CHAPTER 2
MODIFIED SKIP LIST & LITERATURE SURVEY

This thesis draws from several areas of concurrent data structure access i.e. locking approach based on mutual exclusion, non-blocking concurrent access, and correctness proof of approach applied for concurrent access. The following sections review relevant aspect of these areas and identify the specific ideas that which have influenced the work.

2.1 Preliminary

The data structures which are the preliminary of this thesis proposed work are linked list, skip list, modified skip list and priority queue. The modified skip list is the ground of my work of area.

2.1.1 Linked List

A linked list is a generalized data structure that can be used to implement several abstract data types. It is a dynamic data structure, where each individual node has two neighbor nodes, one previous and one next. The linked list can be either singly linked or doubly linked. In a singly linked list each node contains information in the form of a reference to the next node, and in a doubly linked list each node also has a reference to the previous node. New nodes can be inserted anywhere, arbitrary nodes can be deleted and the whole list can be traversed from one end to the other. Figure 2.1 is a general structure of a linked list.

![Figure:2.1 Linked List](image)
2.1.2 WHAT IS SKIP LIST?

Skip list data structure is impartially a simulation of a binary search tree (Pugh [97]) It began with the idea how we can make sorted linked lists better. In a linked lists it is easy to do operations like insertion, and deletion, but it is difficult to locate items efficiently because while searching there is need to walk through the list one item at a time. If we could “skip” over lots of items at a time, then we could solve this problem, one way to think of skip lists is as a hierarchy of sorted linked lists, stacked one on top of other.

An example of a skip list is shown in Figure 2.2. It is customary to visualize a skip list $S$ with list $S_0$ at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to $h$ as the height of skip list $S$.

![Figure 2.2 A skip list up to 6 level](image)

Intuitively, the lists are set up so that $S_{i+1}$ contains more or less every other entry in $S_i$. Entries in $S_{i+1}$ are chosen at random from the entries in $S_i$ by picking each entry from $S_i$ to also be in $S_{i+1}$ with probability $1/2$. i.e, in essence, we "flip a coin" for each entry in $S_i$ and place that entry in $S_{i+1}$ if the coin comes up "heads." Thus, we expect $S1$ to have about $n/2$ entries, $S2$ to have about $n/4$ entries, and, in general, $S_i$ to
have about \( \frac{n}{2} \) entries. In other words, we expect the height \( h \) of \( S \) to be about \( \log n \). The halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists, however. This structure uses randomization and has a probabilistic time complexity of \( O(\log n) \) where \( n \) is the maximum number of elements in the list.

### 2.1.3 MODIFIED SKIP LIST

It is a data structure introduced by [Cho & Sahni [84]] in which each node has one data field & three pointer fields: left, right and down. The left and right fields are used to maintain each level \( l \) chain as a doubly list and the down field of a node \( x \) at level \( l \) points to the leftmost node in the level \( l-1 \) chain that has key value larger than the key in \( x \).

![Figure: Modified Skip List with 4 levels](image)

In the above figure 2.3 each element is in exactly one double linked. The maximum level of the presented MSL is four. Each respective chain worked as a double linked list, \( H \) and \( T \), respectively points to the head & tail of the current chain.
2.1.3.1 Operations on MSL

The operation that we can perform on MSL are searching, insertion, deletion. The probalistic complexity of the these operations is \( O(\log n) \). The underneath sections describe the operations performed on MSL.

The node structure used for each node of MSL chain is:

```c
struct
{
    int value
    struct node * left
    struct node * right
    struct node * down
    int key
}
```

Figure:2.4 Structure of MSL node

2.1.3.1.1 Searching

Suppose we are given a search key \( k \). We begin the MSL Search method by setting a position variable \( p \) to the top-most head. If \( p \) is null, then the search terminates otherwise we locate for the key which is the largest entry in MSL top chain with key less than or equal to the search key \( k \). After every check, we also check is the present key is searched key if not the loop continue otherwise it ends with successful search operation Otherwise, we drop down to the next lower level in the present tower by setting \( p \rightarrow \text{left} \rightarrow \text{down} \). The search process also saved the predecessor and successor of a searched node \( k \) for further reference in local variable save. An algorithm for searching of key \( k \) is given in figure:2.5
The algorithm for insertion of new entry $d$ in existing MSL and explanations of the steps performed also given underneath figure 2.6.

The insertion algorithm for modified skip lists uses random number generator to decide the level for new entry. We begin the insertion of a new entry $(d)$ by performing the search_node$(d \rightarrow key)$ operation. To check the existence of duplicate of the new entry in existing MSL $H$, it also saves the preceding and succeeding node help in insertion. If the duplicate entry found the insert operation returns fail and stop otherwise it check whether the generated level $k$ is less than or more than existing level i.e current_level. If it is more then current_level. The new level is created with their respective head and tail. we call this new node as $x$ in the presented algorithm.

Figure: 2.5 searching in MSL

2.1.3.1.2 Insertion

The algorithm for insertion of new entry $d$ in existing MSL and explanations of the steps performed also given underneath figure 2.6.

The insertion algorithm for modified skip lists uses random number generator to decide the level for new entry. We begin the insertion of a new entry $(d)$ by performing the search_node$(d \rightarrow key)$ operation. To check the existence of duplicate of the new entry in existing MSL $H$, it also saves the preceding and succeeding node help in insertion. If the duplicate entry found the insert operation returns fail and stop otherwise it check whether the generated level $k$ is less than or more than existing level i.e current_level. If it is more then current_level. The new level is created with their respective head and tail. we call this new node as $x$ in the presented algorithm.

```
Algorithm: Search_Node(k)
{
1. p=head
2. while(p # NULL)
   {
3. while(p \rightarrow data \rightarrow key < k)
               {
4. save=p
5. p=p \rightarrow right
6. if(p \rightarrow data \rightarrow key == k)
7. report & stop
               
8. else
9. p=save \rightarrow down.
   }
}
```
The left and right pointers of new node x are updated. The down pointer of x is updated according to the status of variable k. (if k>current_level) then the appropriate node at level current_level-1 is updated accordingly. (if k=1) then the down pointer is set as null, if it is not, then there is need to set the down field of x as well as down field of node at level k+1 and k-1.

Algorithm: Insert_node(d) begin

1. Randomly generate the level k at which new node d to be inserted.
2. Search the MSL H, for d\rightarrow key saving information useful for insertion.
3. If d\rightarrow key exist in H then
4. Write “duplicate value “
5. Return
6. Else
7. Get a new node x and set x\rightarrow data=d
8. If(k>current_level) then
9. Current_level=current_level+1
10. Create a new chain in MSL H, with head node, node x and tail
11. Connect this chain to H
12. Update H
13. Set x\rightarrow down to the appropriate node of current-1 chain
14. if K=1
15. update the left and right pointer of new node x as well as preceding and succeeding of new node x.
16. set x\rightarrow down to null
17. else
18. update the left and right pointer of new node x as well as preceding and succeeding of new node x.
19. set x\rightarrow down to appropriate node node of k-1 chain
20. update the down field of the appropriate node at level k+1
21. end

Figure: 2.6 insertion in MSL
2.1.3.1.3 Deletion

Like the search and insertion algorithms, the deletion algorithm for a modified skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to perform a delete(y) operation. We begin by executing method Search(x). If the searched key stores an entry with key different from y, we return null. Otherwise, we remove x and set the down pointer of node at level k+1 which is pointing to node x. The below figure 2.7 is representing the algorithm of deletion.

Algorithm:Delete(y)
Begin
    1. Search the MSL,H for a node x with d->key =y, saving information useful for deletion.
    2. If (d->key!=y) is then
    3. Write “fail delete operation”, and return
    4. Else
    5. Let k be the level at which z is found
    6. Find the node p at level k+1 whose down pointer pointing to the node x i.e p->down=x
    7. Set p->down=x->right
    8. delete x from the level k chain.
    9. end

Figure:2.7 Deletion in MSL

2.1.4 Priority Queue

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), Find Min (finds the item with minimum (or maximum) priority), and Delete Min (removes the item with minimum (or maximum) priority). Delete Min returns the item removed.
2.2 Literature survey

For the purpose of this thesis the literature survey covers a period of 1983 to 2010. The literature work on “Modified Skiplist Data Structure & Concurrent Access” divided into two main areas.

1 Concurrent Access of data structure with locking approach
2 Lock-free approach and concurrent Access to data structure

2.2.1 Concurrent Access Data Structure with Locking Technique.

{Michael and Scott[61]} presented a linearizable lock-based stack having a representation of linear linked lists with a top pointer and a global lock is used to control the access to stack. This kind of implementation scale poorly because even after reducing the contention on lock, the top of the stack is still a bottleneck, again {Michael and Scott [55] } presented a lock-based queue that improve on naïve single-lock approach by using separate locks for the head and tail pointers of a linked list based queue. This approach allows enqueue operation to execute in parallel with a dequeue operation. The approach of using “dummy” nodes in the queue avoid acquiring both the head and tail locks in the case that the queue is empty, and therefore it avoids deadlock.

A concurrent implementation of any search tree structure can be fulfilled by using a single exclusive lock. Concurrency can be improved somewhat by using a reader-writer lock to allow all read-only like search operations to execute concurrently with each other while holding the lock in shared mode, while update (insert or delete) operations exclude all other operations by acquiring the lock in exclusive mode. By using fine-grained locking strategies that is using
one lock per node, rather than a single lock for the entire tree we can improve concurrency further. [Kung and Lehman [4]] in his paper "Concurrent manipulation of binary search trees." presented a concurrent binary search tree implementation in which the update operations hold only a constant number of node locks at a time, and the search operations are never blocked. However, this kind of implementation never makes an attempt to keep the search tree balanced.

In the context of concurrent access to B+-trees is given by {Lehman and Yao [69]} in his paper "Efficient Locking for Concurrent Operations on B-trees", who define B^{link}-trees: B+-trees with "links" from each node in the tree to its right neighbor at the same level of the tree. These links allow us to "separate" the splitting of a node from modifications to its parent to reflect the splitting. Specifically, in order to split a node n, we can create a new node n_0 to its right, and install a link from n to n_0. If an operation that is descending the tree reaches node n while searching for a key position that is now covered by node n_0 due to the split, the operation can simply follow the link from n to n_0 to recover.

This allows a node to be split without preventing access by concurrent operations to the node’s parent. As a result, update operations do not need to simultaneously lock the entire subtree. The algorithm update operations as well as search operations use the lock coupling technique so that no operation ever holds more than two locks at a time, which significantly improves concurrency. This technique has been further refined by {Sagiv [98]} “Concurrent operations on b-trees with overtaking.” so that operations never hold more than one lock at a time.
Lehman and Yao do not address how nodes can be merged, instead allowing delete operations to leave nodes under full.

In a concurrent linked list global locking is used to prevent concurrent manipulation, the most popular approach to concurrent lock based linked lists is lock coupling (Bayer and Schkolnick [79], Lea [42]). In this approach, each node has an associated lock. A thread traversing the linked list releases a node’s lock only after acquiring the lock of the next node in the list.

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). DeleteMin returns the item removed. the concurrent access of priority queue in the literature are linearizable versions of the heap structures. Again, the basic idea is to use fine grained locking of the individual heap nodes to allow threads accessing different parts of the data structure to do so in parallel where possible. A key issue in designing such concurrent heaps is that traditionally insert operations proceed from the bottom up and delete-min operations from the top down, which creates potential for deadlock. {Biswas and Browne [29]} in “Simultaneous update of priority structures.” presented a lock-based heap algorithm assuming specialized “cleanup” threads to overcome deadlocks. {Rao and Kumar. [83]} in his paper “Concurrent access of priority queues.” suggest to overcome the drawbacks of previous one using an algorithm that has both insert and delete-min operations proceed from the top down. {Hunt, Michael, and Parthasarathy, [13]} in “An efficient
algorithm for concurrent priority queue heaps.”, present a heap based algorithm that overcomes many of the limitations of the above schemes, especially the need to acquire multiple locks along the traversal path in the heap. It proceeds by locking for a short duration a variable holding the size of the heap and a lock on either the first or last element of the heap.

Unfortunately, the empirical evidence shows, the performance of {Hunt, Michael, and Parthasarathy, [13]} does not scale beyond a few tens of concurrent processors. As concurrency increases, the algorithm's locking of a shared counter location, introduces a sequential bottleneck that hurts performance. The root of the tree also becomes a source of contention and a major problem when the number of processors is in the hundreds.

In summary, both balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention. The solution proposed by {Lotan And Shavit.[10]} “SkipList-Based Concurrent Priority Queues”, algorithm was to design concurrent priority queues based on the highly distributed SkipList data structures of {Pugh [96], [11]}.

Author introduced the SkipQueue, a highly distributed priority queue based on a simple modification of Skip List aorithm. Inserts in the SkipQueue proceed down the levels as in {Pugh[96], [97]}. For Delete-min, multiple minimal" elements are to be handed out concurrently. This means that one must coordinate the requests, with minimal contention and bottlenecking, even though Delete-mins are interleaved with Insert operations. The solution was as follows, keep a specialized delete pointer which points to the current minimal item in this list. By
following the pointer, each Delete-min operation directly traverses the lowest level list, until it finds an unmarked item, which it marks as deleted.” It then proceeds to perform a regular Delete operation by searching the SkipList for the items immediately preceding the item deleted at each level of the list and then redirecting their pointers in order to remove the deleted node.

2.2.2 Lock-free approach and concurrent Access to data structure.

{Treiber [80]} proposed a lock-free concurrent stack implementation. Here the stack is represented as a singly-linked list with a top pointer and used CAS to modify the value of the top pointer atomically. {Michael and Scott [57]} compare the performance of Treiber’s stack to an optimized nonblocking algorithm based on {Herlihy and Wing [49]}, and several lock-based stacks such as an MCS lock {Mellor and Scott [36]} in low load situations. They concluded that Treiber’s algorithm yields the best overall performance, and that this performance gap increases as the degree of multiprogramming grows. However, because the top pointer is a sequential bottleneck, even with an added backoff mechanism to reduce contention, the Treiber stack offers little scalability as concurrency increases{Hendler, Shavit and yerushalmi [7]}. The author observe that any stack implementation can be made more scalable using the elimination technique of {Shavit and Touitou [66]}. Elimination allows pairs of operations with reverse semantics—like pushes and pops on a stack to complete without any central coordination, and therefore substantially aids scalability. The idea is that if a pop operation can find a concurrent push operation to “partner” with, then the pop operation can take the push operation’s value, and both operations can return immediately. The net effect of
each pair is the same as if the push operation was followed immediately by the pop operation, in other words, they eliminate each other’s effect on the state of the stack. Elimination can be achieved by adding a collision array from which each operation chooses a location at random, and then attempts to coordinate with another operation that concurrently chose the same location.

A concurrent queue is a data structure that provides enqueue and dequeue operations with the usual FIFO semantics. An algorithm for linked list based queue was introduced by the Hendler and Shavit.[6] in his paper “Work dealing”. Valois[30] in his paper on “Implementing Lock-Free queues.” presented a list-based nonblocking queue. The 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.

The author proposed a special mechanism to free and allocate memory by associating 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. Lamport [40] presented a lock-free (actually wait-free) implementation of a queue based on a static array, with a limited
concurrency supporting only one producer and one consumer. {Giacomoni [32] } presented a cache-aware modification which, instead of using shared head and tail indices, synchronize directly on the array elements. {Herman and Damian-Iordache [93] } outlined a wait-free implementation of a shared queue for any number of threads, although non-practical due to its high time complexity and limited capacity. {Gong and Wing [16] } and later {Shann [90] } presented a lock-free shared queue based on a cyclic array and the CAS primitive, though with the drawback of using version counters, thus requiring double-width CAS for storing actual items. {Tsigas and Zhang [87] } presented a lock-free extension of {Lamport [82] } for any number of threads where synchronization is done both on the array elements and the shared head and tail indices using CAS, and the ABA problem is avoided by exploiting two (or more) null values.

A survey done by {Michael and Scott [3] } in his work “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” Describes numerous flawed attempts at devising general multiple enqueuers, multiple dequeuers nonblocking queue implementations. Drawing ideas from previous work author presented an algorithm that 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 & Daniel [62]] introduced a scaling technique for queue data structure which was earlier applied to LIFO data structures like stack. They transformed existing non scalable 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 [Shavit and Touitou [67]]. 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.

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. [35]] in his paper on “Lock-free linked lists using compare and-swap.” 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 [23]} in “A pragmatic implementation of non-blocking linked-lists.”, 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 by {Michael [24]} in “High performance dynamic lockfree hash tables and list-based sets.” The represented technique used {Harris [95]} design to implement the lock free linked list structure. The represented algorithm was compatible with efficient memory management techniques unlike {Harris [95]} algorithm.

A double-linked list implementation of dictionary based on the non-available CAS2 atomic primitive presented by {Fomitchev and Ruppert [44]}. {Greenwald [47]} {Attiya and Hillel [17]} presented a CAS2-based implementation that also supports disjoint-access parallelism. A more general doubly-linked list implementation supporting general list semantics was presented by {Sundell and Tsigas [74]}

The doubly-ended queue abstract data type is a collection of items in which the earliest as well as the latest added item may be accessed. Basic operations are PushLeft (add to the head), PopLeft (remove from the head), PushRight (add to the tail), and PopRight (remove from the tail). PopLeft and PopRight return the item removed. Large efforts have been put on the work on so called work-stealing deques. These data structures only support three operations and with a limited level of concurrency, and are specifically aimed for scheduling purposes. {Arora [75]}
presented a lock-free work-stealing deque implementation based on the CAS atomic primitive. {Hendler [8]} improved this algorithm to also handle dynamic sizes. Several lock-free implementations of the deque abstract data type for general purposes, although based on the non-available CAS2 atomic primitive, have been published in the literature [46, 68, 70, 56]. {Michael [56]} presented a lock-free deque implementation based on the CAS primitive.

There exist several algorithms and implementations of concurrent priority queues. The literature on concurrent priority queues consists mostly of algorithms based on two paradigms: search trees and heaps. Most of the concurrent priority queue have been proposed, usually based on heap structure tree{Shavit & Touitou [repeat], Mohan [4], Dominique, Orlarey & Stephane [10]}. Nonblocking linearizable heap-based priority queue algorithms have been proposed by {Herlihy [50]} in “A methodology for implementing highly concurrent data objects.”, {Barnes [12]} in “Wait free algorithms for heaps”, and {Israeli and Rappoport.[28]} in “Efficient wait-free implementation of a concurrent priority Queue”. {Sundell and Tsigas.[21]} in “Fast and Lock-Free Concurrent Priority Queues for Multithread System” given an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multiprocessor) systems as well as pre-emptive (multiprocess) systems. The algorithm was based on the {Pugh [97]} “Skip Lists: A Probabilistic Alternative to Balanced Trees.” This structure uses randomization and has a probabilistic time complexity of $O(\log N)$ where $N$ is the maximum number of elements in the list. In order to make the SkipList construction concurrent and non-blocking; author used three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS).
### 2.3 Comparison & Analysis

These issues motivated us to compile the literature survey, and presented as concurrent algorithm applied on different data structure with their respective merits & demerits. The aim of this survey is to research and develop efficient concurrent data structures with distinct operations applied on them.

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Algorithm</th>
<th>Merits</th>
<th>Demerits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Systems programming: Coping with parallelism</td>
<td>Simple and can be expected to be quite efficient.</td>
<td>Contention and an inherent sequential bottleneck.</td>
</tr>
<tr>
<td></td>
<td>A scalable lock-free stack algorithm</td>
<td>Due to elimination technique there is high degree of parallelism.</td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>Implementing Lock-Free queues.</td>
<td>Algorithm no longer needs the snapshot. since the only intermediate state that the queue can be in is if the tail pointer has not been updated</td>
<td>Required either an unaligned compare &amp; swap or a Motorola like double-compare and swap, both of them are not supported on any architecture.</td>
</tr>
<tr>
<td></td>
<td>Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.</td>
<td>The algorithm was simple, fast and practical .it was the clear algorithm of choice for machine that provides a universal atomic primitive.</td>
<td>Pointers are inserted using costly CAS</td>
</tr>
<tr>
<td></td>
<td>Using elimination to implement scalable and</td>
<td>1. scaling technique, allows multiple enqueue and dequeue</td>
<td>1. The elimination back off queue is practical only for very short queues as in</td>
</tr>
<tr>
<td>Type</td>
<td>Description</td>
<td>Properties</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>lock-free FIFO queues.</td>
<td>operations to complete in parallel. 2. The concurrent access to the head and tail of the queue do not interfere with each other as long as the queue is non-empty.</td>
<td>order to keep the correct FIFO queue semantics, the enqueue operation cannot be eliminated unless all previous inserted nodes have been dequeued. 2. scalable in performance as compare to previous one but having high overhead.</td>
<td></td>
</tr>
<tr>
<td>Concurrent manipulation of binary search trees.</td>
<td>Algorithm never blocked the search operations</td>
<td>Search tree is not balanced</td>
<td>Tree</td>
</tr>
<tr>
<td>Efficient Locking for Concurrent Operations on B-trees,</td>
<td>Small number of locks used</td>
<td>Expansive locks</td>
<td>A symmetric concurrent b-tree algorithm</td>
</tr>
<tr>
<td>A symmetric concurrent b-tree algorithm</td>
<td>They involved the merging as a part of deletion.</td>
<td>Expansive locking</td>
<td>An efficient algorithm for concurrent priority queue heaps</td>
</tr>
<tr>
<td>Allows concurrent insertion and deletion in opposite direction.</td>
<td>The performance does not scale beyond a few tens of concurrent processors.</td>
<td>Priority queue</td>
<td></td>
</tr>
<tr>
<td>Skip list-Based Concurrent Priority Queues</td>
<td>Designed a scalable concurrent priority queue for large scale multiprocessor.</td>
<td>Algorithm based on locking approach.</td>
<td>Fast and Lock-Free Concurrent Priority Queues for Multithread System.</td>
</tr>
<tr>
<td>This was a first lock-free approach for concurrent priority queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list</td>
<td>A highly concurrent priority queue based on the b-link tree</td>
<td>Avoid the serialization bottleneck</td>
<td>Needs node to be locked in order to be rebalance</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Lock-free linked lists using compare-and-swap</td>
<td>Reduced interference of concurrent operations using backlink nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A pragmatic implementation of non-blocking linked-lists</td>
<td>For making successful updating of nodes, every node to be deleted was marked</td>
<td>Difficult to implement</td>
</tr>
<tr>
<td></td>
<td>High performance dynamic lock-free hash tables and list-based sets.</td>
<td>Efficient with memory management techniques</td>
<td>Poor in performance.</td>
</tr>
</tbody>
</table>

Figure: 2.8 Comparison of various techniques to concurrent Access