

# Skip Lists

---

**Kang Mintong (programmer, tester, reporter)**

**Bao Yifan (programmer, tester, reporter)**

**Ye Fengshuo (programmer, tester, reporter)**

**Data: 2020-06-01**

# Chapter 1: Introduction

This project is about implementing skip lists to operate insertion, deletion, and searching. Also, theoretical analysis of time bound and experimental testing time result is required to be illustrated.

The skip list is a probabilistic 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.

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.

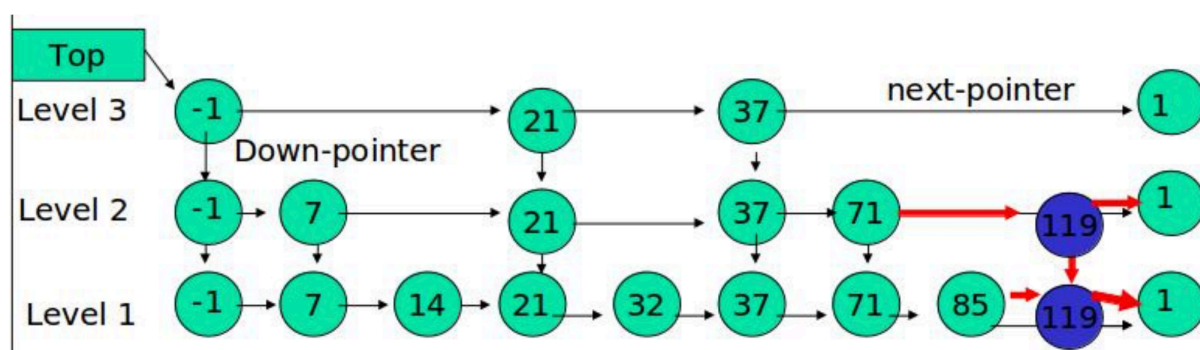
## Chapter 2: Data Structure / Algorithm Specification

### Main data structure

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequences of the items. A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.

As the figure shows, a skip list is built up of layers. The lowest layer is an ordinary ordered linked list. The higher layers are also ordered linked list but with some elements skipped[1].

figure 2-1: structure of skip list



We define a class of skip list to store different nodes and different layers of the skip list. Also, we define a class of node to specify the value in that node and two links of the next node.

## Algorithm specification

### Searching

When an element is tried to search, the search begins at the head element of the top list. It proceeds horizontally until the current element is greater than or equal to the target. If current element and target are matched, it means they are equal and search gets finished.

If the current element is greater than target, the search goes on and reaches to the end of the linked list, the procedure is repeated after returning to the previous element and the search reaches to the next lower list. The pseudocode is as follows.

```
Search(key)
    p = top-left node in S
    while (p.below != null) do           //Scan down
        p = p.below
        while (key >= p.next) do        //Scan forward
            p = p.next
    return p
```

### Insertion

First, we always insert the key into the bottom list at the correct location. Then, we have to produce the new element. We do so by randomly choosing. If it comes up heads, we produce the new element. By randomly choosing, 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. The pseudocode is as follows.

```
Insert(key)
    p = Search(key)
    q = null
    i = 1
    repeat
        i = i + 1                       //Height of tower for new element
        if i >= h
            h = h + 1
            createNewLevel()             //Creates new linked list level
        while (p.above == null)
            p = p.prev                   //Scan backwards until you can go up
        p = p.above
        q = insertAfter(key, p)          //Insert our key after position p
    until CoinFlip() == 'Tails'
    n = n + 1
    return q
```

## Deletion

In deletion, we can first find the position of the key in skip list and then delete all positions of that value. Finally, we should remove all blank rows.

```
Search for all positions p_0, ..., p_i where key exists
if none are found
    return
Delete all positions p_0, ..., p_i
Remove all empty layers of skip list
```

## Choosing a random level

To generate a randomized result to decide the level of nodes, we also implement an algorithm to randomly choosing. We say that a fraction  $p$  of the nodes with level  $i$  pointers also have level  $i+1$  pointers. (for many circumstances,  $p = 1/2$ ).

```
randomLevel()
lvl := 1
-- random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
    lvl := lvl + 1
return lvl
```

## Chapter 3: Testing Results

The main task of testing in this project is to prove the operation time bounding of skip list with size  $n$  is  $O(\log n)$ . So we have following test code. We design the test cases to testify the average time bounding. We construct a skip list of size  $n$  first and then do searching, insertion and deletion repeatedly to make results more accurately.

```
int main(){
    //to measure time more accurately
    int N,times;
    //define the number of
    operations and times of the program execution
    cout<<"Please enter the size of skip list: "<<endl;
    cin>>N;
    cout<<"Please enter the times the operation needs to be executed: "<<endl;
    cin>>times;
    srand( (unsigned)time( NULL ));
    double dur=0;
    clock_t start,end;
    skipList list;
    for(int i = 0;i < N;i++) list.Insert(i);
    //construct skip list of
    size N
    start = clock(); //start timing
```

```

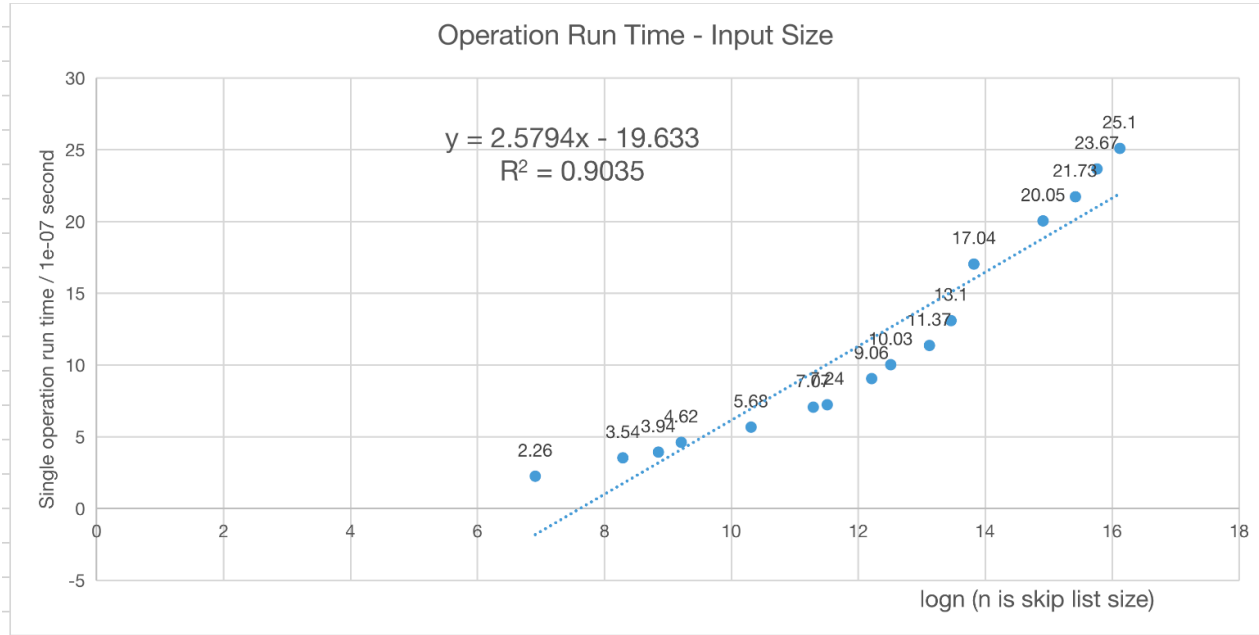
for(int j=0;j<times;j++)                                //do test many times
{
    list.Search(rand()%N);                                //search random element
    int key=rand()%N;
    list.Insert(key);                                     //insert and delete to
ensure the size unchanged
    list.Delete(key);
}
end=clock();
dur = (double)(end - start);
cout<<endl<<"Running time of single operation on skiplist with size "
<<N<<" is "<<(dur/CLOCKS_PER_SEC/times/3)<<endl; //print out time of single
operation
}

```

## Run time table

Case Number	Skip list size n	logn	Time per operation(1e-07 second)
1	1000	6.91	2.26
2	4000	8.29	3.54
3	7000	8.85	3.94
4	10000	9.21	4.62
5	30000	10.31	5.68
6	80000	11.29	7.07
7	100000	11.51	7.24
8	200000	12.21	9.06
8	300000	12.51	10.03
9	500000	13.12	11.37
10	700000	13.46	13.10
11	1000000	13.82	17.04
12	3000000	14.91	20.05
13	5000000	15.42	21.73
14	7000000	15.76	23.67
15	10000000	16.12	25.10

## Run time VS Input size figure



## Analysis and Proof

From the figure, because the  $R^2$  equals 0.9035 which means the data relationship between run time and logarithm of skip list size is linear, we can say that single operation run time of skip list is promotional to  $\log n$  (n is the size of skip list). Experimentally, we can get the conclusion that the expected time for the skip list operations is  $O(\log n)$ .

To demonstrate theoretical details of the time bounding, we provide a rigorous proof in Chapter 4.

## Chapter 4: Analysis and Comments

### Time and space complexities

#### Time complexity

In the following, we prove that the average time complexity is  $O(\log n)$  by referencing proof in [2].

First, we prove that the expected height of skip list is  $O(\log n)$ .

Because every element has a probability of  $1/2$  to go to the upper level, the probability  $P_i$  that an element in skip list with  $n$  total elements gets up to level  $i$  is as follows.

$$P_i = \frac{1}{2^i} \quad (1)$$

Therefore, the expected elements  $E_i$  in level  $i$  can be described as follows.

$$E_i = \frac{n}{2^i} \quad (2)$$

Suppose the expected height of skip list is  $h$ .

$$\frac{n}{2^h} = E_h = 1 \quad (3)$$

From (3), we know the expected height of skip list.

$$h = \log_2 n \quad (4)$$

Next, we prove that the steps on every level is small too.

Suppose  $C_i$  refers to expected steps to go up  $i$  levels. Because for every step, the probability of going up is  $1/2$  and the probability of staying at the same level is  $1/2$ , we have following equation.

$$C_i = 1 + 0.5C_{i-1} + 0.5C_i \quad (5)$$

From (5), we can derive as following.

$$C_i = C_{i-1} + 2 \quad (6)$$

From (6), we know that number of expected steps on every level is 2.

Considering we have  $O(\log n)$  levels, the total steps of searching, insertion and deletion is  $O(\log n)$ .

So the average time complexity of skip list operation is  $O(\log n)$ .

## Space complexity

From (2), we know that the expected number of nodes  $N$  can be derived as following.

$$N = \frac{n}{2^1} + \frac{n}{2^2} + \frac{n}{2^3} + \dots + 1 = 2n \quad (7)$$

In the algorithm, we also store the information of each level which costs  $O(\log n)$  space and other variable just cost  $O(1)$  space.

Therefore, the space complexity of the algorithm is  $O(n)$ .

## Comparison with other data structure

Compared with basic data structures with similar functions, skip list has the same average time bounding as them but easier to implement. Operations on AVL tree, splay tree, red-black tree, B+ tree all have time complexity of  $O(\log n)$  but they are much more difficult to implement. Also, the above structures has the same space complexity as skip list which is  $O(n)$ . As we can see, randomized method in skip list reduced the work load of coding but also works well.

## Further possible improvements

Usually, the going upper probability  $p$  is set to be  $1/2$  because symmetry always works well. However, that may not be true theoretically and experimentally. Referring to some essays, it's not easy to illustrate rigorous details of choosing the best  $p$ . The following is a picture extracted from an essay[3].

$p$	Normalized search times ( <i>i.e.</i> , normalized $L(n)/p$ )	Avg. # of pointers per node ( <i>i.e.</i> , $1/(1-p)$ )
$1/2$	1	2
$1/e$	0.94...	1.58...
$1/4$	1	1.33...
$1/8$	1.33...	1.14...
$1/16$	2	1.07...

TABLE 1 – Relative search speed and space requirements, depending on the value of  $p$ .

From the table, we can know that  $p=1/e$  works best in the value above when considering time bound. For a better implement,  $p=1/e$  might be more efficient but it need some complex math proof.

## Appendix: Source Code

```
#include<iostream>
#include<vector>
#include<math.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
#define INF 0x3f3f3f3f //infinite number to define the bound of every
level
#define PRECISION 9999 // the precision random value between(0,1)

class Node{ //class storing node information
public:
    int searchKey; // the search key
    vector<Node*> Next; // Next[i] means the ith level linked list
    int nodeLevel;
    Node(){} //different kinds of construction function
    Node(int key):searchKey(key){}
    Node(int key,int random_level):searchKey(key),nodeLevel(random_level)
{Next.resize(nodeLevel+1);}
};

class skipList{ //class storing skip list information
public:
    int maxLevel;
    int Level; //current level of list
```



```

Node* Header;
Node* NIL;
void Insert(int value);           //Insert a node with Key(value) to
skip_list
void Delete(int value);           //Delete a node with Key(value) from
skip_list
Node* Search(int value);          //Search a node with Key(value) from
skip_list
int RandomLevel(double p = 0.5);  //Generate node level according to
probability -- default is 0.5
skipList(int mlevel = 16);        //Constructor -- default maxLevel is 16
-- containing up to 2^16 elements
void PrintList();                 //debug
Node* MakeNode(int key,int level); //make a node with specified key and
level
};
void skipList::PrintList(){        //print every node's level position
Node* x = Header->Next[0];
while(x->searchKey != INF){        //transverse every node
cout << x->searchKey << "is on level " <<x->nodeLevel<<endl;
x = x->Next[0];
}
}
Node* skipList::MakeNode(int key,int level){    //construct a new node to
insert into skip list
Node* node = new Node;
node->searchKey = key;node->nodeLevel = level;node->Next.resize(level+1);
return node;
}
skipList::skipList(int mlevel):maxLevel(mlevel),Level(0){
Header = new Node;  NIL = new Node;
Header->searchKey = -INF;           //define smallest bounding
NIL->searchKey = INF;              //NIL value is biggest
Header->Next.resize(maxLevel+1);
Header->nodeLevel = mlevel;         //Header is of maximum level
for(int i = 0; i <= maxLevel;i++)  //Initialize Header
Header->Next[i] = NIL;
}
Node* skipList::Search(int key){    //search operation in skip list
Node *x = Header;
int i;
for( i = Level; i >= 0;i--){        //transverse every level
while(x->Next[i]->searchKey < key){
x = x->Next[i];                    //visit the next node
}
}
x = x->Next[0];                     // final state at level 0
if(x->searchKey == key) return x;
else return NULL;                  //fail

```

```

}
int skipList::RandomLevel(double p){ // p=1/2 in usual case
    int level = 0;
    double prob=rand()%(PRECISION+1)/((double)(PRECISION+1)); //generate random
number
    while(level <maxLevel){
        prob = rand()%(PRECISION+1)/((double)(PRECISION+1)); //use random number
to determine the level number
        if(prob >= p)break;
        level = level+1;
    }
    return level;
}
void skipList::Insert(int key){ //insertion operation in skip list
    Node* update[maxLevel];
    Node* x = Header;
    int i;
    for(i = Level;i >= 0;i--){ //Search -- record last updated node in each
level of list
        while(x->Next[i]->searchKey < key){
            x = x->Next[i];
        }
        update[i] = x;
    }
    x = x->Next[0];
    if(x->searchKey == key) return; //do nothing --
    else{
        int v = RandomLevel();
        if(v > Level){
            for(i = Level+1; i <= v;i++){
                update[i] = Header; //higher than original structure,
set header
            }
            Level = v;
        }
        x = MakeNode(key,v);
        for(i = 0;i <= v;i++){ //update connection
            x->Next[i] = update[i]->Next[i];
            update[i]->Next[i] = x;
        }
    }
}
void skipList::Delete(int key){ //deletion in skip list
    Node* update[maxLevel];
    Node* x = Header;
    int i;
    for(i = Level;i >= 0;i--){ //Search -- record last updated node in
each level of list
        while(x->Next[i]->searchKey < key){

```

```

        x = x->Next[i];          //visit next node
    }
    update[i] = x;              //update information
}
x = x->Next[0];
if(x->searchKey == key){        //Find
    for(i = 0; i <= Level;i++){
        if(update[i]->Next[i] != x)break;    //no key -- do nothing
        update[i]->Next[i] = x->Next[i];    //update connection
    }
    delete x;
}
while(Level > 0 && Header->Next[Level] == NIL){ //adjust the height of
skip list
    Level--;                    //remove blank ones
}
}
int main(){                    //to measure time more accurately
    int N,times;                //define the number of
operations and times of the program execution
    cout<<"Please enter the size of skip list: "<<endl;
    cin>>N;
    cout<<"Please enter the times the operation needs to be executed: "<<endl;
    cin>>times;
    srand( (unsigned)time( NULL ));
    double dur=0;
    clock_t start,end;
    skipList list;
    for(int i = 0;i < N;i++) list.Insert(i);    //construct skip list of
size N
    start = clock();    //start timing
    for(int j=0;j<times;j++)                    //do test many times
    {
        list.Search(rand()%N);                //search random element
        int key=rand()%N;
        list.Insert(key);                    //insert and delete to
ensure the size unchanged
        list.Delete(key);
    }
    end=clock();
    dur = (double)(end - start);
    cout<<endl<<"Running time of single operation on skiplist with size "
<<N<<" is "<<(dur/CLOCKS_PER_SEC/times/3)<<endl; //print out time of single
operation
}

```

## References

---

[1] <https://www.thecrazyprogrammer.com/2014/12/skip-list-data-structure.html>

[2] <https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf>

[3] Pugh W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6): 668-676.

## Author List

---

Kang Mintong (programmer, tester, reporter)

Bao Yifan (programmer, tester, reporter)

Ye Fengshuo (programmer, tester, reporter)

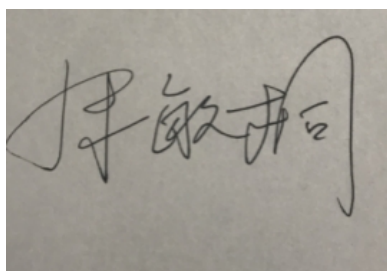
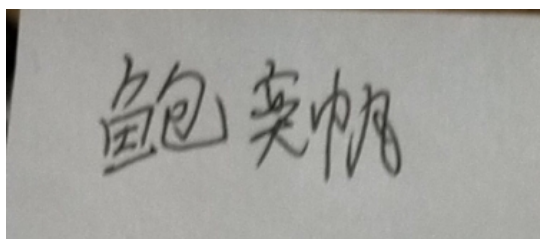
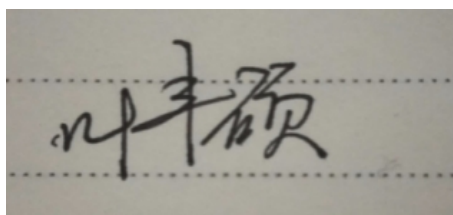
## Declaration

---

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

## Signatures

---

Handwritten signature of Kang Mintong in black ink on a light-colored background.Handwritten signature of Bao Yifan in black ink on a light-colored background.Handwritten signature of Ye Fengshuo in black ink on a light-colored background with horizontal dashed lines.