CHAPTER 4
LOCK-FREE CONCURRENT ACCESS TO MSL

4.1 INTRODUCTION

In this era the trend of increasing software demands continues consistently, the traditional approach of faster processes comes to an end, forcing major processor manufactures to turn to multi-threading and multi-core architectures, in what is called the concurrency revolution. At the heart of many concurrent applications lie concurrent data structures. Concurrent data structures coordinate access to shared resources; implementing them is hard. Designing concurrent access of 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 thread-safety 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 approach that helps maintaining data integrity among threads is to use lock primitives. Mutexes, semaphores, and critical sections are used to ensure that certain sections of code are executed in exclusion[Eshcar Hillel,[48] ]

The main goal of this chapter is to provide an efficient and practical lock-free implementation of modified skip list data structure. That is
suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. The approach discussed in previous chapter for concurrent MSL based on mutual exclusion{Pushpa , Ranjeet[77]}, Causes blocking which has several drawbacks and degrades the system’s. The non-blocking algorithms have been shown to be of big practical importance in practical applications {Tsigas,[50],[51]}

4.2 Concurrent operations on MSL

This chapter describes the simple concurrent algorithms for access and update of MSL. Our algorithm is based on the approach of {Sundell, [23]}. The algorithm is implemented using common synchronization primitives that are available in modern systems. Double link list is used as a basic data structure of modified skip list. Due to concurrent structure of MSL, shared memory multiprocessor system configuration is contemplated in Figure 4.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can run many tasks at a time. The co-operating tasks, probably running on different processors, use shared data objects fabricated in the shared memory to co-ordinate and communicate. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory {Sundell, [23]}. 
4.3 Algorithm

We present a concurrent modified skip list algorithm with lock-free technique supporting three methods, search_node, insert_node, del_node:

**search_node (key):** search for a node with key k equal to key, and return true if key found otherwise return false.

**Insert_node (d):** inserts adds d to the set and returns true iff d was not already in the set

**del_node (v):** removes v from the set and returns true iff v was in the set

For insertion or deletion of a node from the concurrent modified skip list, we need to change the respective set of prev and next pointers. These pointers have to be changed consistently, but it is not obligatory to change them at once. The presented algorithm is based on the approach of {Sundell,[23]} according to that we can think through the doubly linked list as being a singly linked list with auxiliary information in the left pointers, with the right pointers being updated before the left pointers. Thus, the right pointers always form a consistent singly linked list and thus define the nodes positional relations in the logical abstraction of the doubly linked list, but the left
pointers only give hints as to where to find the previous node. The down pointer of modified skip list is according its criteria, as we explained in chapter 1. The beneath figure 4.2 shows the field of a node.

```
Union link::word
<p, d> :< pointer to node, Boolean>
Structure Node
{
Value: pointer to word
left: union link
right: union link
down :union link
}
//local variables
Node, prev, prev2, next, next2:pointer to node
Last,link1: union link
Union link::word
<p, d> :< pointer to node, Boolean>
```

Figure:4.2 Node Structure & Variable used

4.3.1 Memory Management Policy

The concurrent traversal of nodes in MSL makes an incessant allocation and reclamation of nodes, in such kind of scenario several aspects of memory management need to be taken into account, like no node should be reclaimed and then later re-allocated while some other process is traversing this node. Our algorithm has applied the technique of reference counting and chosen the lock-free memory management scheme invented by {Valois, [54]} and corrected by {Michael, [56]}, which makes use of the FAA, TAS and CAS atomic synchronization primitives. These primitives are already discussed in chapter 2.
issues related to memory management during the concurrent access of MSL are:

- One problem, that arises with non-blocking implementations of MSL that are based on the linked-list structure, is when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted then the new node will be deleted as well, as illustrated in Figure 4.3. Our algorithm of insertion has resolved this issue with the latest method introduced by {Harris [94]} is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. Any concurrent insert operation will then be notified about the deletion, when its CAS operation will fail.

![Figure: 4.3 Concurrent Insertion & Deletion of Node](image)

- Another issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management:
**READ_NODE(node ** address):** De-reference the pointer and increase the reference counter for the corresponding node. In case the pointer is marked, NULL is returned.

**RELEASE_NODE(node: pointer to node):** Decrement the reference counter on the corresponding given node. If the reference count reaches zero, then call RELEASE_NODE on the nodes that this node has owned pointers to, then reclaim the node.

**COPY_NODE(node: pointer to Node):** pointer: Increase the reference counter for the corresponding given node.

### 4.3.2 search_node

In concurrent environment, while searching for a node in MSL a processes will eventually reach nodes that are marked to be deleted (line 10). It might be due to forcefully preemption of deletion process that invoked the corresponding operation.

The searching operation helps the delete process (help_del) to finish the pending work before continuing the search operation. However, it is only necessary to help the part of the delete operation on the current level in order to be able to traverse to the next node. The search operation traverses in several steps through the next pointers (starting from node1) at the current level until it finds a node that has the same or higher key value than the given key (lines 8-20). See Figure 4.4.
Flow Chat (4-1) : Searching in lock Free MSL
Flow Chat (4-2) : Searching in lock Free MSL
Figure 4.4 searching in lock free MSL

```c
Pointer to node function search_node (int v)
{
    node *t, *t_right, *save[maxlevel], *found_node
    int i
    t = COPY_NODE(head)
    t = head->right
    while (t != NULL)
    {
        while (t->value < v)
        {
            if (is_marked(t->right))
                t = help_Del(t)
            t_right = read_node(t->right)
            save[i] = t
            i = i - 1
        }
        if (t->value == v)
            break
        else
        {
            t = t->left->down
            i = i - 1
        }
    }
    found_node = t
    return found_node
}
```
4.3.3 Insert_node

The insert operation is shown in figure: 4.5, the algorithm begins with a search_node operation in (lines 5-23) to find the node after which the new node should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found. While doing a search if a node is found whose deletion mark is set, then the search_node operation calls the help_del (line no. 10), to delete the mark node. If there exist already a node with key same as new node with value v (line no 14), then insertion algorithm exit otherwise it searched for node with key value more than the new node value v. After inserting a new node, it is possible that it is deleted by a concurrent delete_node operation before it has been inserted completely at the particular level.

The level at which new node to be inserted depends on the level generated by random no. generator function in (k). If the value of k is more than the current _level the new level is created for the new value, with new head and tail. The down pointer of new node is set, according to the (lines 43-44) of given algorithm. If (k<current_level) then the new node to be inserted in between (line no 49) . The algorithm step(lines 73-83) tries to update the left pointer of the next node. The linearizability point of the insert operation is the successful CAS operation in (line 61). The main steps for insertion algorithm are:

(I) after setting the new node’s left and right pointers, atomically update the right pointer of the to-be-previous node,

(II) Atomically update the left pointer of the to-be-right node.

(III) Atomically update the down pointer of nodes according to the value generated by random no. generator function.
Flow Chat (5-1): Insertion in lock Free MSL
Flow Chat (5-2) : Insertion in lock Free MSL

1. The down pointer of node at K-1 level is updated accordingly.

2. If K < temp
   - Insert the new node x after save [k]

3. For inserting the new node after save [k] find out previous of next of new node to be inserted

4. While TRUE DO
   - If prev right, l = next
     - Yes: Release node of next = Read node (prev right)
     - No: Use the CAS Synchronization primitive to update the next pointer of previous of to be inserted node

5. Break back-off

6. Prev2 = copy node

7. Break back-off

8. While T DO

I
Flow Chat (5-3): Insertion in lock Free MSL

1. To check whether the previous of node to be updated is marked for deletion
2. Now again using CAS, update the previous pointer of next of the node to be deleted if it is updated
3. If K=1
   - Yes
5. We need to update the downstream filled of node according to the value of K
6. STOP
Function insert_node( value: pointer to word)

1. node*t,*save[max],*t_right,*found_node
2. k=randomlevel ()
3. temp, i = current_level
4. t=COPY_NODE (head)
5. while (t !=NULL) 
6. {
7. while( t->key<key)
8. { 
9. If ( IS_MARKED (t->right))
10. t=help_Del(t)
11. save[i]=t
12. t=READ_NODE(t->right)
13. }
14. if (t->key= =key) 
15. break
16. else
17. {
18. t=t->left->down
19. i=i-1
20. save[i]=t
21. }
22. }
23. found_node=t
24. new_node=create_node ( value)
25. node1=COPY_NODE(head)
26. If(k>temp)
27. {


Function insert_node( value: pointer to word)

28. //create new head and tail
29. h1=createnode(∞)
30. copy_node(h1)
31. h1→left=null
32. h1→right=x
33. h1→down=head
34. RELEASE_NODE(H1)
35. t1=CreateNode(∞)
36. COPY_NODE(t1)
37. t1→left=x
38. t1→right=NULL
39. t1→down=tail
40. RELEASE_NODE(t1)
41. New_node→left=h1
42. New_node→right=t1
43. if((save[k-1]→right→down)==NULL
    OR(save[k-1]→right→value>new_node→value))
44. New_node→down=save[k-1]→right
45. RELEASE_NODE(new_node)
46. RELEASE_NODE(t1)
47. RELEASE_NODE(h1)
48. }
49. node*t,*save[max],*t_right,*found_node
50. k=randomlevel()
51. temp, i = current_level
52. t=COPY_NODE(head)
53. while (t!=NULL)
54. {
55. while( t→key<key)
Function insert_node( value: pointer to word)

57.  If ( IS_MARKED (t→right))
58.    t=help_Del(t)
59.    save[i]=t
60.    t_right=READ_NODE(t→right)
61.  }
62.  if (t→key==key)
63.    break
64.  else
65.    {
66.      t=t→left→down
67.      i=i-1
68.      save[i]=t
69.    }
70.  }
71.  found_node=t
72.  New_node=create_node ( value)
73.  node1=COPY_NODE(head)
74.  If(k>temp)
75.    {
76.      //create new head and tail
77.      h1=createnode(∞)
78.      copy_node(h1)
79.      h1→left=null
80.      h1→right=new_node
81.      h1→down=head
82.      RELEASE_NODE(H1)
83.      t1=CreateNode(∞)
84.      COPY_NODE (t1)
85.      t1→left=new_node
Function insert_node( value: pointer to word)

86.   t1→right=NULL
87.   t1→down=tail
88.   RELEASE_NODE(t1)
89.   New_node→left=h1
90.   New_node→right=t1
91.   if((save[k-1]→right→down)==NULL OR(save[k-1]→right→value>new_node→value))
92.       New_node→down=save[k-1]→right
93.   if ((save[k+1]→right→down)==NULL OR (save[k+1]→right→value <new_node→value)) then
94.       save [k+1]→right→down =new_node
95.   new_node→down=null
96. }
97.   RELEASE_NODE(new_node)
98.   return true
99. }}

Figure 4.5 Lock free insertion

3.4 Delete_node

The delete operation uses search operation to locate the node with key k, and then uses two stage processes to perform the deletion. Firstly the node is logically deleted by marking the reference contained in it (delete_node→right→value).., secondly the node is physically deleted. The main steps of the algorithm for deleting a node at an arbitrary position are the following: (I) Set the deletion mark on the right pointer of the to-be-deleted node, (II) Set the deletion mark on the left pointer
of the to-be-deleted node, (III) Set the deletion mark on the down pointer of the to-be-deleted node, IV) Atomically update the right pointer of the previous node of the to-be-deleted node, (V) Atomically update the left pointer of the right node of the to-be deleted node, (VI) atomically update the down pointer for a node which was pointed by down pointer of to be deleted node. The figure 4.6 is representing the deletion of node from lock free concurrent MSL.

The algorithm first assume in (line 24) that it is not empty, if it is then function return null. To check whether the node to delete is already marked or not is done in(lines 28-29). If it is marked, it tries to update the right pointer of ‘prev’ node by calling help_del function. Once the node is successfully marked by the CAS operation in (line 32);it tries in (line 35) to update the left pointer of the ‘next’ node by calling update_prev function.

In the delete operations two procedure ( help_del, update_down) and function(update_prev) are called. The Help Delete sub-procedure set the deletion mark of the prev pointer and then atomically update the next pointer of the previous node of the to-be-deleted node.. The algorithm repeatedly tries in (lines 6-42) to delete the given marked node (node) by changing the next pointer from the previous non-marked node.

The purpose of update_prev function is to update the left pointer of a node, The algorithm repeatedly tries in lines The Help Insert sub-function, see Figure 6, tries to update the prev pointer of a(lines 15-39) node and then return a reference to a possibly direct previous node. given node (node), given a suggestion of a previous (not necessarily the directly previous) node (prev). Before trying to update the prev
pointer with the CAS

The procedure update_down is to update the down pointer of ode which are associated with ‘node’. if the ‘node’ is present at first level i.e. level 1, there is need to update the down pointer of node above the ‘node’ i.e. save[i+1] which pointing to need, now it starts pointing to the right of ‘node’. if ‘node’ is in-between there is need to update the node at adjacent levels.
Flow Chat (6-1) : Deletion in lock Free MSL
Flow Chat (6-2) : Deletion in lock Free MSL

1. If the node connected with del node is marked then
   - If IS_marked (Del_node→right) yes
     - Help_del (del_node)
     - The CAS is used to mark the del_node, if is successfully
       - Yes
       - Help_del (del_node)
       - The update_prev function is called for updating the previous node
       - Update_down is called to update the down field of node connected with del_node
       - back_off
       - STOP
Flow Chat (7) : Mark _Prev. for deletion in lock Free MSL
Flow Chat (8): Help_del for deletion in lock Free MSL
Flow Chat (9) : Update_down for deletion in lock Free MSL
Function delete_node (int key):boolean

{  
    node *delete_node,*prev,*suce,*up,*s[max]
    temp, i =current_level
    t=COPY_NODE (head)
    While (t! =NULL)
    {  
        while( t→key<key)
        {
            If ( IS_MARKED (t→right→value))
            t=Help_Del(t)
            save[i]=t
            t=READ_NODE(t→right)
        }  
        if (t→key==key)
        break
        else
        {  
            t=t→left→down
            i=i-1
            save[i]=t
        }  
    }  
    del_node=t
    while T do
    {  
        if (del_node==NULL) then
        RELEASE_NODE(del_node)
        return null
        link1=del_node→right
        if IS_MARKED(link1) then
        help_del(del_node)
30. RELEASE_NODE(node)
31. Continue
32. if CAS( &del_node->right,link1<link1.p,T) then
33. help_del(del_node)
34. next=READ_NODE(&del_node->right)
35. prev=update_prev(del_node,next)
36. RELEASE_NODE(prev)
37. release_node(next)
38. update_down(del_node)
39. break
40. RELEASE_NODE(del_node)
41. back-off
42. return
43. }

Figure: 4.6 Lock Free Deletion in Concurrent MSL
Procedure  Help_Del(node: pointer to Node)
{
    1.  Mark_Prev(node)
    2.  last=NULL
    3.  prev= READ_NODE(&node\rightarrow left)
    4.  next= READ_NODE (&node\rightarrow right)
    5.  while T do
    6.    if prev == next  then
    7.      break
    8.    if IS_MARKED(next\rightarrow right) then
    9.      Mark_Prev(next)
   10.     next:= READ_NODE (&next\rightarrow right)
   11.     RELEASE_NODE(next)
   12.     next=next2
   13.     continue
   14.     prev2= READ_NODE (&prev\rightarrow right)
   15.     if prev2 = NULL then
   16.        if last != NULL then
   17.          Mark_Prev(prev)
   18.         next2= READ_NODE (&prev\rightarrow right)
   19.         if CAS(&last\rightarrow right,<prev,F>,<next2,F>) then
   20.             RELEASE_NODE(prev)
   21.         else
   22.             RELEASE_NODE(next2)
   23.             RELEASE_NODE(prev)
   24.         prev=last
   25.        last=NULL
   26.     else
   27.        prev2=READ_NODE(&prev\rightarrow left)
   28.        RELEASE_NODE(prev)
29. prev=prev2
30. continue
31. if prev2 != node then
32. if last != NULL then
33. RELEASE_NODE(last)
34. last:=prev
35. prev=prev2
36. continue
37. RELEASE_NODE(prev2)
38. if CAS(&lprev->right, <node,F>,<next,F>) then
39. COPY_NODE(next)
40. RELEASE_NODE(node)
41. break
42. Back-Off
43. if last != NULL then RELEASE_node(last)
44. RELEASE_NODE(left)
45. RELEASE_NODE(next)
46. }

Figure: 4.7 help_del procedure in concurrent MSL
Function update_prev(prev, node: pointer to Node): pointer to Node

1. {
2.   last=null
3.   while T do
4.     prev2:=READ_NODE(&prev→right)
5.     if prev2 = null then
6.       if last != null then
7.         Mark_Prev(prev)
8.     next2:=READ_NODE(&prev→right)
9.     if CAS(&last→right,<prev,F>,<next2,F>) then
10.      RELEASE_NODE (prev)
11.    Else
12.      RELEASE_NODE (next2)
13.      RELEASE_NODE (prev)
14.      prev=last
15.      last=null
16.    else
17.      prev2=READ_NODE(&prev→left)
18.      RELEASE_NODE (prev)
19.      prev=prev2
20.      continue
21.      link1=node→left
22.      if IS_MARKED(link1) then
23.      RELEASE_NODE (prev2)
24.      break
25.    if prev2!= node then
26.      if last!= null then
27.      RELEASE_NODE (last)
28.      last=prev
29.      prev:=prev2
30. continue
31. RELEASE_NODE (prev2)
32. if link1->p = prev then
33. break
34. if prev->right = node and CAS(
35. &node->left,link1,<prev,F>) then
36. COPY_NODE(prev)
37. RELEASE_NODE (link1->p)
38. if IS_MARKED(prev->left) then break
39. Back-Off
40. if last != NULL then
41. RELEASE_NODE (last)
42. return prev
43. RELEASE_NODE (prev2)
44. if link1->p = prev then
45. break
46. if prev->right = node and CAS(
47. &node->left,link1,<prev,F>) then
48. COPY_NODE(prev)
49. RELEASE_NODE (link1->p)
50. if IS_MARKED(prev->left) then break
51. Back-Off
52. if last != NULL then
53. RELEASE_NODE (last)
54. return prev
55. }

Figure:4.8 update_prev function
Figure: 4.9 mark_prev procedure: to mark the node

Function update_down(node: pointer to Node)
{
    1. While(true) do
    2. {
    3.     if(i==1) // to update the down pointer of nodes associated with node, here i is the level of del_node
    4.         if(save[i+1]→down == node && (! IS_MARKED(save[i+1]))) then
    5.             save[i+1]→down = node→right
    6.         if IS_MARKED(save[i+1]) then
    7.             help_del(save[i+1])
    8.             save1 = READ_NODE(&save[i+1]→right)
    9.         else // the level of node to delete is more than 1
    10.            if(save[i+1]→down == node && (! IS_MARKED(save[i+1]))) then
    11.                save[i+1]→down = node→right
    12.                RELEASE_NODE(node→down)
    13.            }
    14.        }

Figure 4.10: update_down procedure: to update the down pointer of node associate with node to delete
4.4 Correctness

In this section we describe the correctness of presented algorithm. Here we outline a proof of linearizability [Herlihy[26]] and then we prove that algorithm is lock-free. Few definitions are required before giving proof of correctness.

**Definition 1** We denote with $M_t$ the abstract internal state of a modified skip list at the time $t$. $M_t$ is viewed as a list of values $(v_1, \ldots, v_n)$. The operations that can be performed on the modified skip list are Insert (I) and Delete(D). The time $t_1$ is defined as the time just before the atomic execution of the operation that we are looking at, and the time $t_2$ is defined as the time just after the atomic execution of the same operation. The return value of true$_2$ is returned by an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1; S_2$, where $S_1$ is the conditional state before the operation $O_1$, and $S_2$ is the resulting state after performing the corresponding operation:

\begin{align*}
M_{t_1} : I(v_1), & \quad M_{t_2} = M_{t_1} + [v_1] \quad (1) \\
M_{t_1} = : D() = \text{NULL} \quad , \quad M_{t_2} = \emptyset \quad (2) \\
M_{t_1}[v_1] + M_1 : D() = v_1, & \quad M_{t_2} = M_1 \quad (3)
\end{align*}

**Definition 2** In order for an implementation of a shared concurrent data object to be linearizable [10], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.
**Definition 3** The value \( v \) is present (\( i.M[i]=v \)) in the abstract internal state \( M \) of implementation, when there is a connected chain of next pointers (i.e. \( \text{prev} \rightarrow \text{link} \rightarrow \text{right} \)) from a present node in the doubly linked list that connects to a node that contains the value \( v \), and this node is not marked as deleted (i.e. \( \text{is}_\text{marked}(\text{node})=\text{false} \) ).

**Definition 4** The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub operations after the decision point, the operation will have the same result. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.

We will now use these definitions to show the execution history of point where the concurrent operation occurred atomically.

**Lemma 1**: A insert_node operation \((I(v))\), takes effect atomically at one statement.

Proof: The decision, state-read and state-change point for an insert operation which succeeds \((I(v))\), is when the CAS sub-operation \( \text{CAS}(&\text{prev} \rightarrow \text{right}, \text{next}, \text{new_node}) \) of insert operation succeeds. The state of the modified skip list was \((M_{t1} = M_1)\) directly before the passing of the decision point. The state of the modified skip list after passing the decision point will be \( MT_2 = [v] + M_1 \) as the next pointer of the save[k] node was changed to point to the new node which contains the value \( v \). Consequently, the linearizability point will be the CAS sub-operation in that line.

**Lemma 2**: A delete_node operation which fails \((D() =\text{NULL})\), takes effect atomically at one statement.
Proof: The decision point for a delete operation which tails (D() = NULL) is the check in line if (del_node==NULL) then . Passing of the decision point gives that the value v we are searching for deletion is not exist in modified skip list i.e (M_{t1} = NULL).

Lemma 3 : A delete_node operation which succeeds (D() = v), takes effect atomically at one statement.

Proof: The decision point for a delete operation which succeeds (D() = v) is when the CAS sub-operation inline [next=read_node (del_node→right)] succeeds. Passing of the decision point together with the verification in line [if is_marked(link1) then ]. Directly after passing the CAS sub-operation in [if CAS(&del_node→right, link1 <link1.p,T) then] (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the Modified skip list (¬∃ i.M_{t2} [i] = v). Unfortunately this does not match the semantic definition of the operation.

4.5 Conclusion

This chapter has introduced a non-blocking concurrent modified Skip list using a remarkably simple algorithm in a lock free environment. The lock free algorithm for searching, insertion and deletion on MSL are given. The insert algorithm considers each MSL chain as a single linked list. The nodes which have to delete are marked with the deletion marked on left and right pointer of a node. If a search operation while searching encounter a node whose deletion mark is set,it first suspend the search operation and try to delete the marked node,after that resume to previous operation. The presented algorithm is first step to lock free algorithmic implementation of modified skip list; it uses a fully described lock free memory management scheme.