITH CONCURRENT NON-BLOCKING INSERT AND REMOVE OPERATIONS

TECHNICAL FIELD

The present invention relates to coordination amongst processors in a multiprocessor computer, and more particularly, to structures and techniques for facilitating non-blocking access to concurrent shared objects.

Background Art

Non-blocking algorithms can deliver significant performance benefits to parallel systems. However, there is a growing realization that existing synchronization operations on single memory locations, such as compare-and-swap (CAS), are not expressive enough to support design of efficient non-blocking algorithms. As a result, stronger synchronization operations are often desired. One candidate among such operations is a double-word compare-and-swap (DCAS). If DCAS operations become more generally supported in computers systems and, in some implementations, in hardware, a collection of efficient current data structure implementations based on the DCAS operation will be needed.

Massalin and Pu disclose a collection of DCAS-based concurrent algorithms. *See e.g.,* H. Massalin and C. Pu, *A Lock-Free Multiprocessor OS Kernel*, Technical Report TR CUCS–005–9, Columbia University, New York, NY, 1991, pages 1-19. In particular, Massalin and Pu disclose a lock-free operating system kernel based on the DCAS operation offered by the Motorola 68040 processor, implementing structures such as stacks, FIFO-queues, and linked lists. Unfortunately, the disclosed algorithms are centralized in nature. In particular, the DCAS is used to control a memory location common to all operations, and therefore limits overall concurrency.

Greenwald discloses a collection of DCAS-based concurrent data structures that improve on those of Massalin and Pu. *See e.g.,* M. Greenwald. *Non-Blocking Synchronization and System Design*, Ph.D. thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999, 241 pages. In particular, Greenwald discloses implementations of the DCAS operation in software and hardware and discloses two DCAS-based concurrent double-ended queue (deque) algorithms implemented using an array. Unfortunately, Greenwald's algorithms use DCAS in a restrictive way. The first, described in Greenwald, *Non-Blocking Synchronization and System Design*, at pages 196-197, used a two-word DCAS as if it were a three-word operation, storing two deque end pointers in the same memory word, and performing the DCAS operation on the two pointer word and a second word containing a value. Apart from the fact that Greenwald's algorithm limits applicability by cutting the index range to half a memory word, it also prevents concurrent access to the two ends of the deque. Greenwald's second algorithm, described in Greenwald, *Non-Blocking Synchronization and System Design*, at pages 217-220) assumes an array of unbounded size, and does not deal with classical array-based issues such as detection of when the deque is empty or full.

Arora et al. disclose a CAS-based deque with applications in job-stealing algorithms. *See e.g.,* N. S. Arora, Blumofe, and C. G. Plaxton, *Thread Scheduling For Multiprogrammed Multiprocessors*, in

Proceedings of the 10th Annual ACM Sy nposium on Parallel Algorithms and Architectures, 1998. Unfortunately, the disclosed non-blockin g implementation restricts one end of the deque to access by only a single processor and restricts the other en d to o nly pop operations.

Accordingly, improved techniques are desired that do not suffer from the above-described drawbacks of prior approaches.

DISCLOSURE OF INVENTION

A set of structures and techniques are described herein whereby an exemplary concurrent shared object, namely a double-ended queue (deque), is provided. Although a described non-blocking, linearizable deque implementation exemplifies several advantages of realizations in accordance with the present invention, the present invention is not limited thereto. Indeed, based on the description herein and the claims that follow, persons of ordinary skill in the art will appreciate a variety of concurrent shared object implementations. For example, although the described deque implementation exemplifies support for concurrent push and pop operations at both ends thereof, other concurrent shared objects implementations in which concurrency requirements are less severe, such as LIFO or stack structures and FIFO or queue structures, may also be implemented using the techniques described herein.

Accordingly, a novel array-based concurrent shared object implementation has been developed that provides non-blocking and linearizable access to the concurrent shared object. In an application of the underlying techniques to a deque, the array-based algorithm allows uninterrupted concurrent access to both ends of the deque, while returning appropriate exceptions in the boundary cases when the deque is empty or full. An interesting characteristic of the concurrent deque implementation is that a processor can detect these boundary cases, e.g., determine whether the array is empty or full, without checking the relative locations of the two end pointers in an atomic operation.

BRIEF DESCRIPTION OF DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

FIGS. 1A and **1B** illustrate exemplary empty and full states of a double-ended queue (deque) implemented as an array in accordance with the present invention.

FIG. 2 illustrates successful operation of a `pop_right` operation on a partially full state of a deque implemented as an array in accordance with the present invention.

FIG. 3 illustrates successful operation of a `push_right` operation on a empty state of a deque implemented as an array in accordance with the present invention.

FIG. 4 illustrates contention between opposing `pop_left` and `pop_right` operations for a single

remaining element in an almost empty state of a deque implemented as an array in accordance with the present invention.

FIGS. 5A, 5B and **5C** illustrate the results of a sequence of `push_left` and `push_right` operations on a nearly full state of a deque implemented as an array in accordance with the present invention. Following successful completion of the `push_right` operation, the deque is in a full state. **FIGS. 5A, 5B** and **5C** also illustrate an artifact of the linear depiction of a circular buffer, namely that, through a series of preceding operations, ends of the deque may wrap around such that left and right indices may appear (in the linear depiction) to the right and left of each other.

The use of the same reference symbols in different drawings indicates similar or identical items.

MODE(S) FOR CARRYING OUT THE INVENTION

The description that follows presents a set of techniques, objects, functional sequences and data structures associated with concurrent shared object implementations employing double compare-and-swap (DCAS) operations in accordance with an exemplary embodiment of the present invention. An exemplary non-blocking, linearizable concurrent double-ended queue (deque) implementation is illustrative. A deque is a good exemplary concurrent shared object implementation, in that it involves all the intricacies of LIFO-stacks and FIFO-queues, with the added complexity of handling operations originating at both of the deque's ends. Accordingly, techniques, objects, functional sequences and data structures presented in the context of a concurrent deque implementation will be understood by persons of ordinary skill in the art to describe a superset of support and functionality suitable for less challenging concurrent shared object implementations, such as LIFO-stacks, FIFO-queues or concurrent shared objects (including deques) with simplified access semantics.

In view of the above, and without limitation, the description that follows focuses on an exemplary linearizable, non-blocking concurrent deque implementation which behaves as if access operations on the deque are executed in a mutually exclusive manner, despite the absence of a mutual exclusion mechanism. Advantageously, and unlike prior approaches, deque implementations in accordance with some embodiments of the present invention allow concurrent operations on the two ends of the deque to proceed independently.

Computational Model

One realization of the present invention is as a deque implementation, employing the DCAS operation, on a shared memory multiprocessor computer. This realization, as well as others, will be understood in the context of the following computation model, which specifies the concurrent semantics of the deque data structure.

In general, a *concurrent system* consists of a collection of *n processors*. Processors communicate through shared data structures called *objects*. Each object has an associated set of primitive *operations* that provide the mechanism for manipulating that object. Each processor *P* can be viewed in an abstract sense as a

sequential thread of control that applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations where an operation A *precedes* another operation B, if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. A *sequential history* is a history in which each invocation is followed immediately by its corresponding response. The *sequential specification* of an object is the set of *legal* sequential histories associated with it. The basic correctness requirement for a concurrent implementation is *linearizability*. Every concurrent history is "equivalent" to some legal sequential history which is consistent with the real-time order induced by the concurrent history. In a linearizable implementation, an operation appears to take effect atomically at some point between its invocation and response. In the model described herein, a shared memory location L of a multiprocessor computer's memory is a linearizable implementation of an object that provides each processor P_i with the following set of sequentially specified machine operations:

$Read_i\ (L)$ reads location L and returns its value.

$Write_i\ (L,v)$ writes the value v to location L.

$DCAS_i\ (L1, L2, o1, o2, n1, n2)$ is a double compare-and-swap operation with the semantics described below.

Implementations described herein are *non-blocking* (also called *lock-free*). Let us use the term *higher-level operations* in referring to operations of the data type being implemented, and *lower-level operations* in referring to the (machine) operations in terms of which it is implemented. A non-blocking implementation is one in which even though individual higher-level operations may be delayed, the system as a whole continuously makes progress. More formally, a *non-blocking* implementation is one in which any history containing a higher-level operation that has an invocation but no response must also contain infinitely many responses concurrent with that operation. In other words, if some processor performing a higher-level operation continuously takes steps and does not complete, it must be because some operations invoked by other processors are continuously completing their responses. This definition guarantees that the system as a whole makes progress and that individual processors cannot be blocked, only delayed by other processors continuously taking steps. Using locks would violate the above condition, hence the alternate name: *lock-free.*

Double-word Compare-and-Swap Operation

Double-word compare-and-swap (DCAS) operations are well known in the art and have been implemented in hardware, such as in the Motorola 68040 processor, as well as through software emulation. Accordingly, a variety of suitable implementations exist and the descriptive code that follows is meant to facilitate later description of concurrent shared object implementations in accordance with the present invention and not to limit the set of suitable DCAS implementations. For example, order of operations is merely illustrative and any implementation with substantially equivalent semantics is also suitable. Furthermore, although exemplary code that follows includes overloaded variants of the DCAS operation and facilitates efficient implementations of the later described push and pop operations, other implementations,

including single variant implementations may also be suitable.

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1==old1) && (*addr2==old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else {
      return false;
    }
  }
}

boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val *new1, val *new2) {
  atomically {
    temp1 = *addr1;
    temp2 = *addr2;
    if ((temp1 == old1) && (temp2 == old2)) {
      *addr1 = *new1;
      *addr2 = *new2;
      *new1 = temp1;
      *new2 = temp2;
      return true;
    } else {
      *new1 = temp1;
      *new2 = temp2;
      return false;
    }
  }
}
```

Note that in the exemplary code, the DCAS operation is overloaded, i.e., if the last two arguments of the DCAS operation (new1 and new2) are pointers, then the second execution sequence (above) is operative and the original contents of the tested locations are stored into the locations identified by the pointers. In this way, certain invocations of the DCAS operation may return more information than a success/failure flag.

The above sequences of operations implementing the DCAS operation are executed atomically using support suitable to the particular realization. For example, in various realizations, through hardware support (e.g., as implemented by the Motorola 68040 microprocessor or as described in M. Herlihy and J. Moss, *Transactional memory: Architectural Support For Lock-Free Data Structures*, Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992, 12 pages), through non-blocking software emulation (such as described in G. Barnes, *A Method For Implementing Lock-Free Shared Data Structures,* in *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993 or in N. Shavit and D. Touitou, *Software transactional memory*, *Distributed Computing*, 10(2):99–116, February 1997), or via a blocking software emulation.

Although the above-referenced implementations are presently preferred, other DCAS

implementations that substantially preserve the semantics of the descriptive code (above) are also suitable. Furthermore, although much of the description herein is focused on double-word compare-and-swap (DCAS) operations, it will be understood that N-location compare-and-swap operations ($N \geq 2$) may be more generally employed, though often at some increased overhead.

A Double-ended Queue (Deque)

A *deque* object S is a concurrent shared object, that in an exemplary realization is created by an operation of a constructor operation, e.g., `make_deque(length_S)`, and which allows each processor P_i, $0 \leq i \leq n-1$, of a concurrent system to perform the following types of operations on S: `push_right`$_i(v)$, `push_left`$_i(v)$, `pop_right`$_i()$, and `pop_left`$_i()$. Each push operation has an input, v, where v is selected from a range of values. Each pop operation returns an output from the range of values. Push operations on a full deque object and pop operations on an empty deque object return appropriate indications.

A concurrent implementation of a deque object is one that is linearizable to a standard sequential deque. This sequential deque can be specified using a state-machine representation that captures all of its allowable sequential histories. These sequential histories include all sequences of push and pop operations induced by the state machine representation, but do not include the actual states of the machine. In the following description, we abuse notation slightly for the sake of clarity.

The state of a deque is a sequence of items $S = \langle v_0, \ldots, v_k \rangle$ from the range of values, having cardinality $0 \leq |S| \leq$ `length_S`. The deque is initially in the empty state (following invocation of `make_deque(length_S)`), that is, has cardinality 0, and is said to have reached a full state if its cardinality is `length_S`.

The four possible push and pop operations, executed sequentially, induce the following state transitions of the sequence $S = \langle v_0, \ldots, v_k \rangle$, with appropriate returned values:

`push_right`(v_{new}) if S is not full, sets S to be the sequence $S = \langle v_0, \ldots, v_k, v_{new} \rangle$

`push_left`(v_{new}) if S is not full, sets S to be the sequence $S = \langle v_{new}, v_0, \ldots, v_k \rangle$

`pop_right()` if S is not empty, sets S to be the sequence $S = \langle v_0, \ldots, v_{k-1} \rangle$

`pop_left()` if S is not empty, sets S to be the sequence $S = \langle v_1, \ldots, v_k \rangle$

For example, starting with an empty deque state, $S = \langle\rangle$, the following sequence of operations and corresponding transitions can occur. A `push_right(1)` changes the deque state to $S = \langle 1 \rangle$. A `push_left(2)` subsequently changes the deque state to $S = \langle 2,1 \rangle$. A subsequent `push_right(3)` changes the deque state to $S = \langle 2,1,3 \rangle$. Finally, a subsequent `pop_right()` changes the deque state to $S = \langle 2,1 \rangle$.

An Array-Based Implementation

The description that follows presents an exemplary non-blocking implementation of a deque based on an underlying contiguous array data structure wherein access operations (illustratively, `push_left`, `pop_left`, `push_right` and `pop_right`) employ DCAS operations to facilitate concurrent access. Exemplary code and illustrative drawings will provide persons of ordinary skill in the art with detailed understanding of one particular realization of the present invention; however, as will be apparent from the description herein and the breadth of the claims that follow, the invention is not limited thereto. Exemplary right-hand-side code is described in substantial detail with the understanding that left-hand-side operations are symmetric. Use herein of directional signals (e.g., left and right) will be understood by persons of ordinary skill in the art to be somewhat arbitrary. Accordingly, many other notational conventions, such as top and bottom, first-end and second-end, etc., and implementations denominated therein are also suitable.

With the foregoing in mind, an exemplary non-blocking implementation of a deque based on an underlying contiguous array data structure is illustrated with reference to **FIGS. 1A** and **1B**. In general, an array-based deque implementation includes a contiguous array `S[0..length_S-1]` of storage locations indexed by two counters, `R` and `L`. The array, as well as the counters (or alternatively, pointers or indices), are typically stored in memory. Typically, the array `S` and indices `R` and `L` are stored in a same memory, although more generally, all that is required is that a particular DCAS implementation span the particular storage locations of the array and an index.

In operations on `S`, we assume that `mod` is the modulus operation over the integers (e.g., `-1 mod 6 = 5, -2 mod 6 = 4`, and so on). Henceforth, in the description that follows, we assume that all values of `R` and `L` are modulo `length_S`, which implies that the array `S` is viewed as being circular. The array `S[0..length_S-1]` can be viewed as if it were laid out with indexes increasing from left to right. We assume a distinguishing value, e.g., "null" (denoted as 0 in the drawings), not occurring in the range of real data values for `S`. Of course, other distinguishing values are also suitable.

Operations on `S` proceed as follows. Initially, for empty deque state, `L` points immediately to the left of `R`. In the illustrative embodiment, indices `L` and `R` always point to the next location into which a value can be inserted. If there is a null value stored in the element of `S` immediately to the right of that identified by `L` (or respectively, in the element of `S` immediately to the left of that identified by `R`), then the deque is in the empty state. Similarly, if there is a non-null value in the element of `S` identified by `L` (respectively, in the element of `S` identified by `R`), then the deque is in the full state. **FIG. 1A** depicts an empty state and **FIG. 1B** depicts a full state. During the execution of access operations in accordance with the present invention, the use of a DCAS guarantees that on any location in the array, at most one processor can succeed in modifying the entry at that location from a "null" to a "non-null" value or vice versa.

An illustrative pop_right access operation in accordance with the present invention follows:

```
val pop_right {
  while (true) {
    oldR = R;
5   newR = (oldR - 1) mod length_S;
    oldS = S[newR];
    if (oldS == "null") {
      if (oldR == R)
        if (DCAS(&R, &S[newR],
10                oldR, oldS, oldR, oldS))
          return "empty";
    }
    else {
      newS = "null";
15    if (DCAS(&R, &S[newR],
              oldR, oldS, &newR, &newS))
        return newS;
      else if (newR == oldR) {
        if (newS == "null") return "empty";
20    }
    }
  }
}
```

To perform a pop_right, a processor first reads R and the location in S corresponding to R-1 (Lines 3-5, above). It then checks whether S[R-1] is null. As noted above, S[R-1] is shorthand for S[R-1 mod length_S]. If S[R-1] is null, then the processor reads R again to see if it has changed (Lines 6-7). This additional read is a performance enhancement added under the assumption that the common case is that a null value is read because another processor "stole" the item, and not because the queue is really empty. Other implementations need not employ such an enhancement. The test can be stated as follows: if R hasn't changed and S[R-1] is null, then the deque must be empty since the location to the left of R always contains a value unless there are no items in the deque. However, the conclusion that the deque is empty can only be made based on an instantaneous view of R and S[R-1]. Therefore, the pop_right implementation employs a DCAS (Lines 8-10) to check if this is in fact the case. If so, pop_right returns an indication that the deque is empty. If not, then either the value in S[R-1] is no longer null or the index R has changed. In either case, the processor loops around and starts again, since there might now be an item to pop.

If S[R-1] is not null, the processor attempts to pop that item (Lines 12-20). The pop_right implementation employs a DCAS to try to atomically decrement the counter R and place a null value in S[R-1], while returning (via &newR and &newS) the old value in S[R-1] and the old value of the counter R (Lines 13-15). Note that the overloaded variant of DCAS described above is utilized here.

A successful DCAS (and hence a successful pop_right operation) is depicted in **FIG. 2**. Initially, $S = \langle v_1, v_2, v_3, v_4 \rangle$ and L and R are as shown. Contents of R and of S[R-1] are read, but the results of the reads may not be consistent if an intervening competing access has successfully completed. In the context of the deque state illustrated in **FIG. 2**, the competing accesses of concern are a pop_right or a

push_right, although in the case of an almost empty state of the deque, a pop_left might also intervene. Because of the risk of a successfully completed competing access, the pop_right implementation employs a DCAS (lines 14-15) to check the instantaneous values of R and of S[R-1] and, if unchanged, perform the atomic update of R and of S[R-1] resulting in a deque state of $S = \langle v_1, v_2, v_3 \rangle$.

If the DCAS is successful (as indicated in **FIG. 2**), the pop_right returns the value v_4 from S[R-1]. If it fails, pop_right checks the reason for the failure. If the reason for the DCAS failure was that R changed, then the processor retries (by repeating the loop) since there may be items still left in the deque. If R has not changed (Line 17), then the DCAS must have failed because S[R-1] changed. If it changed to null (Line 18), then the deque is empty. An empty deque may be the result of a competing pop_left that "steals" the last item from the pop_right, as illustrated in **FIG. 4**.

If, on the other hand, S[R-1] was not null, the DCAS failure indicates that the value of S[R-1] has changed, and some other processor(s) must have completed a pop and a push between the read and the DCAS operation. In this case, pop_right loops back and retries, since there may still be items in the deque. Note that Lines 17-18 are an optimization, and one can instead loop back if the DCAS fails. The optimization allows detection of a possible empty state without going through the loop, which in case the queue was indeed empty, would require another DCAS operation (Lines 6-10).

To perform a push_right, a sequence similar to pop_right is performed. An illustrative push_right access operation in accordance with the present invention follows:

```
val push_right(val v) {
  while (true) {
    oldR = R;
    newR = (oldR + 1) mod length_S;
    oldS = S[oldR];
    if (oldS != "null") {
      if (oldR == R)
        if (DCAS(&R, &S[oldR],
                 oldR, oldS, oldR, oldS))
          return "full";
    }
    else {
      newS = v;
      if DCAS(&R, &S[oldR],
              oldR, oldS, &newR, &newS)
        return "okay";
      else if (newR == oldR)
        return "full";
    }
  }
}
```

Operation of pop_right is similar to that of push_right, but with all tests to see if a location is null replaced with tests to see if it is non-null, and with S locations corresponding to an index identified by, rather than adjacent to that identified by, the index. To perform a push_right, a processor first reads R and the location in S corresponding to R (Lines 3-5, above). It then checks whether S[R] is non-null. If S[R] is

non-null, then the processor reads R again to see if it has changed (Lines 6-7). This additional read is a performance enhancement added under the assumption that the common case is that a non-null value is read because another processor "beat" the processor, and not because the queue is really full. Other implementations need not employ such an enhancement. The test can be stated as follows: if R hasn't changed and S[R] is non-null, then the deque must be full since the location identified by R always contains a null value unless the deque is full. However, the conclusion that the deque is full can only be made based on an instantaneous view of R and S[R]. Therefore, the push_right implementation employs a DCAS (Lines 8-10) to check if this is in fact the case. If so, push_right returns an indication that the deque is full. If not, then either the value in S[R] is no longer non-null or the index R has changed. In either case, the processor loops around and starts again.

If S[R] is null, the processor attempts to push value, v, onto S (Lines 12-19). The push_right implementation employs a DCAS to try to atomically increment the counter R and place the value, v, in S[R], while returning (via &newR) the old value of index R (Lines 14-16). Note that the overloaded variant of DCAS described above is utilized here.

A successful DCAS and hence a successful push_right operation into an empty deque is depicted in **FIG. 3**. Initially, $S = \langle\rangle$ and L and R are as shown. Contents of R and of S[R] are read, but the results of the reads may not be consistent if an intervening competing access has successfully completed. In the context of the empty deque state illustrated in **FIG. 3**, the competing access of concern is another push_right, although in the case of non-empty state of the deque, a pop_right might also intervene. Because of the risk of a successfully completed competing access, the push_right implementation employs a DCAS (lines 14-15) to check the instantaneous values of R and of S[R] and, if unchanged, perform the atomic update of R and of S[R] resulting in a deque state of $S = \langle v_1 \rangle$. A successful push_right operation into an almost-full deque is illustrated in the transition from deque states of **FIGS. 5B** and **5C**.

In the final stage of the push_right code, in case the DCAS failed, there is a check using the value returned (via &newR) to see if the R index has changed. If it has not, then the failure must be due to a non-null value in the corresponding element of S, which means that the deque is full.

Pop_left and push_left sequences correspond to their above described right hand variants. An illustrative pop_left access operation in accordance with the present invention follows:

```
val pop_left {
  while (true) {
    oldL = L;
    newL = (oldL + 1) mod length_S;
    oldS = S[newL];
    if (oldS == "null") {
      if (oldL == L)
        if (DCAS(&L, &S[newL],
                 oldL, oldS, oldL, oldS))
          return "empty";
    }
```

```
      else {
        newS = "null";
        if (DCAS(&L, &S[newL],
                 oldL, oldS, &newL, &newS))
          return newS;
        else if (newL == oldL) {
          if (newS == "null") return "empty";
        }
      }
    }
  }
```

An illustrative `push_left` access operation in accordance with the present invention follows:

```
val push_left(val v) {
  while (true) {
    oldL = L;
    newL = (oldL - 1) mod length_S;
    oldS = S[oldL];
    if (oldS != "null") {
      if (oldL == L)
        if (DCAS(&L, &S[oldL],
                 oldL, oldS, oldL, oldS))
          return "full" ;
    }
    else {
      newS = v;
      if (DCAS(&L, &S[oldL],
               oldL, oldS, &newL, &newS))
        return "okay";
      else if (newL == oldL)
        return "full";
    }
  }
}
```

FIGS. 5A, 5B and **5C** illustrate operations on a nearly full deque including a `push_left` operation (**FIG. 5B**) and a `push_right` operation that result in a full state of the deque (**FIG. 5C**). Notice that `L` has wrapped around and is "to-the-right" of `R`, until the deque becomes full, in which case again `L` and `R` cross. This switching of the relative location of the `L` and `R` pointers is somewhat confusing and represents a limitation of the linear presentation in the drawings. However, in any case, it should be noted that each of the above described access operations (`push_left`, `pop_left`, `push_right` and `pop_right`) can determine the state of the deque, without regard to the relative locations of `L` and `R`, but rather by examining the relation of a given index (`R` or `L`) to the value in a corresponding element of `S`.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Many variations, modifications, additions, and improvements are possible. Plural instances may be provided for components described herein as a single instance. Finally, boundaries between various components, services, servlets, and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of claims that follow. Structures and functionality presented as discrete components in the exemplary

configurations may be implemented as a combined structure or component. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined in the claims that follow.

WHAT IS CLAIMED:

1. A method of managing access to an array susceptible to concurrent operations on a sequence encoded therein, the method comprising:

executing as part of a pop operation, a double compare and swap (DCAS) to atomically update a then-current, end identifying index for the array and a element of the array adjacent to that identified by the end identifying index; and

returning from the DCAS, on failure thereof, an indication by which an empty state of the array is detectable.

2. The method of claim 1,

wherein the indication by which the empty state of the array is detectable is indicative of presence of a distinguishing value in the adjacent element.

3. The method of claim 1, wherein the array encodes a double-ended queue as a circular buffer of bounded size, the end identifying index and an opposing end identifying index delimiting the sequence.

4. A method according to claim 1, 2 or 3,

wherein the pop operation is a left pop operation;

wherein the end identifying index is a left-end index; and

wherein the adjacent element is to the right of the identified element.

5. A method according to claim 1, 2 or 3,

wherein the pop operation is a right pop operation;

wherein the end identifying index is a right-end index; and

wherein the adjacent element is to the left of the identified element.

6. A method of managing access to an array susceptible to concurrent operations on a sequence encoded therein, the method comprising:

executing as part of a push operation, a double compare and swap (DCAS) to atomically update a then-current, end identifying index for the array and an element of the array identified by the end identifying index; and

returning from the DCAS, on failure thereof, an indication by which a full state of the array is detectable.

7. The method of claim 6,

wherein the indication by which the full state of the array is detectable is indicative of absence of a distinguishing value in the identified element.

8. A method according to claim 6 or 7,
wherein the push operation is a left push operation; and
wherein the end identifying index is a left-end index.

9. A method according to claim 6 or 7,
wherein the push operation is a right push operation; and
wherein the end identifying index is a right-end index.

10. A method of providing concurrent access to a double-ended data structure of bounded size implemented using a circular buffer technique, the method comprising:
as part of an access to a first-end of the double-ended data structure, performing in alternate legs of a conditional branch:
a first multi-way compare and swap on then-current contents of a first-end index store and a corresponding element of the double-ended data structure to disambiguate a retry state and a boundary condition state of the double-ended data structure;
a second multi-way compare and swap on then-current contents of the first-end index store and a corresponding element of the double-ended data structure, the second multi-way compare and swap performing the access and, on failure thereof, returning an indication disambiguating a retry state and the boundary condition state of the double-ended data structure,
wherein the conditional branch discriminates between presence and absence of a distinguishing value in an element of the double-ended data structure corresponding to the then-current contents of the first-end index store.

11. The method of claim 10,
wherein the access includes a pop from the first-end of the double-ended data structure;
wherein the boundary condition state is an empty state of the double-ended data structure; and
wherein the retry state results from a concurrently performed push or pop access at the first-end of the double-ended data structure.

12. The method of claim 10,
wherein the access includes a push onto the first-end of the double-ended data structure;
wherein the boundary condition state is a full state of the double-ended data structure; and
wherein the retry state results from a concurrently performed push or pop access at the first-end of the double-ended data structure.

13. A method according to claim 10, 11 or 12, wherein the double-ended data structure includes a double-ended queue (deque).

14. A method according to claim 10, 11 or 12, wherein the multi-way compare and swap is a double compare and swap (DCAS).

15. A method of managing concurrent access to a double-ended queue (deque), the method comprising:

employing, in an implementation of a pop operation, execution of a double compare and swap (DCAS) to interrogate instantaneous values of a first end index and a deque element adjacent to that identified thereby for a signature indicative of an empty state of the array, the signature including presence in that adjacent element of a distinguishing value,

wherein successful execution of an opposing end pop operation includes execution of a DCAS to atomically update a second end index and a deque element adjacent to that identified thereby, the update of that adjacent element storing the distinguishing value therein.

16. The method of claim 15, further comprising:

wherein successful execution of a competing, same end pop operation includes execution of a DCAS to atomically update the first end index and a deque element adjacent to that identified thereby, the update of that adjacent element storing the distinguishing value therein.

17. The method of claim 15, further comprising:

wherein the first end index is a left index and, if the state of the deque is non-empty, the deque element adjacent to that identified thereby is a left most element of the deque;

wherein the second end index is a right index and, if the state of the deque is non-empty, the deque element adjacent to that identified thereby is the right most element of the deque.

18. The method of claim 15,

wherein the pop operation is a left pop operation and the opposing end pop operation is a right pop operation; and

wherein the first end index is a left end index and the element adjacent to that identified thereby is adjacent to the right.

19. A method according to any of claims 15 to 18, wherein the distinguishing value is encoded as a null value.

20. A method according to any of claims 15 to 19,

employing, in an implementation of a push operation, execution of a double compare and swap (DCAS) to interrogate instantaneous values of a third end index and a deque element identified thereby for a signature indicative of an full state of the deque, the signature including absence in that identified deque element of a distinguishing value,

wherein successful execution of an opposing end push operation includes execution of a DCAS to atomically update a fourth end index and a deque element identified thereby, the update of the identified deque element storing a value other than the distinguishing value therein.

21. The method of claim 20,

wherein the first end index and the third end index identify a same end of the deque; and

wherein the second end index and the fourth end index identify a same end of the deque.

22. The method of claim 20,

wherein the first end index and the fourth end index identify a same end of the deque; and

wherein the second end index and the third end index identify a same end of the deque.

23. A method of managing concurrent access to a double-ended queue (deque), the method comprising:

employing, in an implementation of a push operation, execution of a double compare and swap (DCAS) to interrogate instantaneous values of a first end index and a deque element identified thereby for a signature indicative of a full state of the deque, the signature including absence in that identified deque element of a distinguishing value,

wherein successful execution of an opposing end push operation includes execution of a DCAS to atomically update an opposing end index and a deque element identified thereby, the update of the identified deque element storing a value other than the distinguishing value therein.

24. The method of claim 23,

wherein successful execution of a competing, same end push operation includes execution of a DCAS to atomically update the first end index and a deque element identified thereby, the update of that adjacent element storing a value other than the distinguishing value therein.

25. A method of managing concurrent access to an array susceptible to competing accesses at same and opposing ends thereof, the method comprising:

executing as part of a first access operation, a double compare and swap (DCAS) to atomically update a first end identifying index and an element of the array corresponding to a then-current value thereof;

executing as part of a competing second access operation, a DCAS to atomically update a second end identifying index and an element of the array corresponding to a then-current value thereof,

wherein, if successful completion of one of the first and the second competing access operations results in a boundary condition state of the array, the DCAS of the other of the first and the second access operations fails and returns an indication thereof.

26. The method of claim 25,
wherein the first access operation and the competing second access operation are competing pop operations;
wherein the array elements corresponding to the first and second indices are each adjacent to that identified by the respective index;
wherein the boundary condition state is an empty state; and
wherein the adjacent element referenced by the failing one of the competing pop operations encodes a distinguishing value signifying the empty state.

27. The method of claim 26,
wherein the competing pop operations are competing opposing end pop operations; and
wherein the first index and the second index identify opposing ends of the array;

28. The method of claim 26,
wherein the competing pop operations are competing same end pop operations; and
wherein the first index and the second index identify a same end of the array;

29. The method of claim 25,
wherein the first access operation and the competing second access operation are competing push operations;
wherein the array elements corresponding to the first and second indices are each identified by the respective index;
wherein the boundary condition state is an full state; and
wherein the array element referenced by the failing one of the competing push operations encodes a value other than a distinguishing value.

30. The method of claim 29,
wherein the competing push operations are competing opposing end push operations; and
wherein the first index and the second index identify opposing ends of the array;

31. The method of claim 29,
wherein the competing push operations are competing same end push operations; and
wherein the first index and the second index identify a same end of the array;

32. A double-ended queue (deque) implementation comprising:
a contiguous array S of bounded size encoded in an addressable store;

a left index L and a right index R into the contiguous array, the contiguous array S, the left index L and the right index R together defining a circular buffer with state including a sequence of zero or more values encoded in the contiguous array between elements S[L] and S[R] thereof; and

a computer readable encoding of at least a first access operation, execution of the first access operation operating at a particular end of the sequence and employing a double compare and swap (DCAS) to atomically update a corresponding one, but not both, of the left and right indices L and R and an element of the contiguous array adjacent to the contiguous array element identified thereby.

33. The double-ended queue (deque) implementation of claim 32,

wherein the first access operation includes a push; and

wherein, on failure, the DCAS returns an indication by which a full state of the contiguous array is detected.

34. The double-ended queue (deque) implementation of claim 32,

wherein the first access operation includes a pop; and

wherein, on failure, the DCAS returns an indication by which an empty state of the contiguous array is detected.

35. A double-ended queue (deque) implementation according to claim 32, 33 or 34, further comprising:

computer readable encodings of at least three additional access operations,

wherein the first and the three additional access operations together include push and pop operations at left and rights end of the sequence, respectively.

36. A concurrent shared object implementation comprising:

a contiguous array encoded in an addressable store;

opposing indices into the contiguous array usable to delimit therebetween a portion of the contiguous array for storage of a sequence of zero or more data values; and

a computer readable encoding of push and pop operations defined to operate on elements of the contiguous array and on respective of the opposing indices,

wherein the push operation employs a first instance of a double compare and swap (DCAS) operation to atomically update one of the opposing indices and a corresponding element of the contiguous array while returning on failure, an indication by which a full state of the contiguous array is detected, and

wherein the pop operation employs a second instance of a DCAS operation to atomically update one of the opposing indices and a corresponding element of the contiguous array while returning on failure, an indication by which an empty state of the contiguous array is detected.

37. The concurrent shared object implementation of claim 36,
wherein concurrent shared object includes a deque; and
wherein the computer readable encoding of push and pop operations includes:
opposing end variants of the push operation; and
opposing end variants of the push operation.

38. The concurrent shared object implementation of claim 36,
wherein concurrent shared object includes a queue or FIFO; and
wherein the computer readable encoding of push and pop operations operate on opposing ends of the queue or FIFO.

39. The concurrent shared object implementation of claim 36,
wherein concurrent shared object includes a stack or LIFO; and
wherein the computer readable encoding of push and pop operations operate on a same end of the stack or LIFO.

40. A computer program product encoded in at least one computer readable medium, the computer program product comprising:
at least one functional sequence implementing an access operation on a concurrent shared object, the concurrent shared object instantiable circular buffer of bounded size implementing a contiguous array delimited by a pair of end identifying indices;
instances of the at least one functional sequence concurrently executable by plural processors of a multiprocessor and each including a double compare and swap (DCAS) to atomically update a corresponding one of the end identifying indices and an element of the array corresponding to a then-current value thereof; and
the DCAS of the at least one functional sequence responsive to a corresponding boundary condition state of the concurrent shared object.

41. A computer program product as recited in 40,
wherein the at least one functional sequence includes opposing end variants of push and pop operations on the concurrent shared object;
wherein the boundary condition state corresponding to push operations is a full state of the array; and
wherein the boundary condition state corresponding to pop operations is an empty state of the array.

42. A computer program product as recited in 40,
wherein the at least one computer readable medium is selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.

43. An apparatus comprising:

plural processors;

a store addressable by each of the plural processors;

first- and second-end index stores accessible to each of the plural processors for identifying opposing ends of a bounded-size contiguous array encoded in circular buffer form in the addressable store; and

means for coordinating competing access operations, the coordinating means employing in each instance thereof, at least one double compare and swap (DCAS) operation to disambiguate a retry state and a boundary condition state of the array based on then-current contents of one, but not both, of first- and second-end index stores and an array element corresponding thereto.

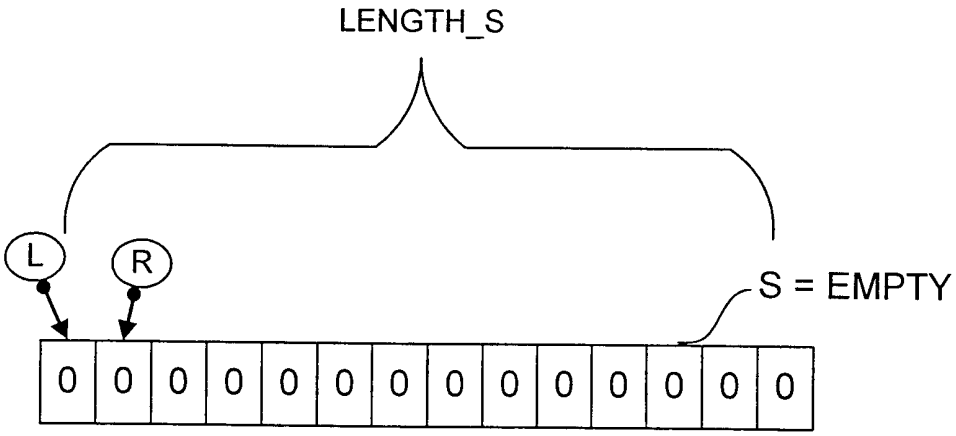

LENGTH_S
L
R
S = EMPTY
0 0 0 0 0 0 0 0 0 0 0 0 0 0

FIG. 1A

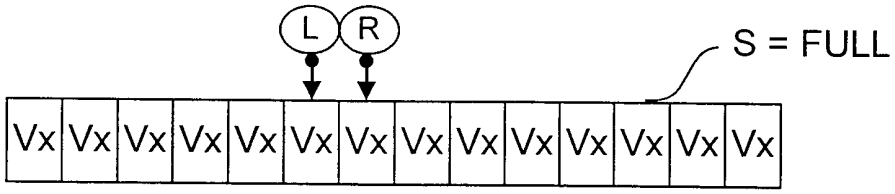

L
R
S = FULL
Vx Vx Vx Vx Vx Vx Vx Vx Vx Vx Vx Vx Vx Vx

FIG. 1B

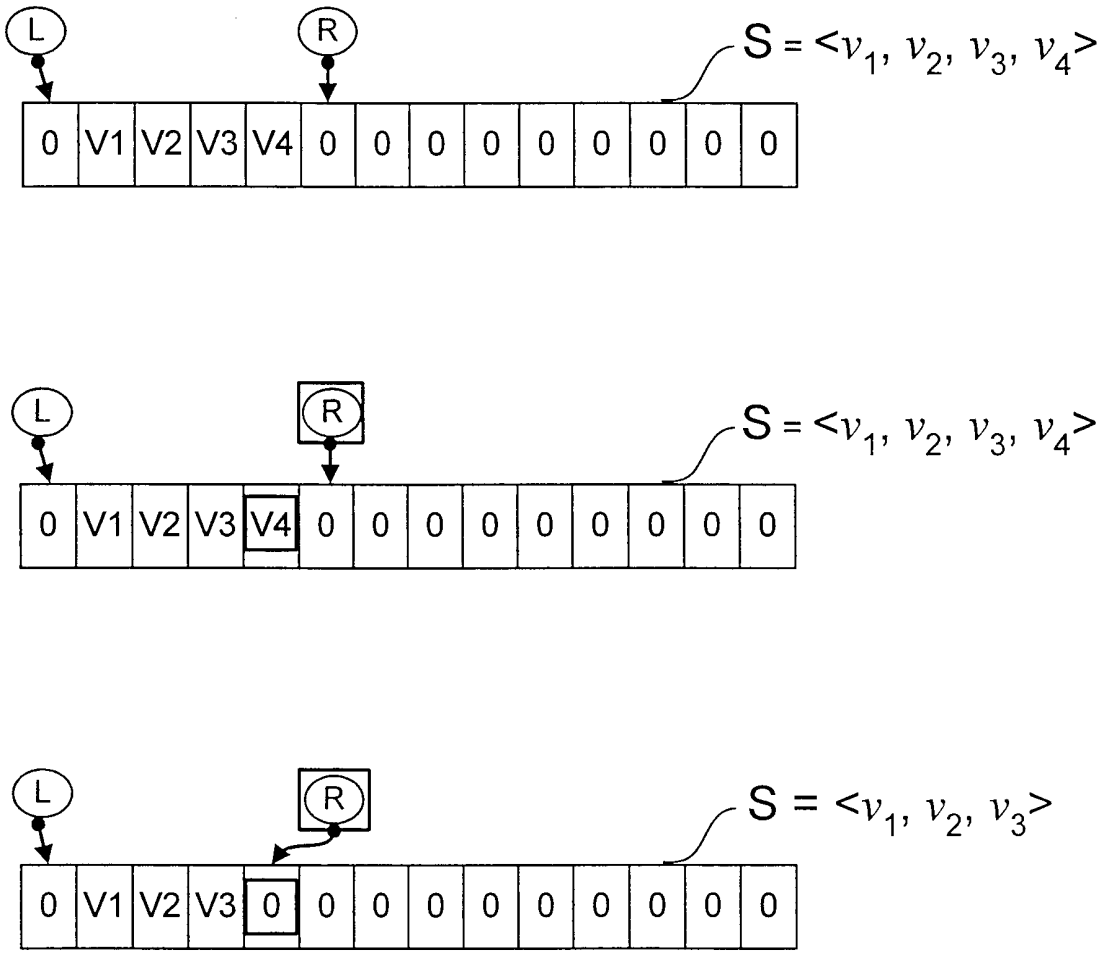

L
R
S = <v_1, v_2, v_3, v_4>
0 V1 V2 V3 V4 0 0 0 0 0 0 0 0 0
L
R
S = <v_1, v_2, v_3, v_4>
0 V1 V2 V3 V4 0 0 0 0 0 0 0 0 0
L
R
S = <v_1, v_2, v_3>
0 V1 V2 V3 0 0 0 0 0 0 0 0 0 0

FIG. 2

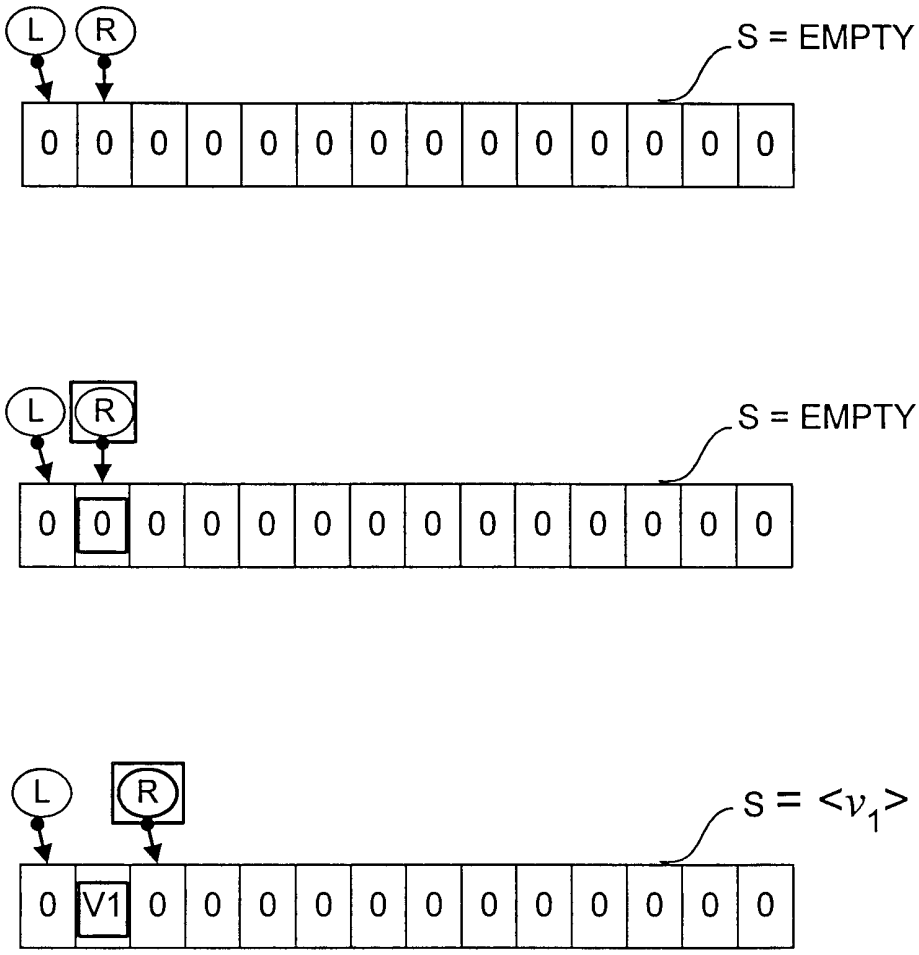

L
R
S = EMPTY
0 0 0 0 0 0 0 0 0 0 0 0 0 0
L
R
S = EMPTY
0 0 0 0 0 0 0 0 0 0 0 0 0 0
L
R
S = <v1>
0 V1 0 0 0 0 0 0 0 0 0 0 0 0

FIG. 3

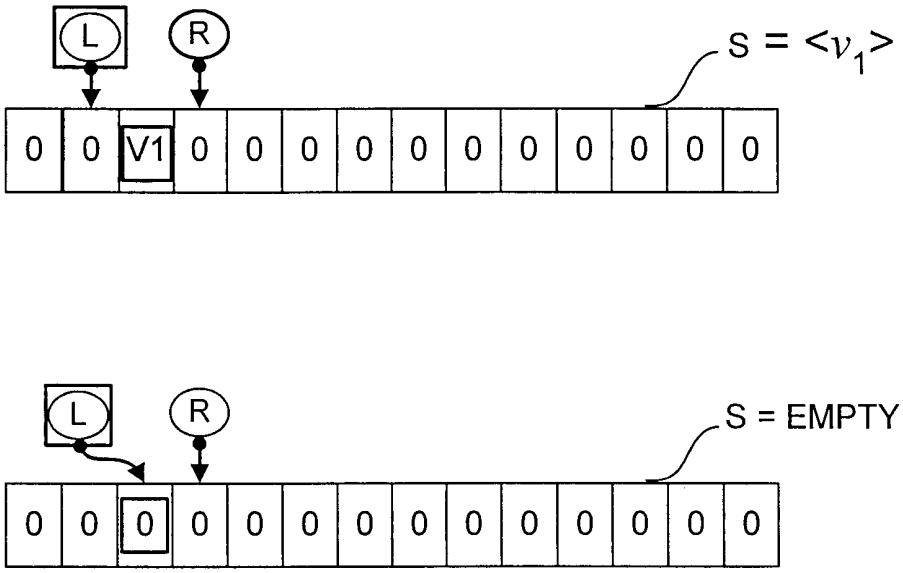


FIG. 4

FIG. 5A

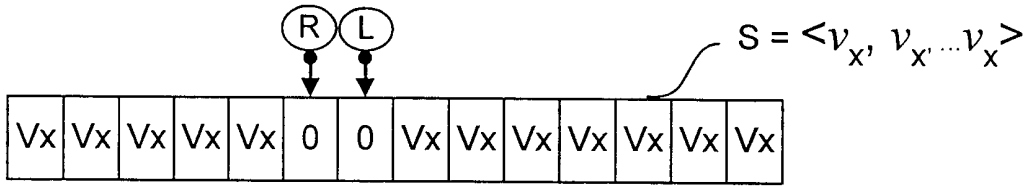


FIG. 5B

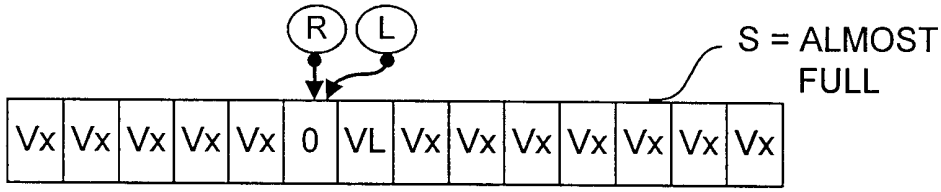


FIG. 5C