Jurnal Mahajana Inforamasi, Vol.2 No 1, 2017
e-ISSN: 2527-8290

Concurrent Access Algorithms for Different Data Structures: A Research Review

Parminder Kaur
Program Study of Information System University Sari Mutiara, Indonesia
Parm.jass89@gmail.com

Abstract

Algorithms refers to a finite set of steps, which when followed solves a number of problems and algorithams for concurrent data structure have gained attention in recent years as multi-core processors have become ubiquitous. Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts. The primary source of this additional difficulty is concurrency. This paper provides an overview of the some concurrent access algorithms for different data structures.

Keywords: concurrency, lock-free, non-blocking, mem-ory management, compares and swap, elimination.

1. Introduction

The logical and mathematical model used to organize the data in the main memory is called **Data Structure**.A concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads or processes on a computer. Designing concurrent data structures and ensuring their correctness is a difficult task, significantly more challenging than doing so for their sequential counterparts. The difficult of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on To manage the difficulty of concurrent programming, multithreaded applications need synchronization to ensure threadsafety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures. The traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. In response, researchers have investigated a variety of alternative synchronization techniques that do not employ mutual exclusion. A synchronization technique is wait-free if it ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads. It is lock-free if it ensures only that some thread always makes progress. While waitfree synchronization is the ideal behavior (thread starvation is unacceptable), lock-free synchronization is often good enough for practical purposes (as long as starvation, while possible in principle, never happens in practice).The synchronization primitives provided by most modern architectures, such as compare-and-swap (CAS) or load-locked/store-conditional (LL/SC) are powerful enough to achieve wait-free (or lock-free) implementations of any linearizable data object [23]. The remaining paper will discussed about the different data structures, concurrency control methods and various techniques given for the concurrent access to these data structures.

2. Data Structures

We can store and organize data in diffrent ways where data structure is the one example to do so.Various data structures are available to make the processing easy to represent the data in the memory of the computer.These data structures have diffrent features and these features should be kept in mind while choosing to use. Logical and mathematical model are used in data structure to organize the data. So diffrent kind of data structures are suited to diffrent kinds of applications.Data structure are divided into two categories, which are **linear data structure** and **non-linear data structure.**

1.Linear Data Structure: this data structure is one in which elements of data forms a sequence. The most, simplest linear data structure is a 1-D array.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Figure:- A 1-D array of 6 elements.

2.Non-Linear Data Structure: this is the one in which its elements do not form a sequence.it means, unlike a linear data structure, each element is not considered to have a unique predecessor and a unique successor. Graphs and trees are the two data structure which comes under this category.

2.1 Linear Data Structure

As we have already described thet what the linear data structure is. So now we will discuss the types of linear data struture.

1.ARRAY: a finite collection of homogeneous elements is called an **ARRAY.** Here, the word 'homogeneous' indicates the the data types in all the elements in the collection should be same. That is **INT** or **CHAR** or **FLOAT** or any built in or user-define data type.

Elements of an array are always stored in contigous memory locations irrespective of the array size. The elements of an array can be referred to by using one or more indices or subscripts. An indices or a subscript is a +ve inteher vaalue, which indcates a position of a particular element in the array. If the number of subscriots required to access any particular element ia an array in one, then it is a **single-dimensional** array. Otherwise it is multi-dimensional array that may be **2-D, 3-d** array and so on.

Single-Dimesional Array: a single-dimensional array is defined as an array in which only one subscript value is used to access its elements. It is the simple and easy form of an array. Before using an array in program, it needs to be declared. Syntax of single-dimensional array's declaration in C is:-

Data_type array _name[Size];

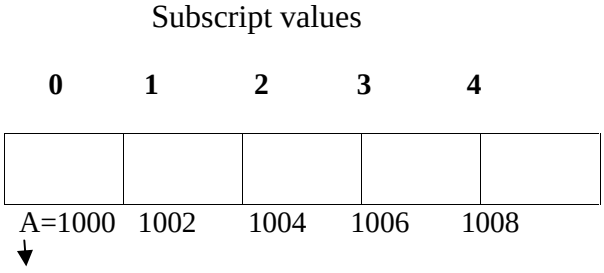


Figure 1.1 Base address and Subscript values of an Array

In this declaration, an integer array og five elements is declared. The array name A refers to the base address of the array.array elements are indexed from 0 to 4 .

Two-Dimensional Array: before we explain two-diensional array, we will have a look on syntax of declaring a two-dimensional array that is :-

Data_type array_name [row_size][column_size];

For example, in the statement int a[3][3], an integer array of three rows and three columns

are declared. Once a compiler reads a two-dimensional array declaration, it allocates a specific amount of memory for this array.

Row 1 Row 2 Row 3

1	2	3	4	5	6	7	8	9

Two-dimensional array in ROW major order

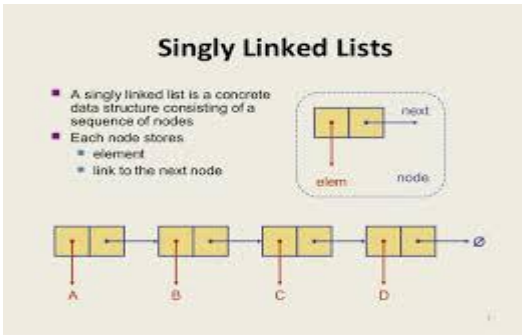
Col1 Col2 Col3

1	4	7	2	5	8	3	7	9

Two-dimensional array in column major order

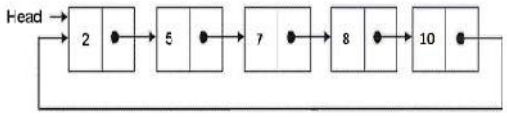
Linked Lists:- same withe arrays, there is also another name included in linear data structure which is called **Linked Lists.** Linked list is a linear collection of similar data elements(collection of elements listed by sequence), called nodes, with each node containing some data and pointer pointing to other nodes in the list.nodes of a linked list can be stored anywhere in the memory. The linear order of the list is maintained by the pointer field in each node.

Singly linked list:- in singly linked list, each node consist of two fields: info and next. The info field contains the data and next field contains the address of memory location where the next code is stored.the last node ofa singly linked list is NULL in its next field indicates the end of list.



(a) singly linked list

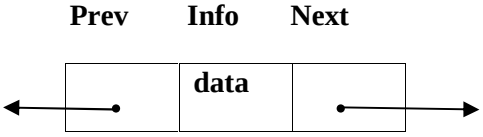
Circular linked list :- a linear linked list in which the next field of the last node points back to the first node instead of containing NULL values is called as **circular linked list**. The main advantage of circular linked list over linear linked list is that by starting with any node in th list, we can reach any of its predecessor nodes.

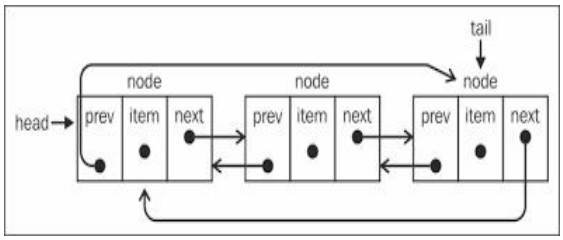


(b) circular linked list

Double linked list:- in singly linked list, each node contains a pointer to the next node and it has no information about the previous node.thus, we traverse only in one direction that is only from beginning to end and sometimes it is required to traverse in the backward direction that is from end to beginning. This can be implemented by maintaining an additional [pointer in each node of the list that points towards the end. Such types of linked lists are called **Double linked lists.**

Each node of doubly linked list consist of thre fields: prev,info and next. The info field contain data, the prev field contains the address of previous field and the next node contains the address of the next node.



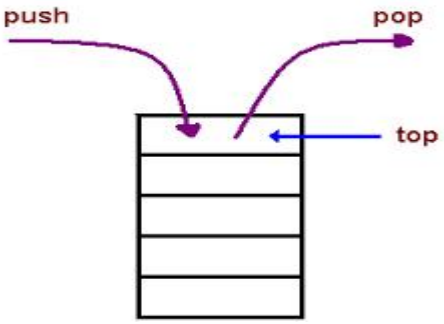


(b) doubly linked list.
(c)

a. Stack

A stack is a linear list of data elements in which addition or deletion of new elements occurs only at one end. This is called **TOP** of the stack. The operation of adding a new element and deleting a new element is the stack is called **PUSH** and **POP.** The last item added to the stack is removed first from the stack. Therefore a stack is called out **Last-in-First-Out (LIFO)** list.

A pile of books in the comman example of stack. A new to be added to the top of the pile is placed at the top and a book to be removed is also taken off from the top. The book that is most recently put on the pile is the first one to taken off from the pile. Similarly the book at the bottom is the last one to removed. Two operations are used in the stack that is **PUSH** and **POP.** Push is used to enter the values in the stack whereas POP is used to delete the values from the stack.

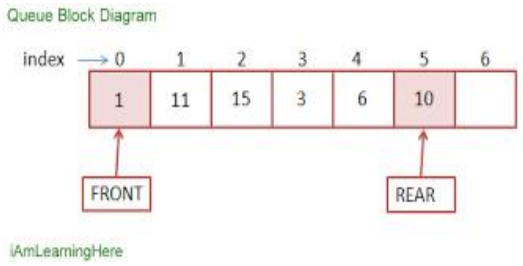


A stack with elements

In this diagram of stack push inserts the new element in the stack where as pop is used to delete the element from the stack. To insert the data element 1 in a STACK, TOP is incremented by 1 and the data item is stored at the top of the Stack. To pop the element from the stack, one element is taken off from the stack and TOP is decremented by 1. Similarly, other data items can be removed from the stack untill the stack is not empty.

b. Queue

A queue is a linear data structure in which addition or insertion of a new element occurs at the one end called **REAR** and deletion of an element occurs at the other end which is called **FRONT**. Since insertion and deletion occurs at the opposite ends of the Queue, the first element that is inserted in the queue is the first one to come out. Therefore this queue is also called **First-in-First-Out(FIFO)**.



A queue with elements

In this diagram 1 is the first element to insert in the queue and the first one to delete from the queue. Other elements will join in the end to the first element for insertion as well deletion.

2.2 NON-LINEAR DATA STRUCTURE

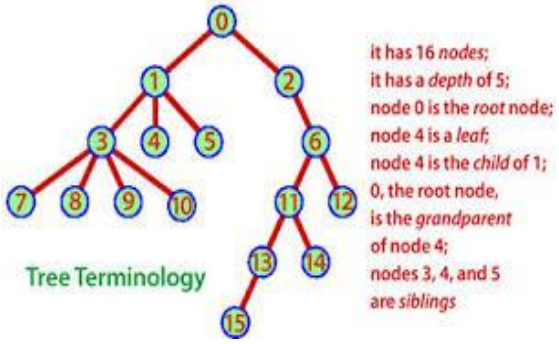
As earlier we have discussed linear data strutures in which data elements makes a sequence. But unlike linear data strutures, non-linear data in the one in which its elements do not form a sequence means each element is not constrained to have a unique predecessor and a unique successor. Trees and graphs are the two data structures which comes under this category.

a. Trees

Many a times, we observe hierarchical relationship between various data elements.

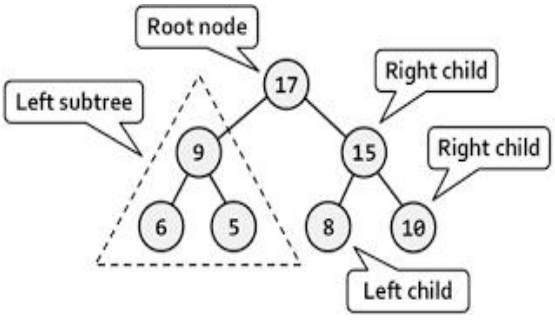
This hierarchical relationship between data elements can be easily represent using a non-linear data structure which is called a **tree**. A **tree** consist of multiple nodes with each node containing zero,one or more pointers to other nodes called child nodes. Each node of a tree

has one parent except a special node at the top of the tree called **root** node of the tree and the nodes at the lowest level are known as leaf nodes. The root node is a special node having no parent node and leaf nodes are the nodes having no child nodes. Any node having a child node as well as parent node is called internal node.



A Tree Structure

There is a special type of tree which is called BINARY tree, which can be either empty or has a finite set of nodes, such as one of the nodes is designated as ROOT node and the remaining nodes are partitioned into two subtrees of the root node is called **left subtree** and **right subtree**. The non-empty left sub-tree and right sub-tree are also binary sub-tree. Unlike a general tree,each node in a binary tree is restricted to have the most only two child nodes.

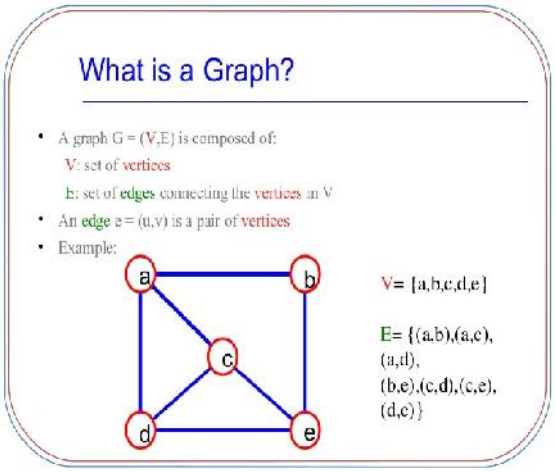


A binary tree

In the above binary tree, root node have left and right subtree , which also have their left and right subtrees respectively. Every node have only two nodes, not more than that.

b. Graph

Formally, a graph G(V,E) consists of a pair of two non-empty sets V and E, where V is a vector or nodes and E is set of Edges. Graphs are used to represent the non-hierarchical relationship among the pairs of data elements. The data is represented graphically by graph. The data elements become the vertices of the graph and the relationship is shown by the edges between two vertices.



A Graph

2.3 Concurrency Control

Simultaneous execution of multiple threads/process over a shared data structure access can create several data integrity and consistency problems:

• Lost Updates.

• Uncommitted Data.

• Inconsistent retrievals

All above are the reasons for introducing the concurrency control over the concurrent access of shared data structure. Concurrent access to data structure shared among several processes must be synchronized in order to avoid conflicting updates. Synchronization is referred to the idea that multiple processes are to join up or handshake at a certain points, in order to reach agreement or commit to a certain sequence of actions. The thread synchronization or serialization strictly defined is the application of particular mechanisms to

ensure that two concurrently executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes. Concurrency control techniques can be divided into two categories.

• Blocking

• Non-blocking

Both of these are discussed in below sub-sections.

a. Blocking

Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Many of the existing concurrent data structure algorithms that have been developed use mutual exclusion i.e. some form of locking.

Mutual exclusion degrades the system's overall performance as it causes blocking, due to that other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. The limitation of blocking approach are given below :-

• Priority Inversion: occurs when a high-priority process requires a lock holded by a lower-priority process.

• Convoying: occurs when a process holding a lock is rescheduled by exhausting its quantum, by a page fault or by some other kind of interrupt. In this case, running processes requiring the lock are unable to progress

• Deadlock: can occur if different processes attempt to lock the same set of objects in different orders.

• Locking techniques are not suitable in a real-time context and more generally, they suffer significant performance degradation on multiprocessors systems.

b. Non-Blocking

Non-blocking algorithm Guarantees that the data structure is always accessible to all processes and an inactive process cannot render the data structure inaccessible. Such an algorithm ensures that some active process will be able to complete an operation in a finite number of steps making the algorithm robust with respect to process failure [22]. In the following sections we discuss various non-blocking properties with different strength.

Wait-Freedom: A method is wait-free if every call is guaranteed to finish in a finite number of steps. If a method is bounded wait-free then the number of steps has a finite upper bound, from this definition it follows that wait-free methods are never blocking, therefore deadlock cannot happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.

Lock-Freedom: Lock-freedom is a weaker property than wait-freedom. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that some call finishes in a finite number of steps is not enough to guarantee that all of them eventually finish. In other words, lock-freedom is not enough to guarantee the lack of starvation.

Obstruction-Freedom: Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called obstruction-free if there is a point in time after

which it executes in isolation (other threads make no steps, e.g.: become suspended), it finishes in a bounded number of steps. All lockfree objects are obstruction-free, but the opposite is generally not true. Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

In the sequential setting, data structures are crucially important for the performance of the respective computation. In the parallel programming setting, their importance becomes more crucial because of the increased use of data and resource sharing for utilizing parallelism. In parallel programming, computations are split into subtasks in order to introduce parallelization at the control/computation level. To utilize this opportunity of concurrency, subtasks share data and various resources (dictionaries, buffers, and so forth). This makes it possible for logically independent programs to share various resources and data structures.

Concurrent data structure designers are striving to maintain consistency of data structures while keeping the use of mutual exclusion and expensive synchronization to a minimum, in order to prevent the data structure from becoming a sequential bottleneck. Maintaining consistency in the presence of many simultaneous updates is a complex task. Standard implementations of data structures are based on locks in order to avoid inconsistency of the shared data due to concurrent modifications. In simple terms, a single lock around the whole data structure may create a bottleneck in the program where all of the tasks serialize, resulting in a loss of parallelism because too few data locations are concurrently in use. Deadlocks, priority inversion, and convoying are also side-effects of locking. The risk for deadlocks makes it hard to compose different blocking data structures since it is not always possible to know how closed source libraries do their locking. Lock-free implementations of data structures support concurrent access. They do not involve mutual exclusion and make sure that all steps of the supported operations can be executed concurrently. Lock-free implementations employ an optimistic conflict control approach, allowing several processes to access the shared data object at the same time. They suffer delays only when there is an actual conflict between operations that causes some operations to retry. This feature allows lock-free algorithms to scale much better when the number of processes increases. An implementation of a data structure is called lock-free if it allows multiple processes/threads to access the data structure concurrently and also guarantees that at least one operation among those finishes in a finite number of its own steps regardless of the state of the other operations. A consistency (safety) requirement for lock-free data structures is linearizability [24], which ensures that each operation on the data appears to take effect instantaneously during its actual duration and the effect of all operations are consistent with the object's sequential specification. Lock-free data structures offer several advantages over their blocking counterparts, such as being immune to deadlocks, priority inversion, and convoying, and have been shown to work well in practice in many different settings.

The remaining paper will explore the access of different data structures like stack, queue, trees, priority queue, and linked list in concurrent environment. How the sequence of data structure operations changes during concurrent access. These techniques will be based on blocking and non-blocking.

III.Literature Review

a. Stack Data Structure

Stack is the simplest sequential data structures. Numerous issues arise in designing concurrent versions of these data structures, clearly illustrating the challenges involved in designing data structures for shared-memory multiprocessors. A concurrent stack is a data structure linearizable to a sequential stack that provides push and pop operations with the usual LIFO semantics. Various alternatives exist for the behavior of these data structures in full or empty states, including returning a special value indicating the condition, raising an exception, or blocking.

There are several lock-based concurrent stack implementations in the literature. Typically, lock-based stack algorithms are expected to offer limited robustness.

The first non-blocking implementation of concurrent link based stack was first proposed by Trieber et al [1]. It represented the stack as a singly linked list with a top pointer. It uses compare-and-swap to modify the value of Top atomically. However, this stack was very simple and can be expected to be quite efficient, but no performance results were reported for nonblocking stacks. When Michael et. al [2] compare the performance of Treiber's stack to an optimized nonblocking algorithm based on Herlihy's methodology [28], and several lock-based stacks such as an MCS lock in low load situations[29]. They concluded that Treiber's algorithm yields the best overall performance, but this performance gap increases as the degree of multiprogramming grows. All this happen due to contention and an inherent sequential bottleneck.

b. Queue Data Structure

A concurrent queue is a data structure that provides enqueue and dequeue operations with the usual FIFO semantics. Valois presented a list-based non blocking queue. The represented algorithm allows more concurrency by keeping a dummy node at the head (dequeue end) of a singly linked list, thus simplifying the special cases associated with empty and single-item. Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or reusing dequeued nodes. If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. Since memory is a limited resource, prohibiting memory reuse is not an acceptable option. Valois therefore proposed a special mechanism to free and allocate memory. The mechanism associates a reference counter with each node. Each time a process creates a pointer to a node it increments the node's reference counter atomically. When it does not intend to access a node that it has accessed before, it decrements the associated reference counter atomically. In addition to temporary links from processlocal variables, each reference counter reflects the number of links in the data structure that point to the node in question. For a queue, these are the head and tail pointers and linked-list links. A node is freed only when no pointers in the data structure or temporary variables point to it. Drawing ideas from the previous authors, Michel et.al [5] presented a new non-blocking concurrent queue algorithm, which is simple, fast, and practical. The algorithm implements the queue as a singly-linked list with Head and Tail pointers. Head always points to a dummy node, which is the first node in the list. Tail points to either the last or second to last node in the list. The algorithm uses compare and swap, with modification counters to avoid the ABA problem. To allow dequeuing processes to free dequeue nodes, the dequeue operation ensures that Tail does not point to the dequeued node nor to any of its predecessors. This means that dequeued nodes may safely be re-used.

The Mark introduced a scaling technique for queue data structure which was earlier applied to LIFO data structures like stack. They transformed existing nonscalable FIFO queue implementations into scalable implementations

using the elimination technique, while preserving lock-freedom and linearizability

In all previously FIFO queue algorithms, concurrent Enqueue and Dequeue operations synchronized on a small number of memory locations, such algorithms can only allow one Enqueue and one Dequeue operation to complete in parallel, and therefore cannot scale to large numbers of concurrent operations. In the LIFO structures elimination works by allowing opposing operations such as pushes and pops to exchange values in a pair wise distributed fashion without synchronizing on a centralized data structure. This technique was straightforward in LIFO ordered structures . However, this approach seemingly contradicts in a queue data structure, a Dequeue operation must take the oldest value currently waiting in the queue. It apparently cannot eliminate with a concurrent Enqueue. For example, if a queue contains a single value 1, then after an Enqueue of 2 and a Dequeue, the queue contains 2, regardless of the order of these operations.

c. Linked List Data Structure

Implementing linked lists efficiently is very important, as they act as building blocks for many other data structures. The first implementation designed for lock-free linked lists was presented by Valois . The main idea behind this approach was to maintain auxiliary nodes in between normal nodes of the list in order to resolve the problems that arise because of interference between concurrent operations. Also, each node in his list had a backlink pointer which was set to point to the predecessor when the node was deleted. These backlinks were then used to backtrack through the list when there was interference from a concurrent deletion. Another lock-free implementation of linked lists was given by Harris. His main idea was to mark a node before deleting it in order to prevent concurrent operations from changing its right pointer. The previous approach was simpler than later one. Yet another implementation of a lock-free linked list was proposed. The represented Technique used design to implement the lock free linked list structure. The represented algorithm was compatible with efficient memory management techniques unlike algorithm.

d. Tree Data Structure

A concurrent implementation of any search tree can be achieved by protecting it using a single exclusive lock. Concurrency can be improved somewhat by using a reader-writer lock to allow all read-only (search) operations to execute concurrently with each other while holding the lock.

e. Priority Queue Data Structure

The Priority Queue abstract data type is a collection of items which can efficiently support finding the item with the highest priority. Basic operations are Insert (add an item), FindMin (finds the item with minimum (or maximum) priority), and DeleteMin (removes the item with minimum (or maximum) priority). Delete Min returns the item removed.

IV. Conclusion

This paper reviews the different data structures and the concurrency control techniques with respect to different data structures (tree, queue, priority queue). The algorithms are categorized on the concurrency control techniques like blocking and non-blocking. Former based on locks and later one can be lock-free, wait-free or obstruction free. In the last we can see that lock free approach outperforms over locking based approach.

References

[1] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. “An efficient algorithm for concurrent priority queue heaps.”

Information Processing Letters, 60(3): 151–157, November 1996

[2] LOTAN, N. SHAVIT. “Skiplist-Based Concurrent Priority Queues”, International Parallel and Distributed Processing Symposium, 2000.

[3] M. Greenwald. “Non-Blocking Synchronization and System Design.” PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, A, 8 1999.

[4] N. Shavit and D. Touitou. “Elimination trees and the construction of pools and stacks.” Theory of Computing Systems, 30:645–670, 1997.

[5] R. Ayani. “LR-algorithm: concurrent operations on priority queues.” In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, pages 22–25, 1991

[6] W. Pugh. Skip Lists: “A Probabilistic Alternative to Balanced Trees.” In Communications of the ACM, 33(6):668{676, June 1990.