School of Computer Science and Technology
Zhejiang University

浙江大學

Skip List

FIRST LAST
FIRST LAST
FIRST LAST

2020-05-20

Contents

1. Introduction

Skip list is a data structure that allows $\mathcal{O}(\log n)$ search complexity as well as $\mathcal{O}(\log n)$ insertion complexity within an ordered sequence of n elements. Thus it can get the best features of an array while maintaining a linked list-like structure that allows insertion, which is not possible in an array.

Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one.

Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than or equal to the element searched for. Via the linked hierarchy, these two elements link to elements of the next sparsest subsequence, where searching is continued until finally we are searching in the full sequence.

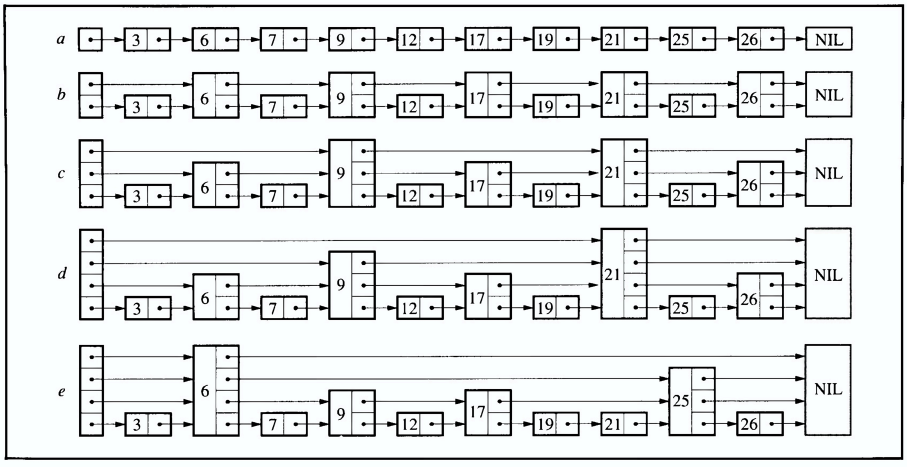

Figure 1.1: Linked Lists with Additional Pointers

2. Algorithm Specification

2.1 Data Structures

2.1.1 Node

To implement a skiplist, we should first define a Node class, this class contains the following elements:

1. key: the key of a node
2. value: the data of the node
3. level: A level i node carries i forward pointers indexed through 0 to i.
4. forward: store all pointers

```
1  class Node
2  {
3  private:
4
5      int key;
6
7      int value;
8
9      unsigned int level;
10
11     Node **forward;
12
13     friend class SkipList;
14
15 public:
16     Node(int key, int value, unsigned int level);
17
18     Node();
19 };
```

2.1.2 SkipList

To implement skip list, we define a class which contains all

1. maxLevel: Maximum level for this skip list
2. currentLevel: current level of skip list
3. header: pointer to header node

4. p: probability

1. `RandomLevel()`: Generate a random level
2. `InsertKey(int key, int value)`: Insert a value in key
3. `FindKey(int key)`:

```
1  class SkipList
2  {
3  private:
4
5      unsigned int maxLevel;
6
7      unsigned int currentLevel;
8
9      Node *header;
10
11     float p;
12
13     [[nodiscard]] unsigned int RandomLevel() const;
14
15 public:
16     explicit SkipList(int maxLevel = 3, float p = 0.5);
17
18     bool InsertKey(int key, int value);
19
20     bool FindKey(int key);
21
22     bool DeleteKey(int key);
23
24     void PrintSkipList();
25 };
```

2.2 Main Operations

2.2.1 Search

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element(it it is in the list).

2.2.2 Insert

2.2.2.1 Generate Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level $i+1$ pointers. To get away from magic constants, we say that a fraction p of the nodes with level i pointers also have level $i+1$ pointers. Levels are generated randomly by an algorithm, without reeference to the number of elements in the list.

Algorithm 1 Search(list, searchKey)

Input: Slip list, search key
Output: required value

```
1: x:=list→header
2: –loop invariant: x→key
3: for i:=list→level downto 1 do
4:     while x→forward[i]→key < searchKey do
5:         i:=list→forward[i]
6:     end while
7: end for
8: x:=x→forward[1]
9: if x→key = searchKey then
10:    return x→value
11: end if
12: return failure
```

Algorithm 2 RandomLevel()

Input:
Output:

```
1: newLevel:=1
2: –random()returns a random value in [0, 1)
3: while random() < p do
4:     newLevel:=newLevel + 1
5: end while
6: return min(newLevel, MaxLevel)
```

2.2.2.2 Insert Value

To insert or delete a node, we simply search and splice. A vector *update* is maintained sp that when the search is compelete, $update[i]$ contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insert/delete.

If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check to see if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

2.2.3 Delete

2.3 Main Function

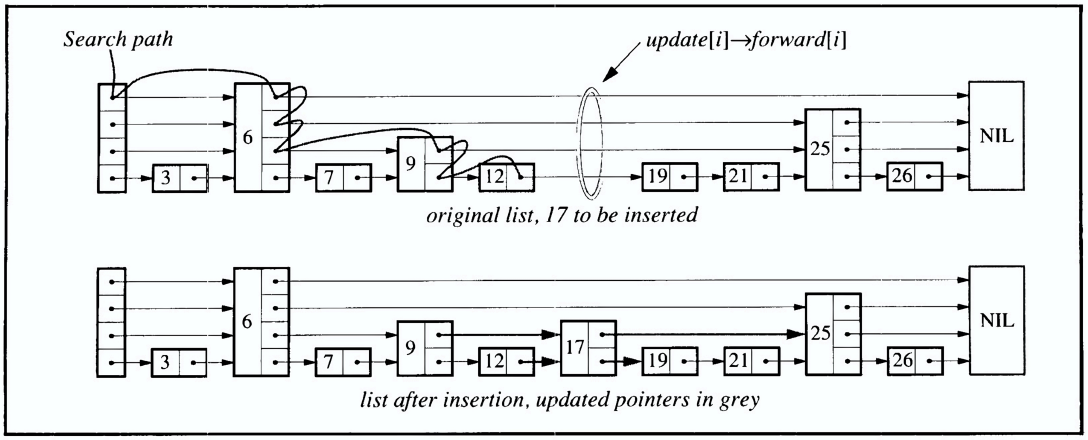

Figure 2.1: Pictorial Description of Steps Involved in Performing an Insertion

Algorithm 3 Insert(list, searchKey, newValue)

Input:
Output:

```
 1: local update[1...MaxLevel]
 2: for  i:=list→level downto 1 do
 3:     while x→forward[i]→key < searchKey do
 4:         x:=x→forward[i]
 5:         –x→key < searchKey ≤ x→forward[1]→key
 6:     end while
 7:     update[i]:=x
 8: end for
 9: x:=x→forward[1]
10: if  x→key = searchKey  then
11:     x→value:=newValue
12: else
13:     newLevel:=RandomLevel()
14:     if newLevel > list→level  then
15:         for i:=list→level+1 to newLevel do
16:             update[i]:=list→header
17:         end for
18:         list→level:=newLevel
19:     end if
20:     x:=makeNode(newLevel, searchKey, value)
21:     for i:=1 to newLevel do
22:         x→forward[i]:=update[i]→forward[i]
23:         update[i]→forward[i]:=x
24:     end for
25: end if
```

Algorithm 4 Delete(list, searchKey, newValue)

Input:
Output:

```
1: local update[1...MaxLevel]
2: x:=list→header
3: for  i:=list→level downto 1 do
4:     while x→forward[i]→key < searchKey do
5:         x:=x→forward[i]
6:     end while
7:     update[i]:=x
8: end for
9: x:=x→forward[1]
10: if  x→key = searchKey  then
11:     for i:=1 to list→level do
12:         if update[i]→forward[i] ≠ x then
13:             break
14:         end if
15:         update[i]→forward[i]:=x→forward[i]
16:     end for
17:     free(x)
18:     while list→level > 1 and list→header→forward[list→level] = NIL do
19:         list→level:=list→level-1
20:     end while
21: end if
```

3. Testing Result

4. Complexity Analysis

The time required to excute the `Search`, `Delete` and `Insert` operation is domained by the time required to search for the appropriate element. For the `Insert` and `Delete` operations, there is an additional cost proporational to the level of the node being inserted or deleted. The time required to find an element is proportional to the length of the search path, which is determined by the pattern in which elements with different levels appears as we traverse the list.

4.1 Probablistic Philosophy

The structure of a skip list is determined only by the number of elements in the skip list and the results of consulting the random number generator. The sequence of operations that produced the current skip list does not matter.

We assume an adversarial user does not have access to the levels of nodes; otherwise, he could create situations with worst-case running times by deleting all nodes that were not level 1.

The probabilities of poor running times for successive operations on the same data structure are NOT independent; two successive searches for the same element will both take exactly the same time.

4.2 Time Complextiy

4.2.1 Definition

The height of the PSL is expected to be about log1/P n. Since, among all elements that made it to a certain level, about every (1/P)th element will make it to the next higher level, one should expect to make 1/p key comparisons per level. Therefore, one should expect about 1/p*log1/p n key comparisons in total, when searching for +. As it will turn out (Theorem 3.3), this is exactly the leading term in the search cost for $+\infty$ in a PSL of n keys.

4.2.2 Search Cost

We divide the steps on the search path into vertical steps (that is, one less than T_n) and horizontal steps(L_{m-1} : number of full horizontal steps on the search path for the $(m-1)^{st}$ key)

$$E\left(C_n^{(m)}\right) = E\left(T_n\right) + E\left(L_{m-1}\right)$$

$E(T_n)$: random variables following the geometric distribution

$$E\left(T_n\right) = \sum_{k\geq 1}\left\{k\left(\left(1-p^k\right)^n - \left(1-p^{k-1}\right)^n\right)\right\} = \sum_{k\geq 1}\left\{1-\left(1-p^{k-1}\right)^n\right\}$$

$E(L_{m-1})$

$$E\left(L_{m-1}\right) = \begin{cases} 1 + \frac{q}{p} \sum_{j=1}^{m-2} \sum_{k \geq 1} p^k \left(1 - p^k\right)^j & \text{if } m = 2, 3, \ldots, n+1 \\ 0 & \text{if } m = 1 \end{cases}$$

At any particular point in the climb, we are at a situation similar to situation a . We are at the ith forward pointer of a node x and we have no knowledge about the levels of nodes to the left of x or about the level of x, other than that the level of x must be at least i. Assume the x is not the header (this is equivalent to assuming the list extends infinitely to the left). If the level of x is equal to i, then we are in situation b. If the level of x is greater than i, then we are in situation c. The probability that we are in situation c is p. Each time we are in situation c, we climb up a level. We use C(k) denotes the expected cost (length) of a search path that climbs up k levels.

An upper bound of C(k) = (L(n) - 1)/p. Because L(n) = log1/p n and p = 1/2, we can get the conclusion that C(k) = 2 * log2 n - 2 = O(log2 n)

4.2.3 Insert/Delete Cost

The Insert and Delete Algorithms are built on the Search Algorithm. After finding the position needed to insert or delete a node, the time complexity of other steps are smaller than $\mathcal{O}(\log n)$. So we can make a conclusion that the Insert and Delete Algorithms are both $\mathcal{O}(\log n)$.

4.3 Space Complexity

Every time a number is inserted, the program will randomly assign a height for node to storage pointer (less than MaxHeight), so its Space Complexity is $\mathcal{O}(n)$

Bibliography

Appendix A: Souce Code

main.cpp

SkipList.hpp

SkipList.cpp

Declare

We hereby declare that all the work done in this project titled "Skip List" is of our independent effort as a group.