Skip Lists

Group 18

Date:2020-06-01

CONTENT

Chapter 1 Introduction

1.1 Background

The **skip list** is a **probabilisitc data structure** that is built upon the general idea of a **linked list**. The skip list uses probability to build subsequent layers of linked lists upon an original linked list. Each additional layer of links contains fewer elements, but no new elements.

A skip list allows $O(log\,N)$ search complexity as well as $O(log\,N)$ insertion complexity within an ordered sequence of n elements. Thus it can get the best features of an **array** (for searching) 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 (see the picture below on the right). 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. The elements that are skipped over may be chosen probabilistically or deterministically, with the former being more common.

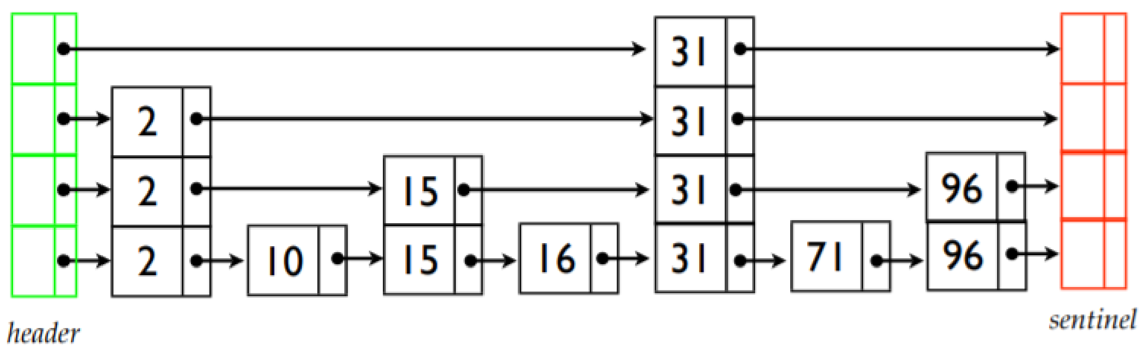
Skip lists are very useful when you need to be able to concurrently access your data structure. Imagine a **red-black tree**, an implementation of the **binary search tree**. If you insert a new node into the red-black tree, you might have to rebalance the entire thing, and you won't be able to access your data while this is going on. In a skip list, if you have to insert a new node, only the adjacent nodes will be affected, so you can still access large part of your data while this is happening.

1.2 Problem Description

In this project, we are required to introduce the skip lists, and to implement insertion, deletion, and searching of skip lists. A formal proof is also expected to show that the expected time for the skip list operations is $O(log\,N)$. Test cases of different sizes need to be generated to illustrate the time bound.

A **skip list** starts with a basic, ordered, **linked list**. This list is sorted, but we can't do a binary search on it because it is a linked list and we cannot index into it. But the ordering will come in handy later. Then, another layer is added on top of the bottom list. This new layer will include any given element from the previous layer with probability p. This probability can vary, but often times $\frac{1}{2}$ is used. Additionally, the first node in the linked list is often always kept, as a header for the new layer. Take a look at the following graphics and see how some elements are kept but others are discarded. Here, it just so happened that half of the elements are kept in each new layer, but it could be more or less--it's all probabilistic. In all cases, each new layer is still ordered.

Each element in the skip list has four pointers. It points to the node to its left, its right, its top, and its bottom. These **quad-nodes** will allow us to efficiently search through the skip list.



Graph 1 An example implementation of a skip list

Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list. $O(N)$ operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to $O(log\, N)$ **search** time. (Choose the level of the $i'th$ finite node to be 1 plus the number of times we can repeatedly divide i by 2 before it becomes odd. Also, $i = 0$ for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allows someone to know where all of the higher-than-level 1 nodes are and delete them.

Chapter 2 Algorithm Specification

2.1 Data Structure

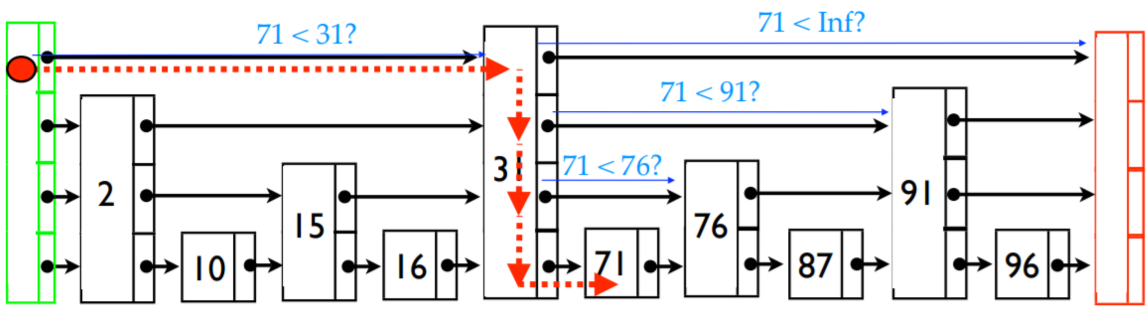
The data structure of the class **Entry** is:

```
class Entry
{
private:
public:
    // The Constructor
    Entry(int k, T v) : value(v), key(k), pNext(nullptr), pDown(nullptr) {}
    // The Copy-constructor
    Entry(const Entry &e) : value(e.value), key(e.key), pNext(nullptr), pDown(nullptr) {}
    int key;
    T value;
    Entry *pNext;
    Entry *pDown;

public:
    /* overload operators */
    bool operator<(const Entry &right)
    {
        return key < right.key;
    }
    bool operator>(const Entry &right)
    {
        return key > right.key;
    }
    bool operator<=(const Entry &right)
    {
        return key <= right.key;
    }
    bool operator>=(const Entry &right)
    {
        return key >= right.key;
    }
    bool operator==(const Entry &right)
    {
        return key == right.key;
    }
    Entry *&next()
    {
        return pNext;
    }
    Entry *&down()
    {
        return pDown;
    }
};
```

2.2 Search

To find an item, we scan along the shortest list until we would "pass" the desired item. At that point, we drop down to a slightly more complete list at one level lower. Remember the searching process is **sorted sequential searching**. In other words, when search for k, if $k = key$, done. Else if $k < next\ key$, go down a level. If $k \geq next\ key$, go right.

An example of searching for 71 is given by the following graph. First we go to 31, and 71>31, so we go right. 71<Inf, so we go down a level to 91. 71<91, so we go down a level to 76. 71<76, so we go down a level to 71. Then 71=71, we finish our searching.



Graph 2 An example search of a skip list

The input to this function is a search value and the output of this function is true or false. True if the value can be found in the skip list and false otherwise.

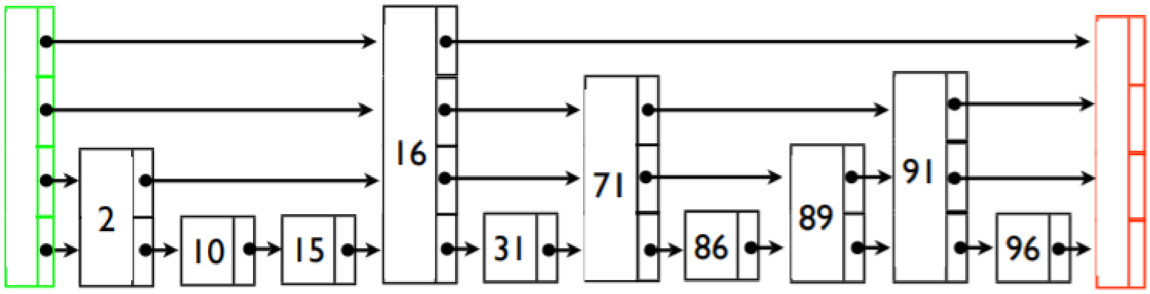
The pseudocode of the search process is:

```
/* search element */
Function search(Entry<T> *entry) const{
      if header->right == nullptr   // judge is_null
            return false;
      // find the accessible point in the top layer
      for i := 0 to levelNum-1
            if *entry < *cur_header->right then
                  cur_header = cur_header->down;
            end if
            else then
                  Entry<T> *cursor := cur_header->right;
                  while cursor->down() != nullptr   //scan down
                        while cursor->next() != nullptr
                              if *entry <= *cursor->next()
                                    break;
                              cursor := cursor->next();
                        end while
                        cursor := cursor->down();
                  end while
                  while cursor->next() != nullptr   //scan forward
                        if *entry > *cursor->next() then
                              cursor := cursor->next();
                        else if *entry == *cursor->next() then
                              return true;
                        else then
                              return false;
                  end while
                  return false; // Node larger than the last element node, return false
            end else
            i:=i+1;
      end for
      return false; // no access point is found
}
```

2.3 Insertion

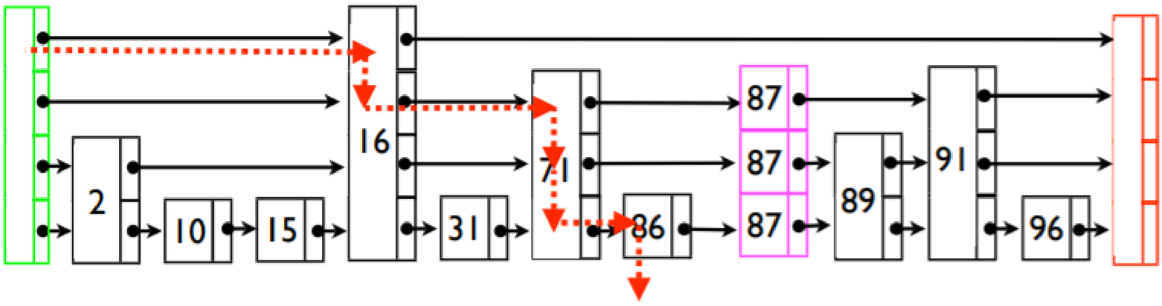
To find an item, we scan along the shortest list until we would “pass” the desired item. At that point, we drop down to a slightly more complete list at one level lower. Remember the searching process is **sorted sequential searching**. In other words, when search for k, if $k = key$, done. Else if $k < next\ key$, go down a level. If $k \geq next\ key$, go right.

To insert or delete an item, we might need to rearrange the entire list. The original **Perfect Skip List** is too structured to support efficient updates so we turn it into **Randomized Skip List**. Relax the requirement that each level have exactly half the items of the previous level. Instead design structure so that we expect 1/2 the items to be carried up to the next level. Because Skip Lists are a randomized data structure, the same sequence of inserts / deletes may produce different structures depending on the outcome of random coin flips.



Graph 3 An example implementation of a randomized skip list

An example of deletion of 87 is given by the following graph. First we need to find 87 and insert node in level 0, then let i = 1. While FLIP() == "heads", insert node into level i and i++. Just insertion into a linked list after last visited node in level i. Finally we get our insertion done.



Graph 4 An example insertion of a skip list

The input to this function is also a value and the output of this function is the topmost position at which the input is inserted. We are using the **Search** method from above.

First, we always insert the key into the bottom list at the correct location. Then, we have to promote the new element. We do so by flipping a fair coin. If it comes up heads, we promote the new element. By flipping this fair coin, we are essentially deciding how big to make the tower for the new element. We scan backwards from our position until we can go up, and then we go up a level and insert our key right after our current position.

While we are flipping our coin, if the number of heads starts to grow larger than our current height, we have to make sure to create new levels in our skip list to accommodate this.

The pseudocode of the insertion process is:

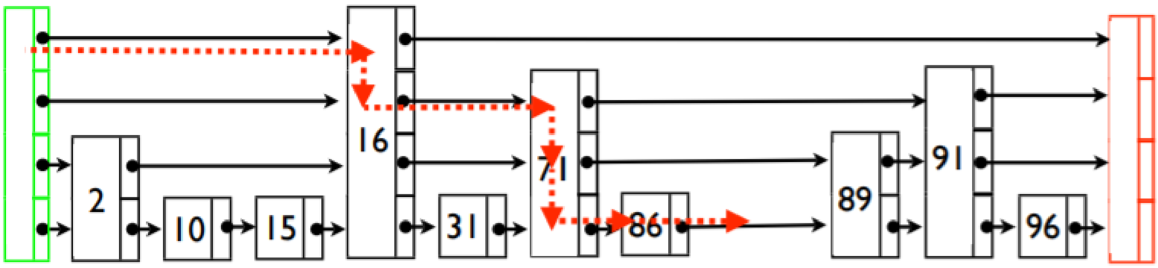
```
/* Insert new element */
Function insert(Entry<T> *entry){
     // Insertion is a series of bottom-up operations
      struct Endpoint *cur_header := header;
      // use the linked list header to get to the bottom
      while cur_header->down != nullptr
            cur_header := cur_header->down;

      // It is necessary to insert elements at the bottom
      // Whether to insert the jumping layers above is determined by random */
      do while random()!=0
            Copy new objects
            //  determine whether the current layer already exists. If not, add it
            cur_lvl:=cur_lvl+1;
            if levelNum < cur_lvl then
                  levelNum:=levelNum+1;
                  Endpoint *new_header := new Endpoint();
                  new_header->down := header;
                  header->up := new_header;
                  header := new_header;
            end if
            // !=1 Represents cur_ The header needs to move up and connect to the underlying pointer
            if cur_lvl != 1 then
                  cur_header := cur_header->up;
                  cur_cp_entry->down() := temp_entry;
```

```
            end if
            temp_entry := cur_cp_entry;
            // Whether there is an element node in the current linked list.
            // If it is an empty linked list, assign a value to the right pointer and jump out of the loop
            if cur_header->right == nullptr then
                  cur_header->right = cur_cp_entry;
                  break;
            end if
            else
                  Create a cursor
                  while true
                        //Find the right insertable point in the current link list cycle
                        //jump out of the current cycle after finding it
                        if *cur_cp_entry < *cursor then
                              //Element is smaller than all elements of the current linked list
                              //insert the chain header
                              cur_header->right := cur_cp_entry;
                              cur_cp_entry->next() := cursor;
                              break;
                        else if cursor->next() == nullptr then
                              //Element is larger than all elements of the current list, insert the end of the list
                              cursor->next() := cur_cp_entry;
                              break;
                        end else if
                        else if *cur_cp_entry < *cursor->next() then
                              // Insert in the middle of the list
                              cur_cp_entry->next() := cursor->next();
                              cursor->next() := cur_cp_entry;
                              break;
                        end else if
                        cursor := cursor->next(); // Move cursor right
                  end while
            end else
      end while
}
```

2.4 Deletion

Deletion takes advantage of the **Search** operation and is simpler than the **Insertion** operation.

An example of deletion of 87 is given by the following graph. Similar to insertion, mainly we need to do is to find 87, delete 87 and restore the other nodes.



Graph 5 An example deletion of a skip list

The input to this function is a deletion value. Since we know when we find our first instance of key, it will be connected to all others instances of key, and we can easily delete them all at once.

The pseudocode of the deletion process is:

```
/* delete */
Function remove(Entry<T> *entry){
      if header->right == nullptr then
            return;
      int lvl_counter := levelNum; // Obtain the number of llevels before entering the cycle
      for i = 0 to levelNum-1
            if *entry == *cur_header->right then
                  Entry<T> *delptr := cur_header->right;
                  cur_header->right := cur_header->right->next();
                  delete delptr;
            end if
            else then
                  Entry<T> *cursor := cur_header->right;
                  while (cursor->next() != nullptr)
                        if (*entry == *cursor->next()then
                              find the node;
```

```
                                delete delptr;
                                break;
                        end if
                        cursor := cursor->next();
                end while
            end else
            // When moving down the chain header pointer
            // determine whether there is an entry node in the current chain list
            if cur_header->right == nullptr then
                Endpoint *delheader := cur_header;
                cur_header := cur_header->down;
                header := cur_header;
                delete delheader;
                levelNum := levelNum - 1;
            end if
            else then
                cur_header := cur_header->down;
            end else
      i:=i+1;
      end for
}
```

Chapter 3: Testing Results

In the testing part, we generated 50,000 random integers in range from 1 to 99,999. There are 14 cases of different input size in each test. For every size in these tests, we repeated the process for 7 times and then calculate the average data as the result. For every test, each operation is done for 1,000 times. This is because fewer operations can't be caught by the program.

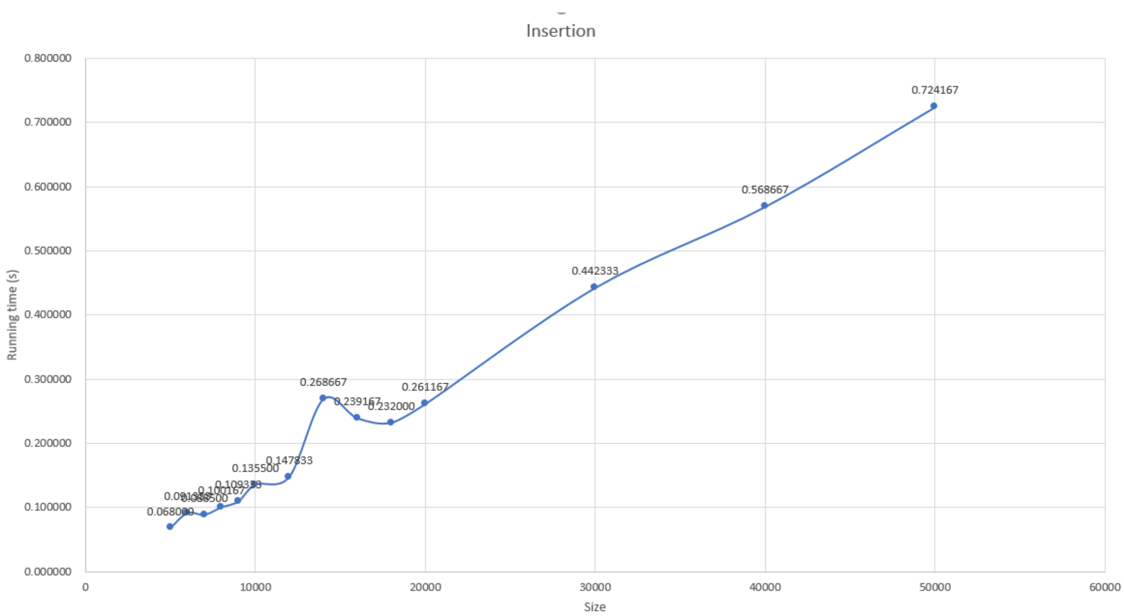
3.1 Insertion Testing Results

In this part, we are going to do 5000, 6000, ... ,50000 insertions. The same size of insertions will be done for seven times and then get the average time.

3.1.1 Testing Table

Size of Input	Average Time (1,000 times; in seconds)
5000	0.068000
6000	0.091333
7000	0.088500
8000	0.100167
9000	0.109333
10000	0.135500
12000	0.147833
14000	0.268667
16000	0.239167
18000	0.232000
20000	0.261167
30000	0.442333
40000	0.568667
50000	0.724167

3.1.2 Testing Graph



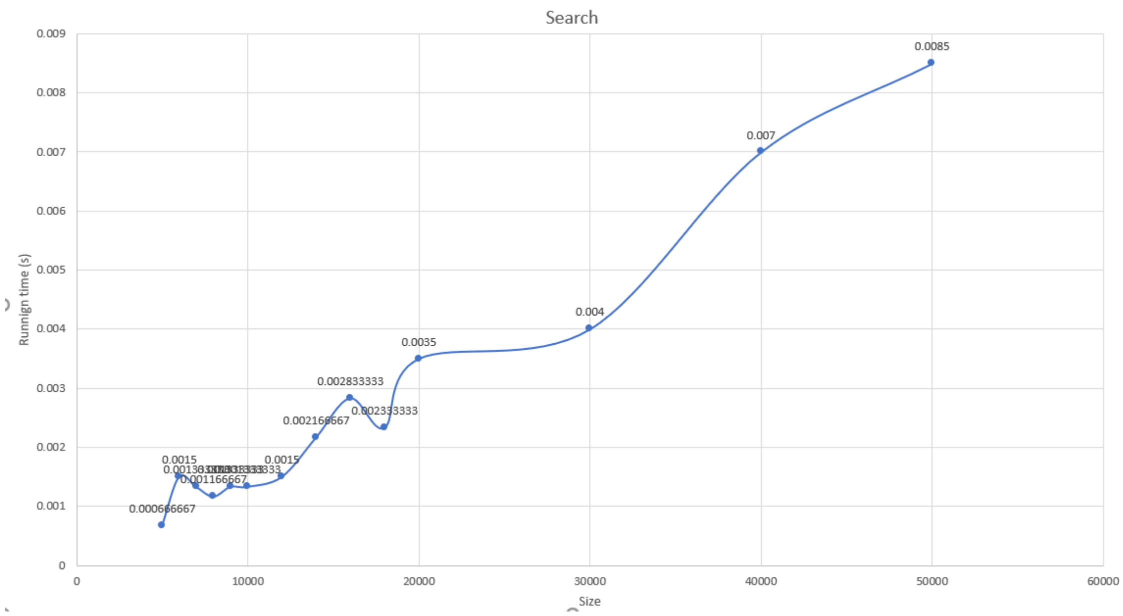
3.2 Search Testing Results

In this part, we are going to do 5000, 6000, ... ,50000 searches. The same size of searches will be done for seven times and then get the average time.

3.1.1 Testing Table

Size of Input	Average Time (1,000 times; in seconds)
5000	0.00067
6000	0.00150
7000	0.00133
8000	0.00117
9000	0.00133
10000	0.00133
12000	0.00150
14000	0.00217
16000	0.00283
18000	0.00233
20000	0.00350
30000	0.00400
40000	0.00700
50000	0.00850

3.2.2 Testing Graph



3.3 Deletion Testing Results

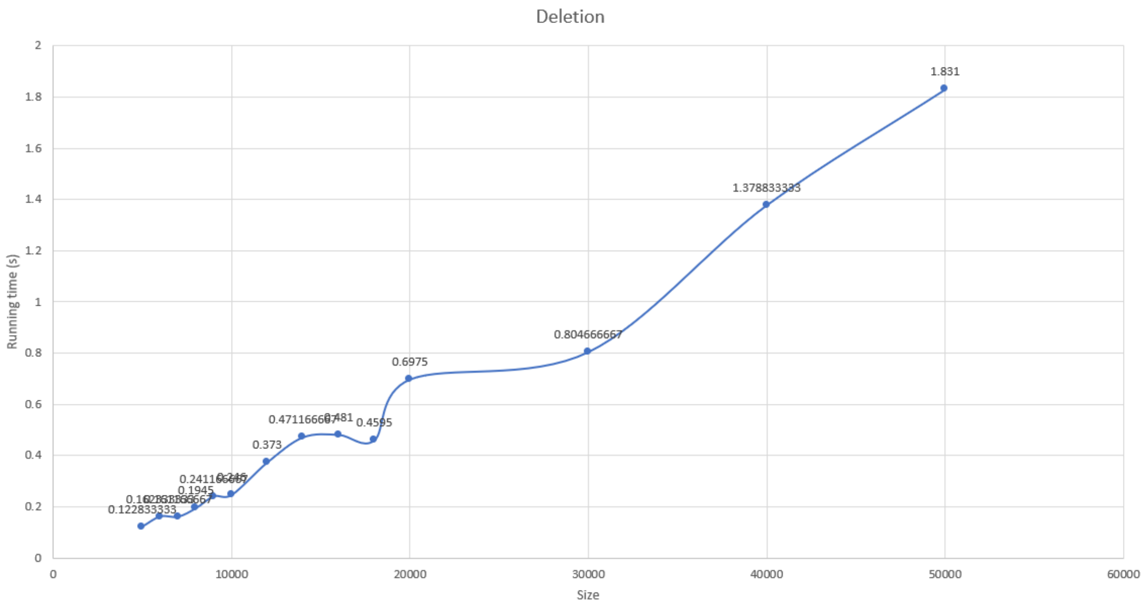
In this part, we are going to do 5000, 6000, ... ,50000 deletions. The same size of deletions will be done for seven times and then get the average time.

3.3.1 Testing Table

Size of Input	Average Time (1,000 times; in seconds)
5000	0.12283
6000	0.16233
7000	0.16117
8000	0.19450
9000	0.24117
10000	0.02460
12000	0.03730
14000	0.47117

Size of Input	Average Time (1,000 times; in seconds)
16000	0.48100
18000	0.45950
20000	0.69750
30000	0.80467
40000	1.37883
50000	1.83100

3.3.2 Testing Graph



Chapter 4: Analysis and Comments

4.1 Analysis

4.1.1 Time Complexity Analysis

Insertion

In the best case, the skip list is empty. In this case, time complexity is $O(1)$

In the worst case, the skip list is only one level and the inserted one is larger than any data in the skip list. In this case, time complexity is $O(N)$

In the average case, the move between adjacent levels is $O(1)$. Since each node is of 50% possibility to generate a lower level, the average comparisons are $O(\log N)$. After that, the update step is $O(\log N)$ because there are at most $\log N$ levels and the move between adjacent levels is $O(1)$. To conclude, the time complexity is $O(\log N)$.

Search

As we discussed in the insertion time complexity analysis, the move between adjacent levels is $O(1)$. Since each node is of 50% possibility to generate a lower level, the average comparisons are $O(\log N)$. To conclude, the time complexity is $O(\log N)$.

Deletion

Since deletion is almost the same as insertion, we omit the analysis here.

4.1.2 Space Complexity Analysis

In our designation, for any node at i level, it may generate a node at $i-1$ level with the possibility of 50%.

Suppose the base list at i level contains n nodes. Then the average space consumption is

$$Space = n + \frac{1}{2}n + \frac{1}{2^2}n + \cdots + \frac{1}{2^k}n + 1$$

On average, $k < logn$

Therefore, the average space complexity is $O(N)$

However, in the worst case, every node generates a lower level. Therefore, the worst space complexity is $O(N \log N)$

4.2 Comments

At first, the size of data were chosen from 100 to 100,000. But it took too long to give an output, and the result will be too intensive at the beginning. Therefore, we selected the middle part of the previous testing result. Simply from the graph we cannot tell whether the result is linear or logarithmic (However, it is interesting that when we constructing the graph, it appeared as logarithm).

Another thing that we discovered is that skip lists behave randomly, thus, the specific running time may differ from others. Therefore, it is important to calculate the average running time basing on couples of tests.

From the graphs we can also discover that the positions of the very first points of each test are not so smooth. This is because the first several test sizes are relatively small and this random algorithm may create different cases of the lists. Therefore, it can be implied that when the testing size growing, the plot will be more like logarithm.

Appendix

I Source Code in C++

```cpp
#pragma once
#ifndef SKIPLIST_H_
#define SKIPLIST_H_
#include <ctime>
#include <cstdlib>
#include<iostream>
using namespace std;
template<typename T>
class Entry {
private:
public:
    // The Constructor
    Entry(int k, T v) :value(v), key(k), pNext(nullptr), pDown(nullptr) {}
    // The Copy-constructor
    Entry(const Entry& e) :value(e.value), key(e.key), pNext(nullptr), pDown(nullptr) {}
    int key;
    T value;
    Entry* pNext;
    Entry* pDown;

public:
    /* overload operators */
    bool operator<(const Entry& right) {
        return key < right.key;
    }
    bool operator>(const Entry& right) {
        return key > right.key;
    }
    bool operator<=(const Entry& right) {
        return key <= right.key;
    }
    bool operator>=(const Entry& right) {
        return key >= right.key;
    }
    bool operator==(const Entry& right) {
        return key == right.key;
    }
    Entry*& next() {
        return pNext;
    }
    Entry*& down() {
        return pDown;
    }
};
template<typename T>
class SkipList_Entry {
private:
    struct Endpoint {
        Endpoint* up;
        Endpoint* down;
        Entry<T>* right;
    };
    struct Endpoint* header;
    int levelNum; // level_number 已存在的层数
    unsigned int seed;
    bool random() {
        srand(seed);
        int ret = rand() % 2;
        seed = rand();
        return ret == 0;
    }
public:
    SkipList_Entry() :levelNum(1), seed(time(0)) {
        header = new Endpoint();
    }
    /* Insert new element */
    void insert(Entry<T>* entry) { // Insertion is a series of bottom-up operations
        struct Endpoint* cur_header = header;
        // use the linked list header to get to the bottom
        while (cur_header->down != nullptr) {
            cur_header = cur_header->down;
        }
        // It is necessary to insert elements at the bottom
        // Whether to insert the jumping layers above is determined by random */
        int cur_lvl = 0; // current_level
        Entry<T>* temp_entry = nullptr; // Temporarily save a node pointer that has finished inserting
        do {
```

```
            Entry<T>* cur_cp_entry = new Entry<T>(*entry); // Copy new objects
            //  determine whether the current layer already exists. If not, add it
            cur_lvl++;
            if (levelNum < cur_lvl) {
                levelNum++;
                Endpoint *new_header = new Endpoint();
                new_header->down = header;
                header->up = new_header;
                header = new_header;
            }
            // !=1 Represents cur_ The header needs to move up and connect to the underlying pointer
            if (cur_lvl != 1) {
                cur_header = cur_header->up;
                cur_cp_entry->down() = temp_entry;
            }
            temp_entry = cur_cp_entry;
            // Whether there is an element node in the current linked list.
            // If it is an empty linked list, assign a value to the right pointer and jump out of the loop
            if (cur_header->right == nullptr) {
                cur_header->right = cur_cp_entry;
                break;
            }
            else {
                Entry<T>* cursor = cur_header->right; // create a cursor
                while (true) {
                    //Find the right insertable point in the current link list cycle
                    //jump out of the current cycle after finding it
                    if (*cur_cp_entry < *cursor) {
                        //Element is smaller than all elements of the current linked list, insert the chain header
                        cur_header->right = cur_cp_entry;
                        cur_cp_entry->next() = cursor;
                        break;
                    }
                    else if (cursor->next() == nullptr) {
                        //Element is larger than all elements of the current list, insert the end of the list
                        cursor->next() = cur_cp_entry;
                        break;
                    }
                    else if (*cur_cp_entry < *cursor->next()) { // Insert in the middle of the list
                        cur_cp_entry->next() = cursor->next();
                        cursor->next() = cur_cp_entry;
                        break;
                    }
                    cursor = cursor->next(); // Move cursor right
                }
            }
        } while (random());
    }

    /* search element */
    bool search(Entry<T>* entry) const {
        if (header->right == nullptr) { // judge is_null
            return false;
        }
        Endpoint* cur_header = header;
        // find the accessible point in the top layer
        for (int i = 0; i < levelNum; i++) {
            if (*entry < *cur_header->right) {
                cur_header = cur_header->down;
            }
            else {
                Entry<T>* cursor = cur_header->right;
                while (cursor->down() != nullptr) {
                    while (cursor->next() != nullptr) {
                        if (*entry <= *cursor->next()) {
                            break;
                        }
                        cursor = cursor->next();
                    }
                    cursor = cursor->down();
                }
                while (cursor->next() != nullptr) {
                    if (*entry > *cursor->next()) {
                        cursor = cursor->next();
                    }
                    else if (*entry == *cursor->next()) {
                        return true;
                    }
                    else {
                        return false;
                    }
                }
                return false; // Node larger than the last element node, return false
            }
        }
        return false; // no access point is found
    }
```

```cpp
    /* delete */
    void remove(Entry<T>* entry) {
        if (header->right == nullptr) {
            return;
        }
        Endpoint* cur_header = header;
        Entry<T>* cursor = cur_header->right;
        int lvl_counter = levelNum; // Obtain the number of llevels before entering the cycle
        for (int i = 0; i < levelNum; i++) {
            if (*entry == *cur_header->right) {
                Entry<T>* delptr = cur_header->right;
                cur_header->right = cur_header->right->next();
                delete delptr;
            }
            else {
                Entry<T> *cursor = cur_header->right;
                while (cursor->next() != nullptr) {
                    if (*entry == *cursor->next()) { // find the node
                        Entry<T>* delptr = cursor->next();
                        cursor->next() = cursor->next()->next();
                        delete delptr;
                        break;
                    }
                    cursor = cursor->next();
                }
            }
            // When moving down the chain header pointer
            // determine whether there is an entry node in the current chain list
            if (cur_header->right == nullptr) {
                Endpoint* delheader = cur_header;
                cur_header = cur_header->down;
                header = cur_header;
                delete delheader;
                levelNum--;
            }
            else {
                cur_header = cur_header->down;
            }
        }
    }
    void printList() {
        if (header->right == nullptr) { // Judge whether the right side of the chain header is a null pointer
            return;
        }
        Endpoint* cur_header = header;
        for (int i = 0; i < levelNum; i++) {
            Entry<T>* cursor = cur_header->right;
            cout << "Level" << i << "\t";
            while (cursor->next() != nullptr) {
                cout << cursor->key << ":" << cursor->value<<"  ";
                cursor = cursor->next();
            }
            cout << cursor->key << ":" << cursor->value<<endl;
            cur_header = cur_header->down;
        }
        cout << endl;
    }
};
#endif // !SKIPLIST_H_
```

II Performance and Run Time Testing Program in C++

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "SkipList.h"
#define MAX_LEVEL 16
#define MAX_NUM 100000
using namespace std;

int main()
{
      clock_t start, total;
      clock_t insertion[14] = {0}, search[14] = {0}, deletion[14] = {0};
      int Test_set[MAX_NUM] = {0};
      int size[14] = {5000, 6000, 7000, 8000, 9000, 10000, 12000, 14000, 16000, 18000, 20000, 30000, 40000, 50000};

      //read in test set
      freopen("test_set.in", "r", stdin);
      freopen("test_result.out", "w", stdout);

```

```
    //performance test
    SkipList_Entry<int> mysl;
    printf("Start performance test:\n");
    for (int i = 0; i < 10; i++)
    {
        Entry<int> my(i, i);
        mysl.insert(&my);
        cout << "after insert key=" << i << " value=" << i << endl;
        mysl.printList();
    }

    for (int i = 0; i < 5; i++)
    {
        Entry<int> my(i, i);
        mysl.remove(&my);
        cout << "after delete key=" << i << " value=" << i << endl;
        mysl.printList();
    }

    for (int i = 6; i < 12; i++)
    {
        Entry<int> my(i, i);
        bool exist = mysl.search(&my);
        cout << "after search key=" << i << " value=" << i << endl;
        if (exist)
            cout << "exist!" << endl;
        else
            cout << "not exist!" << endl;
    }

    //run time test
    printf("\nStart run time test:\n");
    for (int i = 0; i < MAX_NUM; i++)
        scanf("%d", &Test_set[i]);
    for (int i = 0; i < 14; i++)
    {
        SkipList_Entry<int> Test;

        for (int j = 0; j < size[i]; j++)
        {
            Entry<int> my(Test_set[j], j);
            Test.insert(&my);
        }
        start = clock();
        for (int j = 99000; j > 98000; j--)
        {
            Entry<int> my(Test_set[j], j);
            Test.insert(&my);
        }
        total = clock();
        insertion[i] = total - start;

        start = clock();
        for (int j = 99000; j > 98000; j--)
        {
            Entry<int> my(Test_set[j], j);
            Test.remove(&my);
        }
        total = clock();
        deletion[i] = total - start;

        start = clock();
        for (int j = 0; j < 1000; j++)
        {
            Entry<int> my(Test_set[j], j);
            Test.search(&my);
        }
        total = clock();
        search[i] = total - start;

        printf("Now handling %d\n", i);
    }

    printf("Search:------------------------------\n");
    for (int i = 0; i < 14; i++)
    {
        printf("%.3f\n", (double)search[i] / (double)CLOCKS_PER_SEC);
    }
    printf("Insertion:---------------------------\n");
    for (int i = 0; i < 14; i++)
    {
        printf("%.3f\n", (double)insertion[i] / (double)CLOCKS_PER_SEC);
    }
    printf("Deletion:----------------------------\n");
    for (int i = 0; i < 14; i++)
    {
        printf("%.3f\n", (double)deletion[i] / (double)CLOCKS_PER_SEC);
```

```
    }

    fclose(stdin);
    fclose(stdout);
    system("pause");

    return 0;
}
```

III Testing Set Generating Script in PHP

```php
<?php
$file = fopen("test.txt","w+");
$set_array = array();
$size = array(100,200,300,400,500,600,700,800,900,1000,1500,2000,2500,3000,3500,4000,4500,5000,
6000,7000,8000,9000,10000,12000,14000,16000,18000,20000,30000,40000,50000,60000,70000,80000,90000,100000);
for ($fill = 0; $fill < 100000; $fill ++) {
      $random = mt_rand(1,99999);
      array_push($set_array,$random);

}
fwrite($file,"int Test_set[100000] = {") ;
for ($at = 0; $at < 100000; $at ++){
      if ($at!=99999)$string = $set_array[$at].",";
      else $string = $set_array[$at];
      fwrite($file,$string);
      if ($at > 0 && $at%50 == 0) {
            fwrite($file,"\n");
     }
      flush();
}
fwrite($file,"};");
fclose($file);
echo "done.";
?>
```

IV Declaration

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

V Duty Assignments

Programmer:
Tester:
Report Writer: