Skip Lists

Advance Data Structure

Project 7

Zewei Cai Yizhou Chen Jinze Wu

2019-6-6

Chapter 1 Introduction

1.1 Background Information

SkipList is a randomized data structure. Based on parallel linked list, it is easy to implement. The complexity of insertion, deletion and search is O (logN) (in most cases), because its performance is comparable to that of the red-black tree and its implementation is relatively simple, so jump tables are used to replace the red-black tree in many famous projects, such as SkipList, which is the underlying storage structure of LevelDB and Reddis.

There are three commonly used key-value data structures: Hash table, red-black tree and SkipList. They have different advantages and disadvantages.

- Hash table: The fastest insertion and search is O (1); unlock can be achieved if linked list is used; data ordering requires explicit sorting operation.
- Red-Black Tree: Insertion and lookup are O (logn), but the constant term is small; the implementation of unlock-free is very complex, and usually needs to be locked; the data is naturally ordered.
- SkipList: Insertion and lookup are O (logn), but the constant term is larger than the red-black tree; the underlying structure is linked list, which can be implemented without lock; the data is natural and orderly.

1.2 Problem Description

General Description

This project requires us to complete the basic operation of jump table, including insertion, search and deletion, and get its time complexity by testing data of different orders of magnitude. (and prove it in a formal way)

Our Work

To solve this problem, our team first learned about the jump table and completed the code of the jump table independently. Then we tested the performance of the jump table by generating different random data, and proved its time complexity.

Chapter 2 Algorithm Specification

2.1 Structure Description

Specification:

We use two structures to implement a complete skip list.

For the skip list, it has a top node pointer, the current level of the whole list and the upperlevel as the upper bound of the level of the list.

For each node in the skip list, it has a next pointer pointing to the next node in the same level, a down pointer pointing to the node storing the same value in the down level, and its own value.

Pseudo Code:

structure of skip list node

```
1:  Struct skipListNode
2:      pointer of skipListNode next, down
3:      ElementType value
```

structure of skip list

```
1:  Struct skipListNode
2:      Int level, upperlevel
3:      Pointer of skipListNode top
```

2.2 Main Sketch of the Program

Specification:

The whole program mainly consists of 7 functions: a main function, a function to create an empty skip list, a functions to insert an element, a function to find an element, a function to delete an element, a function to generate random level, and a function to print out the whole skip list.

Pseudo code:

Main Function

```
1: function main ( )
2:   Input : N size of the skip list
3:   s = createSkipList (N)
4:   For i = 0 to N-1
5:       input x
6:       insert x into s
7:   initialize insert_time = find_time = delete_time = 0
8:   Input: M number of test cases
7:   For i = 0 to M-1
8:       input x
9:       start = current time
10:      insert x into s
11:      stop = current time
12:      insert_time = insert_time + (stop – start)
13:      input y
14:      start = current time
15:      find y in s
16:      stop = current time
17:      find_time = find_time + (stop – start)
18:      input z
19:      start = current time
20:      delete z from s
21:      stop = current time
22:      delete_time = delete_time + (stop – start)
23:  output insert_time / m, find_time / m, delete_time / m
13: End function
```

2.3 Insert Element Function

2.3.1 random level Function

specification:

Simulating the filp of a coin, every time there’ s a 50% probability to increase one more level. We randomly generate a number, if it is even, add 1 to the level, otherwise stop and return the current level.

pseudo code:

RandomLevel Function

```
1: RandomLevel(max)
2:    k = 1
3:    while rand() mod 2 == 0
4:        k = k + 1
5:    If k > max then
6:        k = max
7:    return k
8: End function
```

2.3.2 Insert

specification:

step1: randomly generate a level for the new nodes

step2: start from top find the starting level to insert

step3: create new levels if need and insert new nodes into each level

pseudo code:

insertElement Function

```
1: insertElement (skipList pointer s, ElementType x)
2:    level = RandomLevel()
3:    skipListNode Array newNodes[level]
4:    For i = level-1 downto 0
5:        create new skip list node
6:        if is the last node then
7:            newNodes[i]->down = NULL
8:        else
9:            newNodes[i]->down = newNodes[i+1]
```

```
10: currentNode = s->top
11: For i = 0 to s->level-level
12:     currentNode = currentNode->down
13:     while currentNode->next->value < x
14:         currentNode = currentNode->next
15: For i = 0 to level
16:     if i < level – s->level then
17:         create new headnode
18:         headnod->next = newNode[i]
19:     else
20:         currentNode = currentNode->down
21:         while currentNode->next < x
22:             currentNode = currentNode->next
23:     insert newNode[i] behind currentNode
24: if level > s->level then
25:     s->level = level
26: return s
27: End function
```

2.4 Find Element Function

Specification:

Start from the top and find the minimal element which is larger than the element we need to find, and then move down level by level.

Pseudo code:

Function findElement

```
1: findElement(skipList pointer s, ElementType x)
2:   currentNode = s->top->down;
3:   while currentNode != NULL
4:       while currentNode->next->value < x
5:           currentNode = currentNode->next
6:       if currentNode->next->value == x
7:           return true
```

```
8 :        currentNode = currentNode->down
10:  return false
11:end function
```

2.5 Delete Element Function

Specification:

Start from the top and find the minimal element which is larger than the element we need to delete, and then move down level by level. If the element is found, it will be removed from the list and then deleted. If the level after delete become empty, then this level will be removed and the head node will be deleted.

Pseudo code:

Function deleteElement

```
1: deleteElement(skipList pointer s, ElementType x)
2:   headNode = currentNode = s->top->down;
3:   while currentNode != NULL
4:        while currentNode->next->value < x
5:             currentNode = currentNode->next
6:        if currentNode->next->value == x
7:             delete currentNode->next
8:        if headNode->next == NULL
9:             remove current level
10 :       headNode = headNode->down
11 :       currentNode = currentNode->down
12:  return s
11:end function
```

Chapter 3 Testing Result

3.1 Test Method Specification

3.1.1 Correctness test

Method:

We use small cases to test if our skip list is implemented correct. We test different operation in different conditions, and print out the generated skip list to check by ourselves.

Data:

Design by ourselves.

Purpose:

To check our implementation of our skip list.

3.1.2 Time consumption test

Method:

To test the time consumption for three operations in skip lists of different sizes:

Step1: input N, size of the skip list; M, number of test cases

Step2: generate a skip list of size N

Step3: respectively do find, insert, delete to the skip list and record time for M times

Then we have the three time consumptions find_time_of_M, insert_time_of _M, delete_time_of M, which are the total time consumptions of M operations. We divide the total time consumptions by M, then we get the time consumption of a single operation.

Data:

Use rand() function to generate N different numbers and use them to generate a skip list of size N:

Use data[i] = 4 * i + rand() % 4 to ensure N different numbers

shuffle data array to generate a random input

Then we generate M test cases:

insert: randomly generate M numbers different from former N numbers

find: 50% pick one number in data array and 50% new random numbers

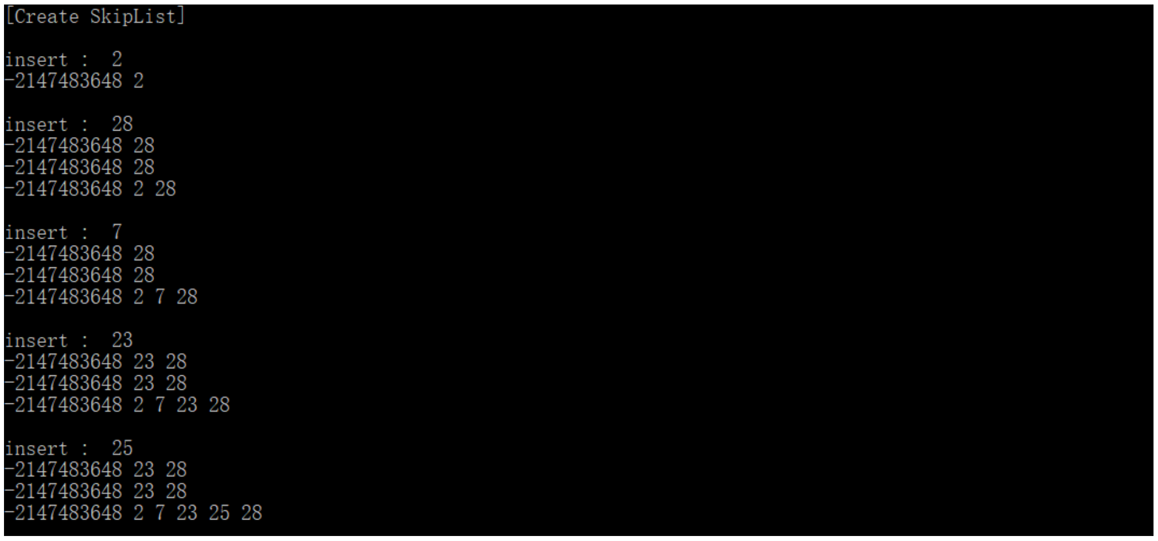
delete: use data[i % N] as the i^th case number to delete

Purpose:

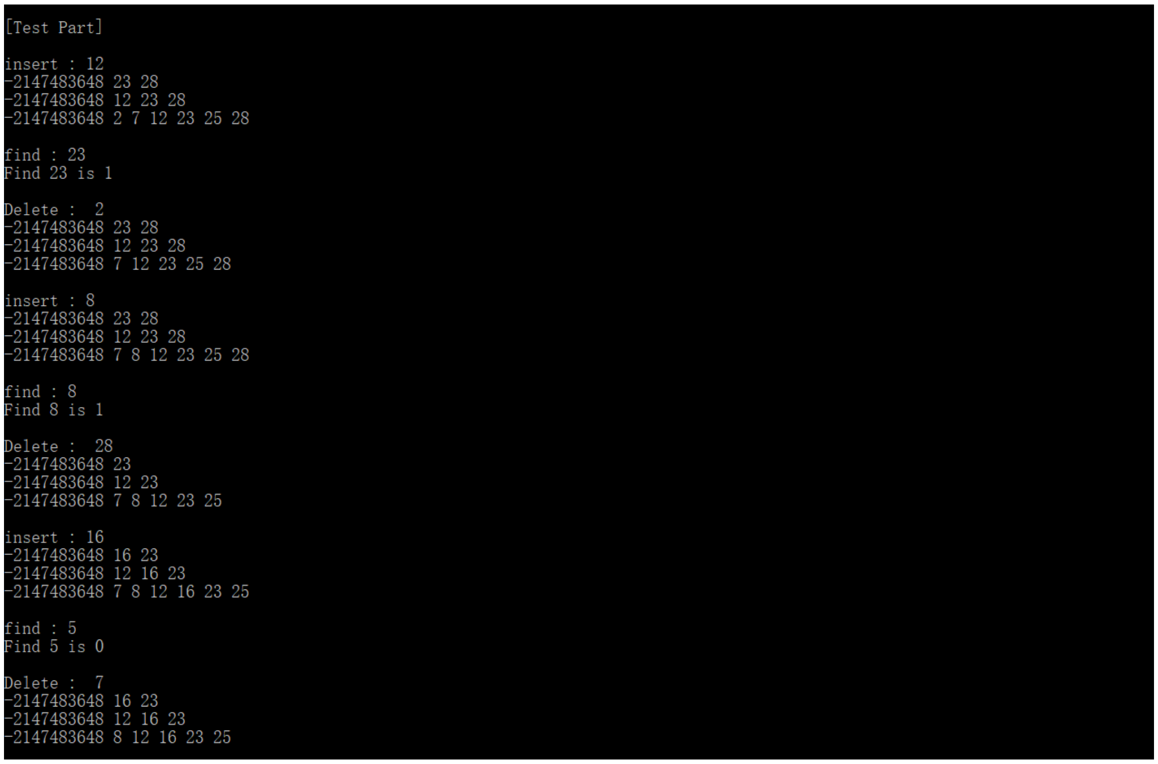
Test the time consumption and check if it can testify our theoretical analysis.

3.3 Test Data and Plot

3.3.1 Correctness test



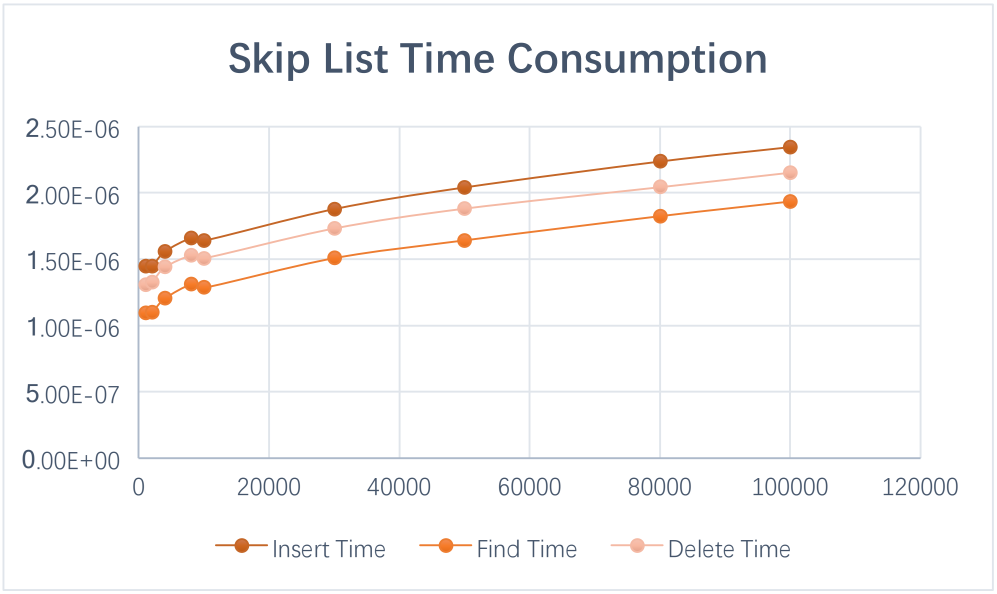
```
[Create SkipList]

insert :  2
-2147483648 2

insert :  28
-2147483648 28
-2147483648 28
-2147483648 2 28

insert :  7
-2147483648 28
-2147483648 28
-2147483648 2 7 28

insert :  23
-2147483648 23 28
-2147483648 23 28
-2147483648 2 7 23 28

insert :  25
-2147483648 23 28
-2147483648 23 28
-2147483648 2 7 23 25 28
```



```
[Test Part]

insert : 12
-2147483648 23 28
-2147483648 12 23 28
-2147483648 2 7 12 23 25 28

find : 23
Find 23 is 1

Delete :  2
-2147483648 23 28
-2147483648 12 23 28
-2147483648 7 12 23 25 28

insert : 8
-2147483648 23 28
-2147483648 12 23 28
-2147483648 7 8 12 23 25 28

find : 8
Find 8 is 1

Delete :  28
-2147483648 23
-2147483648 12 23
-2147483648 7 8 12 23 25

insert : 16
-2147483648 16 23
-2147483648 12 16 23
-2147483648 7 8 12 16 23 25

find : 5
Find 5 is 0

Delete :  7
-2147483648 16 23
-2147483648 12 16 23
-2147483648 8 12 16 23 25
```

3.3.2 Time consumption test

Size(N)	Insert Time	Find Time	Delete Time
1000	1.45E-06	1.09E-06	1.31E-06
2000	1.45E-06	1.10E-06	1.33E-06
4000	1.56E-06	1.21E-06	1.44E-06
8000	1.66E-06	1.31E-06	1.53E-06
10000	1.64E-06	1.29E-06	1.51E-06
30000	1.88E-06	1.51E-06	1.73E-06
50000	2.04E-06	1.64E-06	1.88E-06
80000	2.24E-06	1.82E-06	2.04E-06
100000	2.35E-06	1.93E-06	2.15E-06



Analysis:

Both insert and delete operation are based on find operation, and insert operation is a bit more complex than delete operation. So it is reasonable that insert_time > delete_time > find_time. But generally, there is no significant difference in their actual time cost, and they all show the increasing rate corresponding to logn function. So it is reasonable to believe that the time complexity of the three operations are all O(logn)

Chapter 4 Analysis and Comments

4.1 Analysis and Comments of the Testing Data

- Correctness Analysis
  Through testing three operations and print out the skip list after each operation, we can basically confirm that our implementation is correct.
- Time Complexity Analysis
  From the time testing graph, we can see that the time consumption corresponding to the O(logN) time complexity

4.2 Analysis about Time Complexity

For find operation, its time complexity mainly relies on the length of the search route. To find out the time complexity of find operation, we define:

$$f(N): expectaion\ length\ of\ search\ route\ in\ skip\ list\ of\ size\ N$$
$$p: probability\ for\ one\ node\ to\ exists\ on\ one\ more\ level$$

we have:

$$f(N) = f(pN) + 1 + (1-p) + (1-p)^2 + (1-p)^3 + \cdots = f(pN) + \frac{1}{p}$$

solve:

$$f(N) = f(pN) + \frac{1}{p}$$

we have:

$$f(N) = log_a N\ where\ a = (\frac{1}{p})^p$$

So, for find operation: time complexity is $O\ (log_a N)$

For insert and delete operation, they include adding and deleting nodes, which are associated with height of skip list. So, for insert and delete operation: time complexity is $O\left(log_a N + \max(height)\right) = O\ (log_a N)$

4.3 Analysis about Space Complexity

we define:

$$X_L : number\ of\ nodes\ on\ level\ L$$

then we have:

$$E(X_{L)} = N \times Pr = Np^{L-1}$$

$$Np^{L-1} = 1 \rightarrow L = log_{\frac{1}{p}}N + 1 \rightarrow \max level$$

the expectation height of each node: $h = 1 + p + p^2 + p^3 + \cdots = \frac{1}{1-p}$

the expectation space of N nodes: $s = Nh = \frac{N}{1-p}$

so, the space complexity is O(N)

Appendix

Appendix A Source code

- **Data**

```cpp
#include<iostream>
#include<cstdlib>
#include<ctime>

using namespace std;

int main(void) {

    freopen("data.in","w",stdout);

    int n,m;
    cin>>n>>m;
    int* data = new int[n+m+1];  // to store the insert function and delete func
tion data order

    srand((unsigned)time(NULL));

    /* generate random data */
    for (int i = 0; i < n+m; i++)
        data[i] = 4 * i + rand() % 4;  // make sure they are different

    /* shuffle to generate a random input */
    for (int i = 0; i < n+m-1; i++) {
        int j = i + rand() % (n+m-i);
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }

    /* output to file */
    cout<<n<<endl;
    for (int i = 0; i < n; i++) cout<<data[i]<<" ";
    cout<<endl;

    /* output the test cases */
    cout<<m<<endl;
    for (int i = 0; i < m; i++) {
        int x = (i % 2 == 0) ? rand() % ((n+m) * 4) : data[n+m-1-i];  // generat
e a random number for find function
        cout<<data[n+i]<<" "<<x<<" "<<data[i]<<endl;
    }

    return 0;
}
```

- ## Skip List

```
#include<iostream>
#include<cstdlib>
#include<vector>
#include<algorithm>
#include<ctime>
#include<cstdlib>
#include<cmath>

using namespace std;

const int Min = -2147483648;

/* the structure of each node in the skip list */
typedef struct skipListNode {
    skipListNode *next, *down;
    int value;
} *skipListNodePtr;  // the pointer of skip list node

/* the structure of the skip list */
typedef struct skipList {
    int level, upperlevel;
    skipListNodePtr top;
} *skipListPtr;  // the pointer of skip list

int randomLevel(int max) {
    int k = 1;
    /* initialize the seed */
    /* if the random number is 1 we get the level increased by 1 otherwise stop
*/
    while (rand() % 2)
        k++;
    if (k > max)
```

```cpp
        k = max;
    /* return k as the random generated level */
    return k;
}

/* create a null skip list */
skipListPtr createSkipList(int n) {
    skipListPtr s = new skipList;
    s->level = 0;  // initialize the level to be zero
    s->upperlevel = log(n*1.0) / log(2.0) + 1;
    /* create and initilize the top node */
    s->top = new skipListNode;
    s->top->next = NULL;
    s->top->down = NULL;
    s->top->value = -1;

    return s;  // return the new skip list
}

/* insert one element into the skip list */
skipListPtr insertElement(skipListPtr s,int x) {
    /* get a random level */
    int level = randomLevel(s->upperlevel);

    /* create new nodes for each level */
    skipListNodePtr* newNodes = new skipListNodePtr[level];
    for (int i = level - 1; i >= 0; i--) {
        newNodes[i] = new skipListNode;
        newNodes[i]->value = x;
        newNodes[i]->next = NULL;
        newNodes[i]->down = (i == level-1) ? NULL : newNodes[i+1]; // link the nodes from high level to low level
    }

    skipListNodePtr currentNode = s->top;
    /* make the current node point to the first level to insert*/
    for (int i = 0; i < (s->level - level); i++) {
        currentNode = currentNode->down;  // move down
        while (currentNode->next != NULL && currentNode->next->value < x)
                currentNode = currentNode->next; // move backwards
    }

    /* insert the new nodes into each level */
    for (int i = 0; i < level; i++) {
        if (i < level - s->level) {
            /* create the new level and insert the new nodes */
            skipListNodePtr headNode = new skipListNode;
            headNode->value = Min;
            headNode->next = newNodes[i]; // insert the new nodes
            headNode->down = currentNode->down;
            currentNode->down = headNode;
            currentNode = headNode;
        }
        else {
            /* insert node to the existed levels */
            currentNode = currentNode->down; // move down by 1 level
            //cout<<currentNode->value<<"# ";
            while (currentNode->next != NULL && currentNode->next->value < x) {

                //cout<<"!";
                currentNode = currentNode->next; // move backwards
            }
            /* insert the new node into current level */
            newNodes[i]->next = currentNode->next;
            currentNode->next = newNodes[i];
        }
    }
    /* update the level of the skiplist */
    if (level > s->level)
        s->level = level;
     //cout<<endl;
     return s;
}

/* insert one element into the skip list */
skipListPtr deleteElement(skipListPtr s, int x) {
    /* headNode point to the first node in each line
       currentNode points to the current node we search to */
    skipListNodePtr headNode, currentNode;
    headNode = currentNode = s->top->down;
    while (currentNode != NULL) {
        while (currentNode->next != NULL && currentNode->next->value < x)
            currentNode = currentNode->next; // move backwards
        /* if the node to delete in this level */
        if (currentNode->next != NULL && currentNode->next->value == x) {
            /* delete the node */
            skipListNodePtr temp = currentNode->next;
            currentNode->next = currentNode->next->next;
            delete temp;
        }
        /* if the current level become empty */
        if (headNode->next == NULL) {
            /* remove this level */
            skipListNodePtr temp = headNode;
            s->top->down = headNode->down;
            headNode = headNode->down;
            /* delete the head node */
            delete temp;
            s->level--;
        }
        else
            headNode = headNode->down;  // move down

        currentNode = currentNode->down; // move down
    }

    return s;
}

/* find element in the skip list
   if find return true else return false */
bool findElement(skipListPtr s, int x) {
    skipListNodePtr currentNode = s->top->down;
    while (currentNode != NULL) {
        while (currentNode->next != NULL && currentNode->next->value < x)
            currentNode = currentNode->next;  // move backwards
        if (currentNode->next != NULL && currentNode->next->value == x)
            return true;  // find
        currentNode = currentNode->down;  // move down by 1 level
    }
    return false;  // didn't find through the whole skip list
}
```

```cpp

/* print out the skip list*/
void printSkipList(skipListPtr s) {
    skipListNodePtr head,k;
    head = s->top->down;
    /* if the list is empty */
    if (head == NULL)
        cout<<"emtpy"<<endl;
    /* print the skip list by row */
    while (head != NULL) {
        k = head;
        /* print the row */
        while (k != NULL) {
            cout<<k->value<<" ";
            k = k->next;
        }
        head = head->down;  // move down
        cout<<endl;
    }
}

int main(void) {
    int n,m,x;

    freopen("data.in","r",stdin);

    clock_t start,stop;
    srand((unsigned)time(NULL));

    cin>>n;  // the size of the test case

    /* create an empty skip list */
    skipListPtr s = createSkipList(n);

    /* generate a skip list of size n */
    for (int i = 0; i < n; i++) {
        cin>>x;
        insertElement(s,x);
        //printSkipList(s);
    }
    /* test operation time */
    cin>>m;  // number of test cases
    double insert_time = 0, find_time = 0, delete_time = 0;

    for (int i = 0; i < m; i++) {
        /* test the insert function */
        cin>>x;
        start = clock();
        insertElement(s,x);
        //printSkipList(s);
        stop = clock();
        insert_time += double(stop - start)/CLOCKS_PER_SEC;

        /* test the find function */
        cin>>x;
        start = clock();
        bool b = findElement(s,x);
        stop = clock();
        find_time += double(stop - start)/CLOCKS_PER_SEC;

        /* test the delete function */
        cin>>x;
        start = clock();
        deleteElement(s,x);
        stop = clock();
        delete_time += double(stop - start)/CLOCKS_PER_SEC;
    }

    /* output the test result */
    cout<<"total time: "<<insert_time<<" "<<find_time<<" "<<delete_time<<endl;
    cout<<"time per operation: "<<insert_time/m<<" "<<find_time/m<<" "<<delete_time/m<<endl;

    return 0;
}
```

Appendix B Declaration

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

吴金泽

陈亲舟

蔡泽伟