Lock-Free Linked Lists and Skip Lists

Karan Agrawal
2023MCS2479

April 28, 2024

1 Introduction

- **Introduction**:
  - Mutual exclusion locks commonly used in distributed systems.
  - Lock-free implementations offer progress despite delays or failures.
  - Better performance due to simultaneous modifications by multiple processes.
- **Challenges with Existing Lock-Free Implementations**:
  - Early attempts suffered from inefficiency, low parallelism, excessive copying, and high overhead.
  - Original algorithms tailored to specific data structures often needed.
- **Lock-Free Linked Lists**:
  - Crucial as building blocks for other data structures.

  - New lock-free implementation with better average complexity presented.
  - Addressed lack of theoretical development in the area.

- **Lock-Free Skip Lists**:
  - Skip lists offer $O(\log n)$ expected time complexity for searches, insertions, and deletions.
  - Lock-free implementation based on linked list algorithms presented.

- **Implementation Details**:
  - Asynchronous shared-memory distributed system model considered.
  - Atomic single-word Compare&Swap synchronization primitives used.
  - Implementations are linearizable, ensuring correctness.

- **Analysis**:
  - **Nature of Lock-Free Implementations**: Lock-free implementations allow individual operations to take arbitrarily many steps, making it difficult to evaluate their worst-case cost. Therefore, it's more natural to analyze the average cost of operations, as it evaluates the performance of the system as a whole.
  - **Amortized Analysis Technique**: The analysis employs amortized analysis, which relies on a complex technique of billing part of the cost of each operation to concurrent operations that slow it down by modifying the data structure.

  - **Amortized Cost of an Operation**: The amortized cost of an operation S, denoted $\hat{t}(S)$, is calculated as the actual cost of S plus the total cost billed to S from other operations, minus the total cost billed from S to other operations.
  - **Cost Measurement**: The cost of operations is measured as a function of the size of the list and contention. The contention of an operation S, denoted $c(S)$, is defined as the maximum point contention during the execution of S.
  - **Bound on Amortized Cost**: It's proven that the amortized cost of an operation S is bounded by $O(n(S) + c(S))$, where $n(S)$ is the number of elements in the list when S is invoked, and $c(S)$ is the contention of S.
  - **Average Cost in an Execution**: For any execution E, the average cost of an operation in E is denoted $\bar{t}_E$ and is given by $O(\sum_{S \in E}(n(S) + c(S))/m_E)$, where m_E is the total number of operations invoked during E.
  - **Average Number of Elements and Contention**: The average number of elements in the list during an execution E is denoted $\overline{n}_E$ and is calculated as the sum of $n(S)$ over all operations in E, divided by m_E. Similarly, the average operation contention during E, denoted $\overline{c}_E$, is calculated similarly using $c(S)$.

- **Related Work**:
  - **Valois's Implementation**: Maintained auxiliary nodes between normal nodes to resolve interference between concurrent operations. Included backlink pointers in each node to backtrack through the list during concurrent deletions.
  - **Harris's Implementation**: Marked nodes before deletion to prevent concurrent operations from changing their right

pointers. Algorithms were simpler and generally performed better than Valois's.

- **Michael's Implementation**: Used Harris's design for the underlying data structure. Algorithms were compatible with efficient memory management techniques.
- **Combination of Techniques**: Linked lists combined techniques of marking nodes and using backlink pointers. Introduced new ideas such as flag bits to improve worst-case performance.
- **Comparison of Costs**: Average cost per operation in Valois's implementation could be high even when certain parameters were low. Average cost of operations in Harris's implementation could be worse than the presented implementation.
- **Pugh's Skip List**: Originally designed for sequential accesses. Lock-based concurrent implementations given by Pugh and Lotan-Shavit.
- **Lock-Free Skip Lists**: Sundell and Tsigas presented the first lock-free implementation, supporting various dictionary operations. Fraser presented another implementation using more powerful primitives.
- **Similarities and Differences**: All implementations used the marking technique for deletions in skip lists. Sundell and Tsigas's design allowed processes to overcome interference using backlink pointers. Presented design employed backlink pointers and flag bits for efficient recovery from interference.
- **Memory Management**: Sundell and Tsigas incorporated a reference counting scheme.
- **Fraser's Designs**: Presented skip list designs using more

powerful primitives like multi-word **CAS** and software transactional memory.

– **Experimental Results**: Suggested that lock-free implementations could be a practical alternative to lock-based ones.

2 Paper Summary

Algorithm 1: CAS (Compare-and-Swap)

```
Data: Pointer address, Word old_val, Word new_val
Result: Boolean indicating success of Compare-and-Swap
        operation
value ← *address;
if value == old_val then
    *address ← new_val;
    return true;
else
    return false;
end
```

2.1 Lock-Free Linked Lists

2.1.1 Lock-Free Linked Lists Implementation

- **Problem with Lock-Free Linked Lists:** Ensuring consistency during deletion operations is crucial. When a process tries to delete a node, it must ensure that the node's right pointer remains unchanged by concurrent operations to avoid incorrect executions.

- **Harris's Technique:** Harris introduced a technique where the right pointer of each node is replaced by a composite field called a successor field, which includes a right pointer and a mark bit. This mark bit acts as a toggle to control when the right pointer can be changed. Deletion involves marking the successor field of the node and then physically removing the node from the list.

- **Performance Issues with Harris's Approach:** Harris's approach suffers from performance issues, especially in scenarios with concurrent operations. If a process fails during an operation, it needs to restart from the beginning of the list, leading to poor performance in certain scenarios.

- **Improvements in the Proposed Implementation:** The proposed implementation introduces backlink pointers and flag bits to improve worst-case performance. Backlink pointers help processes recover from failures by traversing back to the nearest unmarked node. Flag bits act as warnings for ongoing deletions, preventing nodes from being marked or changed until the flag is removed.

- **Helping Mechanism:** Processes are allowed to help each other with deletions to maintain lock-freedom. If a process encounters a flagged node hindering its operation, it attempts to complete the corresponding deletion to remove the flag before continuing with its operation.

2.1.2 Algorithms

- **Node Structure**: Each node in the linked list contains fields for key, element, backlink, and successor. The successor field consists of a right pointer, a mark bit, and a flag bit.

- **Head and Tail Nodes**: The head and tail nodes of the list contain dummy keys, referenced by the shared variables `head` and `tail` respectively.

- **Search Operation**: The `Search` routine locates a node with a specified key in the list using the `SearchFrom` routine. It returns pointers to two nodes satisfying the condition $n1.key < k \leq n2.key$.

Algorithm 2: Search(Key k) : Node

```
Input   : Key $k$
Output: Current node (currNode)
(currNode, nextNode) ← SearchFrom($k$, head);
if currNode.key == $k$ then
    return currNode;
end
else
    return NO_SUCH_KEY;
end
```

Algorithm 3: SearchFrom(Key k, Node *currNode): (Node, Node)

```
Input   : Key $k$, Current node (currNode)
Output: Current node (currNode), Next node (nextNode)
nextNode ← currNode.right;
while nextNode.key ≤ $k$ do
    while nextNode.mark == 1 and (currNode.mark == 0
     or currNode.right ≠ nextNode) do
        if currNode.right == nextNode then
            HelpMarked(currNode, nextNode);
        end
        nextNode ← currNode.right;
    end
    if nextNode.key ≤ $k$ then
        currNode ← nextNode;
        nextNode ← currNode.right;
    end
end
return (currNode, nextNode);
```

- **Insert Operation**: The `Insert` routine finds the insertion position using `SearchFrom`, verifies the key is not a duplicate, and attempts to insert the new node between `prevNode` and `nextNode` using a `CAS` operation. It handles failures due to changes in the predecessor node's successor field by recovering from flags or marks.

Algorithm 4: Insert(Key k, Element e) : Node

```
Input  : Key $k$, Element $e$
Output: New node if insertion is successful, error code
        otherwise
(prevNode, nextNode) ← SearchFrom($k$, head);
if prevNode.key == $k$ then
    return DUPLICATE_KEY;
end
newNode ← new Node(key = $k$, element = $e$);
while true do
    prevSucc ← prevNode.succ;
    if prevSucc.flag == 1 then
        HelpFlagged(prevNode, prevSucc.right);
    end
    else
        newNode.succ ← (nextNode, 0, 0);
        result ←
         C&S(prevSucc, (nextNode, 0, 0), (newNode, 0, 0));
        if result == (nextNode, 0, 0) then
            return newNode;
        end
        if result == $(*, 0, 1)$ then
            HelpFlagged(prevNode, result.right);
        end
        while prevNode.mark == 1 do
            prevNode ← prevNode.backlink;
        end
        (prevNode, nextNode) ← SearchFrom($k$, prevNode);
        if prevNode.key == $k$ then
            free newNode;
            return DUPLICATE_KEY;
        end
    end
end
```

- **Delete Operation**: The `Delete` routine performs a three-step deletion of the node. It calls `SearchFrom` to locate the node, then calls `TryFlag` to flag the predecessor node. If successful, it proceeds with marking and physically deleting the node using `TryMark` and `HelpMarked`. It handles cases where the node is deleted or another process flags the predecessor node.

Algorithm 5: Delete(Key k) : Node

```
Input  : Key $k$
Output: Deleted node (delNode)
(prevNode, delNode) ← SearchFrom($k - \epsilon$, head);
if delNode.key ≠ $k$ then
    return NO_SUCH_KEY;
end
(prevNode, result) ← TryFlag(prevNode, delNode);
if prevNode ≠ null then
    HelpFlagged(prevNode, delNode);
end
if not result then
    return NO_SUCH_KEY;
end
return delNode;
```

Algorithm 6: HelpFlagged(Node *prevNode, Node *delNode)

```
Input  : Pointer to flagged node's predecessor (prevNode),
         Pointer to flagged node (delNode)
delNode.backlink ← prevNode;
if delNode.mark == 0 then
    TryMark(delNode);
end
HelpMarked(prevNode, delNode);
```

Algorithm 7: TryMark(Node delNode)

```
Input   : Node to be marked (delNode)
while true do
    nextNode ← delNode.right;
    result ←
     C&S(delNode.succ, (nextNode, 0, 0), (nextNode, 1, 0));
    if result == (∗, 0, 1) then
        HelpFlagged(delNode, result.right);
    end
    if delNode.mark == 1 then
        break;
    end
end
```

Algorithm 8: TryFlag(Node *prevNode, Node *targetNode): (Node, Boolean)

Input : Pointer to predecessor node (`prevNode`), Pointer to target node (`targetNode`)

Output: Pointer to predecessor node (`prevNode`), Boolean result (`result`)

```
while true do
    if prevNode.succ == (targetNode, 0, 1) then
        return (prevNode, false);
    end
    result ←
     C&S(prevNode.succ, (targetNode, 0, 0), (targetNode, 0, 1));

    if result == (targetNode, 0, 0) then
        return (prevNode, true);
    end
    if result == (targetNode, 0, 1) then
        return (prevNode, false);
    end
    while prevNode.mark == 1 do
        prevNode ← prevNode.backlink;
    end
    (prevNode, targetNode) ←
     SearchFrom(targetNode.key − $\epsilon$, prevNode);
    if delNode ≠ targetNode then
        return (null, false);
    end
end
```

Algorithm 9: HelpMarked(Node *prevNode, Node *delNode)

```
Input : Pointer to flagged node's predecessor (prevNode),
        Pointer to flagged node (delNode)
delNode.backlink ← prevNode;
if delNode.mark == 0 then
    TryMark(delNode);
end
HelpMarked(prevNode, delNode);
```

2.1.3 Correctness Proof Overview

- **Invariants Classification**
  Nodes in the data structure are classified into three categories:
  - **Keys are Strictly Sorted (Inv 1):**
    This invariant ensures that the keys of the nodes in the linked list are strictly ordered. In other words, for any two nodes n_1 and n_2, if n_1.right = n_2, then n_1.key ¡ n_2.key.
  - **Linked List Structure (Inv 2):**
    This invariant ensures that the regular and logically deleted nodes form a proper linked list structure. Each node (except head and tail) has exactly one predecessor and one successor. The head node has no predecessor, and the tail node has no successor.
  - **Predecessor Flagging (Inv 3):**
    This invariant ensures that for any logically deleted node, its predecessor is flagged (and unmarked), and its successor is not marked. In other words, if a node n is logically deleted and m is its predecessor, then $m.succ = (n, 0, 1)$ and $(n.right).mark = 0$.

  - **Backlink Stability (Inv 4):**
    This invariant ensures that for any logically deleted node, its backlink points to its predecessor. If a node n is logically deleted and m is its predecessor, then $n.backlink = m$.
  - **No Concurrent Marking and Flagging (Inv 5):**
    This invariant ensures that no node can be both marked and flagged at the same time. This prevents potential inconsistencies that could arise from concurrent marking and flagging operations.

  - Regular: Nodes that are inserted into the list and remain unmarked.
  - Logically Deleted: Nodes marked for deletion but still linked to a regular node.
  - Physically Deleted: Nodes marked for deletion with no regular node linked to them.

- **Invariant Proofs**

  - **Inv 1-3:** Proven through induction on the number of successful Compare-and-Swap (C&S) operations. These invariants ensure key ordering and maintain the linked list structure.
  - **Inv 4:** Demonstrated by induction, showing that a node's backlink, once set, remains unchanged.
  - **Inv 5:** Trivially holds since no node can be both marked and flagged simultaneously.

- **Backlink Stability**
  Demonstrated that a node's backlink, once set, remains constant.

- **Deletion Steps**
  Establishes the three-step deletion process: flagging the predecessor, marking the node, and physically deleting it.

- **SearchFrom Postconditions**
  Defines conditions ensuring consistency during searches, guaranteeing correct node identification and marking.

- **Linearization of Operations**
  Operations are linearized to ensure consistency with the state of the data structure.

  - **Insert:** Linearized when a C&S operation successfully inserts a new node.
  - **Delete:** Linearized after marking the node for deletion.

- **Key Points**

  - **Consistency:** The proof ensures that the data structure maintains consistency during operations, preserving key ordering and linked list structure.
  - **Linearization:** Operations are linearized to reflect their impact on the state of the data structure, ensuring correct behavior under concurrency.
  - **Invariants:** Invariants play a crucial role in reasoning about the correctness of the data structure, providing properties that must hold true at all times.

2.1.4 Performance Analysis of Linked List Data Structure

1. **Billing Scheme**:

- Focuses on essential steps such as C&S attempts, backlink pointer traversals, and pointer updates during searches.
- Categorizes steps into successful C&S's, necessary steps, and extra steps due to concurrent operations.

2. **Amortized Cost Definition**:

   - Defines the amortized cost of successful C&S's as the actual cost plus total billed cost.
   - Amortized cost of an operation is calculated by subtracting billed cost from the operation and adding billed cost from successful C&S's part of the operation.
   - We define the amortized cost of a successful C&S C, denoted $\hat{t}(C)$, to be the sum of its actual cost and the total cost billed to C. Note that the actual cost of a successful C&S is 1.

   $$\hat{t}(C) = (\text{actual cost of } C) + (\text{total cost billed to } C) \quad (1)$$

   - We define the amortized cost of operation S, denoted $\hat{t}(S)$, to be the actual cost of S minus the total cost billed from S to successful C&S's plus the total cost billed to successful C&S's that are part of S. The second term represents the extra cost of S.

   $$\hat{t}(S) = ((\text{necessary cost of } S) + (\text{extra cost of } S) + (\text{cost of successful C\&S} \quad (2)$$

   $$- ((\text{extra cost of } S) + (\text{total cost billed to successful C\&S's that ar} \quad (3)$$

   $$= (\text{necessary cost of } S) + (\text{cost of successful C\&S's performed by } S) \quad (4)$$

   $$+ (\text{total cost billed to successful C\&S's that are part of } S) \quad (5)$$

3. **Billing Mapping (Function β)**:

   - Let Q be the set of essential steps in the entire execution E. Function β maps Q to itself. If some operation S performs step $s \in Q$, β maps this step either to itself or to a successful C&S that is part of another operation as described below.
     - **C&S's**:
       - If a C&S C on the successor field of node n was executed:
         - If C is successful, then it is mapped to itself.
         - If C fails and it is not of the fourth type, it is mapped to the C&S that last modified $n.succ$.
         - If C is of the fourth type and it fails, it is mapped to the C&S that physically deleted the node that C was trying to delete.
     - **Backlink Traversals**:
       - A backlink pointer traversal from node n to node m is mapped to the C&S that marked node n.
     - **Next Node Pointer Updates**:
       - If the update changes next node from m to m':
         - If m is physically deleted before the update, it is mapped to the C&S that physically deleted m.
         - Otherwise, it is mapped to the C&S that inserted m'.
     - **Curr Node Pointer Updates**:
       - If the update sets curr node pointer to node n:
         - If n was inserted into the list after operation S was invoked, the update is mapped to the C&S that inserted n.
         - Otherwise, the update is mapped to itself.

4. **Proof of Amortized Cost Bound**:
   (a) **Bound Proof Objective**:
   - Aim to show that amortized cost of each C&S operation within S is bounded by O(c(S)), where c(S) represents concurrency overhead.

   (b) **Key Properties Proved**:
   - For each step type:
     i. No other steps of the same type within S are mapped to the same successful C&S.
     ii. The successful C&S was indeed performed during the execution of S.

   (c) **Proof Strategies**:
   - Direct proofs for next node updates and curr node pointer updates.
   - Backlink traversals rely on the non-growth property of backlink chains towards the right.
   - Unsuccessful C&S operations' proofs involve lemmas showing existence of opportune moments for success.

   (d) **Bounding Amortized Cost**:
   - Due to proved properties, no more than c(S) steps of each type can be mapped to any successful C&S within S.
   - Hence, the amortized cost of a successful C&S operation is O(c(S)).
   - As at most three successful C&S operations can be part of S, the amortized cost of all successful C&S operations in S is O(c(S)).
   - The overall amortized cost of S is thus O(n(S)) + O(c(S)), where n(S) represents necessary cost.

5. **Conclusion**:

    - Overall amortized cost of an operation is $O(n(S)) + O(c(S))$, representing necessary cost and concurrency overhead respectively.

2.2 Lock-Free Skip Lists

2.2.1 High-level Description

- **Skip List**
    - Skip list: dictionary storing nodes on multiple levels, similar to linked lists.
    - Supports efficient searches, insertions, and deletions.
    - Provides probabilistic balancing with expected operation cost of $O(\log(n))$.
- **Skip List Design**
    - Nodes interconnected on each level like linked lists, with random forward pointers.
    - Probability distribution of pointers approximates balanced structure.
    - Efficient search with exponentially decreasing probability of high-cost operations.
- **Lock-Free Skip List Design**
    - Similar to original skip list design with minor cosmetic differences for integration with previous algorithms.
    - Nodes organized into towers, with root nodes as representatives.

- Head and tail towers created during initialization, not modified afterward.

- **Implementation Issues**
  - Each level viewed as linked list, allowing reuse of previous algorithms for insertions and deletions.
  - Deletions marked at root node, with superfluous nodes physically deleted.
  - Search problem due to presence of superfluous nodes, solved by checking node status before traversal.
  - Maintaining pointer to starting node for searches can be complex, addressed by starting search from bottom of head tower.

2.2.2 Algorithms

Algorithm 10: Search_SL(Key k): RNode

```
Input  : Key $k$
Output: RNode
$(curr\_node, next\_node)$ = SearchToLevel_SL($k$, 1);
if ($curr\_node.key == k$) then
    return $curr\_node$;
end
else
    return $NO\_SUCH\_KEY$;
end
```

Algorithm 11: SearchToLevel_SL(Key k, Level v): (Node, Node)

```
Input   : Key $k$, Level $v$
Output: (Node, Node)
$(curr\_node, curr\_v)$ = FindStart_SL($v$);
while $(curr\_v > v)$ do
    $(curr\_node, next\_node)$ = SearchRight($k, curr\_node$);
    $curr\_node = curr\_node.down$;
    $curr\_v - -$;
    $(curr\_node, next\_node)$ = SearchRight($k, curr\_node$);
end
return $(curr\_node, next\_node)$;
```

Algorithm 12: FindStart_SL(Level v): (Node, Level)

```
Input   : Level $v$
Output: (Node, Level)
$curr\_node = head$;
$curr\_v = 1$;
while $((curr\_node.up.right.key \neq +\infty)$ or $(curr\_v < v))$ do
    $curr\_node = curr\_node.up$;
    $curr\_v + +$;
end
return $(curr\_node, curr\_v)$;
```

Algorithm 13: SearchRight(Key k, Node* $curr\_node$): (Node, Node)

```
Input  : Key $k$, Node* $curr\_node$
Output: (Node, Node)
$next\_node = curr\_node.right$;
while ($next\_node.key \leq k$) do
    while ($next\_node.tower\_root.mark == 1$) do
        $(curr\_node, status, result) =$
         TryFlagNode($curr\_node, next\_node$);
        if ($status == IN$) then
            HelpFlagged($curr\_node, next\_node$);
        end
        $next\_node = curr\_node.right$;
    end
    if ($next\_node.key \leq k$) then
        $curr\_node = next\_node$;
        $next\_node = curr\_node.right$;
    end
end
return $(curr\_node, next\_node)$;
```

Algorithm 14: TryFlagNode(Node* *prev_node*, Node* *target_node*): (Node, status, result)

```
Input  : Node* prev_node, Node* target_node
Output: (Node, status, result)
while loop do
    if ($prev\_node.succ == (target\_node, 0, 1)$) then
        return ($prev\_node, IN, false$);
    end
    $result =$
      c&s($prev\_node.succ, (target\_node, 0, 0), (target\_node, 0, 1)$);

    if ($result == (target\_node, 0, 0)$) then
        return ($prev\_node, IN, true$);
    end
    // Failure
    if ($result == (target\_node, 0, 1)$) then
        return ($prev\_node, IN, false$);
    end
    while ($prev\_node.mark == 1$) do
        $prev\_node = prev\_node.back\_link$;
    end
    $(prev\_node, del\_node) =$
      SearchRight($target\_node.key - \varepsilon, prev\_node$);
    if ($del\_node \neq target\_node$) then
        return ($prev\_node, DELETED, false$);
    end
end
```

Algorithm 15: Insert_SL(Key k, Elem e): RNode

```
Input  : Key $k$, Elem $e$
Output: RNode
($prev\_node, next\_node$) = SearchToLevel_SL($k$, 1);
if ($prev\_node.key == k$) then
    return $DUPLICATE\_KEY$;
end
$newRNode$ = new
  $rnode(key = k, elem = e, down = null, tower\_root = self)$;
$newNode = newRNode$;
$tH = 1$;
while ((FlipCoin() == $head$) and ($tH \leq maxLevel - 1$)) do
    $tH + +$;
end
$curr\_v = 1$;
while loop do
    ($prev\_node, result$) =
      InsertNode($newNode, prev\_node, next\_node$);
    if (($result == DUPLICATE\_KEY$) and ($curr\_v == 1$))
      then
        free newNode;
        return $DUPLICATE\_KEY$;
    end
    if ($newRNode.mark == 1$) then
        if (($result == newNode$) and ($newNode \neq$
          $newRNode$)) then
            DeleteNode (prev_node, newNode);
            return $newRNode$;
        end
    end
    $curr\_v + +$;
    if ($curr\_v = tH + 1$) then
        return $newRNode$;
    end
    $lastNode = newNode$;
    $newNode$ = new $node(key = k, down =$
      $lastNode, tower\_root = newRNode)$;
    ($prev\_node, next\_node$) = SearchToLevel_SL($k, curr\_v$);
end
```

Algorithm 16: InsertNode(Node* *newNode*, Node* *prev_node*, Node* *next_node*): (Node, Node)

```
Input  : Node* $newNode$, Node* $prev\_node$, Node*
         $next\_node$
Output: (Node, Node)
if ($prev\_node.key == newNode.key$) then
    return ($prev\_node, DUPLICATE\_KEY$);
end
while loop do
    $prev\_succ = prev\_node.succ$;
    if ($prev\_succ.flag == 1$) then
        HelpFlagged($prev\_node, prev\_succ.right$);
    end
    else
        $newNode.succ = (next\_node, 0, 0)$;
        $result =$
         c&s($prev\_node.succ, (next\_node, 0, 0), (newNode, 0, 0)$);

        if ($result == (newNode, 0, 0)$) then
            return ($prev\_node, newNode$);
        end
        else if ($result == (*, 0, 1)$) then
            HelpFlagged($prev\_node, result.right$);
        end
        while ($prev\_node.mark == 1$) do
            $prev\_node = prev\_node.back\_link$;
        end
        ($prev\_node, next\_node$) =
         SearchRight($newNode.key, prev\_node$);
        if ($prev\_node.key == newNode.key$) then
            return ($prev\_node, DUPLICATE\_KEY$);
        end
    end
end
```

Algorithm 17: Delete_SL(Key k): RNode

```
Input  : Key $k$
Output: RNode
$(prev\_node, del\_node)$ = SearchToLevel_SL($k - \varepsilon, 1$);
if ($del\_node.key \neq k$) then
    return $NO\_SUCH\_KEY$;
end
$result$ = DeleteNode($prev\_node, del\_node$);
if ($result == NO\_SUCH\_NODE$) then
    return $NO\_SUCH\_KEY$;
end
SearchToLevel_SL($k, 2$);
return $del\_node$;
```

Algorithm 18: DeleteNode(Node* $prev\_node$, Node* del_node): Node

```
Input  : Node* $prev\_node$, Node* $del\_node$
Output: Node
$(prev\_node, status, result)$ =
 TryFlagNode($prev\_node, del\_node$);
if ($status == IN$) then
    HelpFlagged($prev\_node, del\_node$);
end
if ($result == false$) then
    return $NO\_SUCH\_NODE$;
end
return $del\_node$;
```

Algorithm 19: HelpMarked(Node* *prev_node*, Node* *del_node*)

```
Input  : Node* $prev\_node$, Node* $del\_node$
Output:
$next\_node = del\_node.right$;
c&s($prev\_node.succ, (del\_node, 0, 1), (next\_node, 0, 0)$);
```

Algorithm 20: HelpFlagged(Node* *prev_node*, Node* *del_node*)

```
Input  : Node* $prev\_node$, Node* $del\_node$
Output:
$del\_node.back\_link = prev\_node$;
if ($del\_node.mark == 0$) then
    TryMark($del\_node$);
end
HelpMarked($prev\_node, del\_node$);
```

Algorithm 21: TryMark(Node* *del_node*)

```
Input  : Node* $del\_node$
Output:
repeat
    $next\_node = del\_node.right$;
    $result =$
      c&s($del\_node.succ, (next\_node, 0, 0), (next\_node, 1, 0)$);
    if ($result == (*, 0, 1)$) then
        HelpFlagged($del\_node, result.right$);
    end
until ($del\_node.mark == 1$);
```

2.2.3 Lock-Freedom and Performance

- Lock-Freedom

1. **Lemma 109:** Suppose process P is executing a SearchRight routine SR. Then after P performs some finite number of steps, P will complete SR, or P will call a TryFlagNode routine in line 4, or there will be a successful C&S performed by some process.

   Proof:

   If SR enters the loop 3-7, it calls TryFlagNode. If it does not enter this loop, then each iteration of the loop in lines 2-10 costs $O(1)$, and in each iteration lines 9 and 10 are executed...

2. **Lemma 110:** Suppose process P is executing a TryFlagNode routine TFN. Then after P performs some finite number of steps, P will complete TFN, or P will call a SearchRight routine in line 11, or there will be a successful C&S performed by some process.

   Proof:

   If no C&S operations are performed, no new nodes get inserted into the list, and therefore completing the loop in lines 9-10 takes finite time...

- Fine-Tuning Performance and Space Requirements

  1. **Optimizing Insertions:** In our implementation, when an insertion constructs a tower, it performs a separate search each time it needs to insert a new node into that tower...
  2. **Optimizing Deletions:** A deletion does not need to perform a complete search to delete the rest of the nodes of a tower after it deletes the tower's root node...
  3. **Space Efficiency:** For more space efficiency, we can use the same construction as Pugh (see Subsection 4.1.1) for our skip lists...

3 Conclusion

1. **Algorithm Description**:
    - Presented new algorithms for lock-free linked lists.
    - Proved average cost of operations is linear in list length plus contention, regardless of operation sequence or scheduling.
2. **Billing Technique**:
    - Used billing technique applicable to distributed data structures for analysis.
3. **Modular Use**:
    - Demonstrated modularity by using linked list algorithms as basis for lock-free skip list implementation.
4. **Memory Management**:
    - Suggested Valois's reference counting method for memory management, applicable due to absence of cycles among physically deleted nodes.
5. **Future Directions**:
    - Open problem: Obtaining good bound on average expected complexity of lock-free implementations of skip lists (or dictionary data structures).
    - Implementation and analysis technique may aid in addressing this challenge.
    - Challenges remain, such as handling adversaries potentially deleting tall towers used for quick traversal.
6. **Alternative Analysis**:

- Interest in developing practical alternative to worst-case amortized analysis for lock-free data structures.
- Feasible amortized analysis method that bounds average complexity over possible schedules would be valuable.

7. **Acknowledgements**:

- Funding from the Natural Sciences and Engineering Research Council of Canada and Ontario Graduate Scholarship.
- Thanks to Hakan Sundell, Philippas Tsigas, and anonymous referees for contributions and feedback.