Project 7 Skip List

Wang Kedi, Shen Yunfeng, Zhao Yilei

Date: 2022/05/27

Chapter 1: Introduction

Skip List is a data structure for fast searching of ordered sequences of elements, introduced by American computer scientist William Pugh in 1989.

If we want to search a key value in ordered linked list, we should visit the linked list sequentially, which would cost **O(N)**.

When using balanced trees such as B-Tree, Red-Black Tree and AVL Tree, the time bound is lower to **O(logN)**, but implementation would be much more difficult.

By applying probabilistic balancing rather than strictly enforced balancing, Skip List is pretty easy to implement and takes only **O(logN)** for insertion, deletion and searching.

In this project, we are supposed to implement the Insertion, Deletion and Searching operations of the Skip List, and then give a formal proof to show the expected time for all three operations is **O(logN)**.

Chapter 2: Data Structure / Algorithm Specification

2.1 Data Structure - Skip List

A Skip List looks like this:

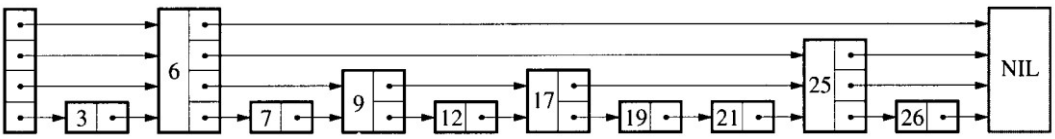


Figure 1. The Skip List

It consists of N(=Max(Level)) linked list, each level are sorted in ascending/descending order.

A node has k forward pointers is called a *level k node*. And thus, we have the definition of a node in Skip List:

```
class node
{
    friend class skiplist;
    protected:
        int key;   //input sequence number
        int val;   //key value
        node* next[MAXLEVEL+1];
    public:
        node(int k, int v) : key(k), val(v) {
            for(int i = 0; i <= MAXLEVEL; i++) next[i] = NULL;
        }
};
```

In order to do the operations conveniently, we then define class skiplist witch consists of nodes and member functions to execute operations we want.

```
class skiplist
{
    protected:
        int level;    //the max_level of current skip list
        node* head;   //the head node
    public:
        skiplist() {};
        skiplist(int l) : level(l), head(NULL) {};
        //only used when init
        void inihead(int k, int v)  {this->head = new node(k, v);}
        void insert(int k, int v);
        void dele(int v);
        void search(int v);
        void print();
};
```

2.2 Algorithm Specification

Because the level of each node is randomly decided, we use function genlevel() to decide which level the node is in.

We would reset the time seed each time we initialize a skip list.

P(node is at ith level) = $1 * \frac{1}{4}^{i-1}$, $i \in [1,\ max\_level],\ i \in \mathbb{N}$.

```
int genlevel()
{
    int level = 1;
    while(rand()%4==0)// P(go to the next level) = 0.25
    {
        level++;
        if(level == MAXLEVEL) return MAXLEVEL;
    }
    return level;
}
```

2.2.1 Searching

To speed up the process of searching, we first visit the top level, visit the linked list sequentially until we find the last node whose value is less than key.

We then go to the lower level and do the same thing, until we reach the base level of the list.

If we want to find key = 17, the process would look like this:

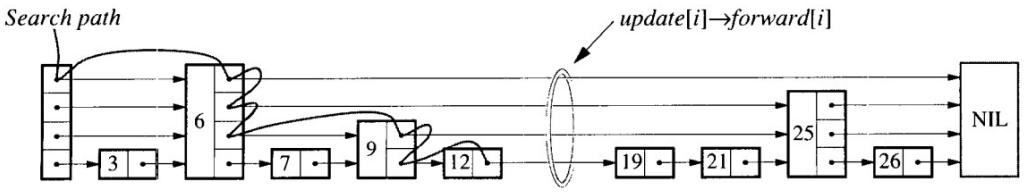


Figure 2. Search 17 in Skip List

When we find the last node whose value is less than key, we would face 2 cases:

- The next node's value is NOT equal to key → the key value is not exist in the list
- The next node's value is equal to key → the key value exist

Pseudocode:

```
skiplist::search(int val)
{
    x := head;
    int i := level;  //max level
    while(i > 0)     //for each level
    {
        while(x->next[i]->val < val)
            x := x->next[i];
        i--;
    }
    x := x->next[1]; //check at base level
    if(x->val == val) then return true;
    else                   return false;
}
```

2.2.2 Insertion

To insert a node, we do searching first to find the right position to place the node.

Array pre[] is used to record the last node whose value is less than key, so that when we reach the base level, we can easily update all the linked list with those reacord.

But when the insertion generates a node with level higher than the current skip list max level, we must update the skip list and initialize the appropriate portions of the update vector.

When we insert key 17 into the skip list, first we do it like Figure 2 shows.

If node 17 has the level bigger than 4 (current max-level) we should update the whole list like Figure 3:

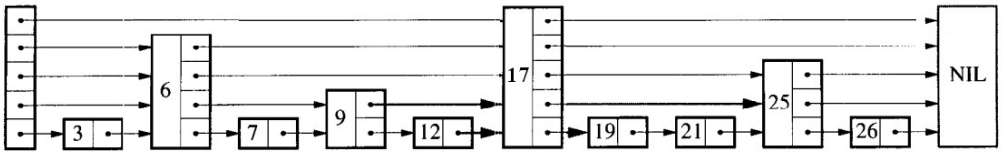


Figure 3. insert 17 with level addition

Pseudocode:

```
skiplist::insert(int k, int v)
{
    // 1 - search position
    x := head;
    int i = level;          //max level
    node *pre[MAXLEVEL+1]; //record the pre-node at each level
    while(i > 0)
    {
        while(x->next[i]->val < val)
            x := x->next[i];
        pre[i] = x;
        i--;
    }
    x := x->next[1];
    // 2 - insert
    if(x->val == val) then return true;
    else //create new node
    {
        int neo_level = genlevel();
        node *tmp = new node(k, v);
        for(int i = 1; i<= neo_level)
        {
            tmp->next[i] := pre[i]->next[i];
            pre[i]->next[i] := tmp;
        }
    }
    return true;
}
```

2.2.3 Deletion

The step of delete operation is similar to insertion, searching its position at first then do the deletion.

After each deletion, we should check whether we delete the maximum element of the list, if so, we should decrease the maximum level of the list.

Figure 4 shows the case when we need to adjust the maximum level:

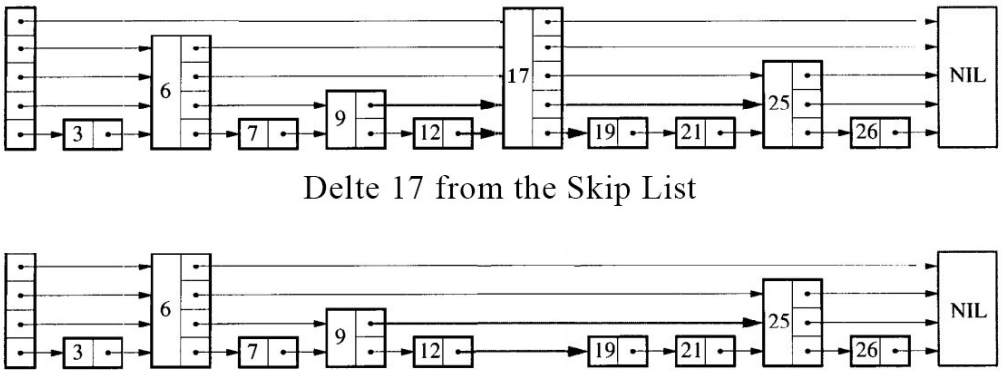


Figure 4. delete 17 with level decreasing

Pseudocode:

```
skiplist::delete(int v)
{
    // 1 - search position
    x := head;
    int i = level;          //max level
    node *pre[MAXLEVEL+1]; //record the pre-node at each level
    while(i > 0)
    {
        while(x->next[i]->val < val)
            x := x->next[i];
        pre[i] = x;
        i--;
    }
    x := x->next[1];
    // 2 - delete
    if(x->val == val) then
    {
        for(int i = level; i > 0; i--)
        {
            if(pre[i]->next[i])
                pre[i]->next[i] = pre[i]->next[i]->next[i];
        }
        return true;
    }
    else                    return false;
}
```

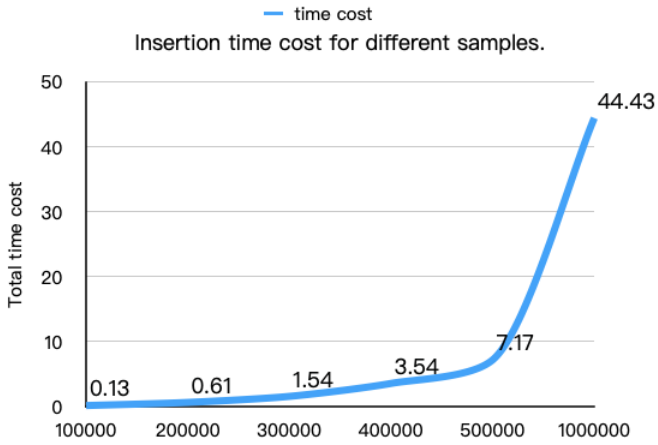
Chapter 3: Testing Results

We have made tests on insertion and deletion of skip lists. In our code, you can define the number of nodes you want to insert or delete. To make the results more specifically, we test 5 large samples.

3.1 Insertion

The insertion has the average time complexity of $O(\log N)$, and have the $O(N)$ time complexity in worst case.

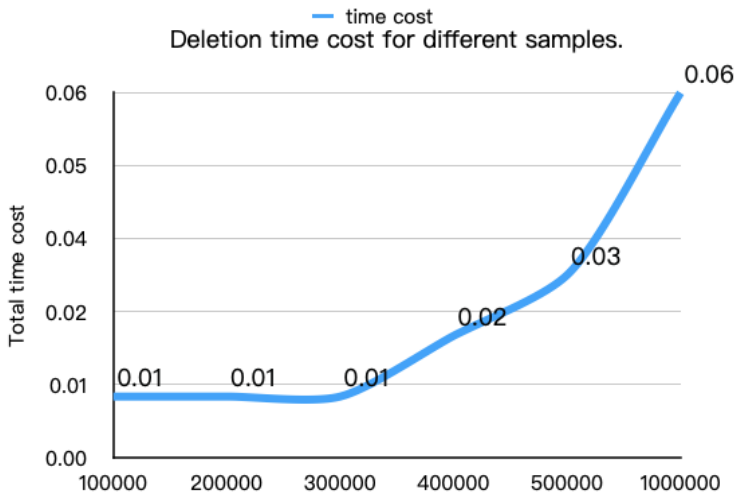
Number of nodes	10^5	2*10^5	3*10^5	4*10^5	5*10^5	10^6
Time cost(s)	0.13	0.61	1.544	3.54	7.169	44.434



3.2 Deletion

The time complexity of deletion is the same as the insertion.

Number of nodes	10^5	2*10^5	3*10^5	4*10^5	5*10^5	10^6
Time cost(s)	0.01	0.01	0.01	0.02	0.03	0.06



3.3 Searching

For skip lists, searching cost a little time. Therefore, we do not test for it specially.

Chapter 4: Analysis and Comments

Time Complexity

Let's think about the worst case at first. Obviously, there exists a worst case when all the nodes are of the same height, and therefore the skip list becomes simply a linked list. Therefore, the search takes `O(n)` time **in the worst case**, and so is insertion and deletion because they both need to find the position to insert into or delete from. However, because the existence of randomized algorithm to decide the height, the probablity becomes very very low, which is `p[1]^n + p[2]^n + ... +`

`p[n]^n`, where `p[i]` is a constant probablity that the element stays in the `ith` level, which is only dependent on `i`. Therefore, the probablity which the worst case takes place obviously converges to 0 because of the randomized algorithm.

Now let's think about the average case. At first, in the randomized algorithm, we generate a random number. If it is a multiple of 4, we add the level by one. Therefore, the chance that a integer is 4's multiple is 0.25, so we decide a constant `p = 0.25` in the randomized algorithm, which is the probablity one element goes to the next level. Therefore, the expected level of a certain element is `1*p + 2*p(1-p) + ... + (i-1)*p(1-p)^(i-2) + ...` which converges to `1/(1-p)`.

Now thinking about how many search we need in a certain level. We know that the probablity that one `ith-level` element becomes `(i+1)th-level` is p. Therefore, in the average case, there is one `(i+1)th-level` element after `1/p` elements. So as expected, there will be `1/p` elements of level i between two elements of level i+1. What's more, we also know that the search in the `ith` level is done between two elements in the `ith+1` level, which means in each level we're expected to do `1/p` steps of searching.

What's more, each 1/p nodes in the ith level, we generate a node of the (i+1)th level. Therefore, suppose the height of the list is h, and `(1/p)^h = C`, where C is a constant. So the skip list contains `O(logn)` elements totally.

Therefore, the expected **average-case time complexity** of search is `O(logn)/p`, which is `O(logn)`.

Insertion and deletion is easy. They only add constant time of additional operations after finding the proper position, which is a search operation. So their average time complexity is `O(logn) + O(1)` = `O(logn)`.

Space Complexity

It seems as if that there's nothing to say about it. There's obviously n nodes to store the input data, each of them containg an array of pointers to the following nodes, whose length is a constant, where the total space complexity is `O(n)`. Taking inserting and deleting into account, we only need to add a constant length array as well, which is `O(1)`. Therefore, the space complexity of the skip list is `O(n)`.

Appendix: Source Code (if required)

```
#include <iostream>
#include <stdlib.h>
#include <cstdlib>
#include <vector>
#include <time.h>

#define MAXLEVEL 5 //Skip_List的最高层数，可以更大
#define MIN(a, b) (a>b)?b:a
#define MAX(a, b) (a>b)?a:b

using namespace std;
```

```
int genlevel();//随机生成层数，向上拓展概率p = 0.25

class node
{
    friend class skiplist;
    protected:
        int key;//插入序号，没啥用
        int val;//数据
        node* next[MAXLEVEL+1];
    public:
        node(int k, int v) : key(k), val(v) {
            for(int i = 0; i <= MAXLEVEL; i++) next[i] = NULL;
        }
};

class skiplist
{
    protected:
        int level;//当前最高层数
        node* head;
    public:
        skiplist() {};
        skiplist(int l) : level(l), head(NULL) {};
        void inihead(int k, int v)  {
            this->head = new node(k, v);
        }
        void insert(int k, int v);
        void dele(int v);
        void search(int v);
        void print();
};

void skiplist::insert(int k, int v)//key是插入，v是值
{
    int l = genlevel();//为新结点随机产生level
    printf("now insert %d %d, level = %d\n", k, v, l);
    this->level = MAX(this->level, l);//更新skip_list的最大level
    node *tmp = new node(k, v);
    node *pre[MAXLEVEL+1], *cur = this->head;

    for(int i = MAXLEVEL; i >= 1; i--)//寻找新结点在每一level的前置结点
    {
        while(cur->next[i]!= NULL && cur->next[i]->val < v) cur = cur-
>next[i];
        pre[i] = cur;
    }

    for(int i = l; i >= 1; i--)//在每一level中插入新结点
    {
        tmp->next[i] = pre[i]->next[i];
        pre[i]->next[i] = tmp;
```

```
    }
    return;
}

void skiplist::dele(int v)
{
    node *pre[MAXLEVEL+1];
    node *cur = this->head;
    //在skip_list中查找v
    for(int i = MAXLEVEL; i >= 1; i--)
    {
        while(cur->next[i] && cur->next[i]->val < v) cur = cur -
>next[i];
        pre[i] = cur;
    }

    if(cur->next[1]==NULL || cur->next[1]->val != v)
    {
        cout << "Delete Failed : " << v << " IS Not Found" << endl;
        return;
    }

    for(int i = MAXLEVEL; i >= 1; i--)
    {
        if(pre[i]->next[i]) pre[i]->next[i] = pre[i]->next[i]->next[i];
    }
    cout << "Delete Success: " << v << " Is Deleted" << endl;
    return;
}

void skiplist::search(int v)
{
    cout << "search result: ";
    node *p = this->head;
    int i = MAXLEVEL;
    while(i>0)
    {
        if(p->next[i])
        {
            while(p->next[i] && p->next[i]->val < v) p = p->next[i] ;
            if(p->next[i]  && v == p->next[i]->val)
            {
                cout << v << " is FOUND" << endl;
                return;
            }
        }
        i--;
    }
    cout << v << " is NOT found" << endl;
    return;
}
```

```
void skiplist::print()
{
    cout << "Print Begin ---" << endl;
    node *q;
    int i = this->level;
    while(i>0)
    {
        if(this->head->next[i])
        {
            q = this->head->next[i];
            cout << "<level " << i << ">: " << endl;
            while(q)
            {
                cout << "[" << q->key << ", " << q->val << "]  ";
                q = q->next[i];
            }
            cout << endl;
        }
        i--;
    }
}

skiplist* ini()
{
    cout << "Initialize Begin ---" << endl;
    skiplist* s0 = new skiplist(1);//ini_level = 1
    s0->inihead(0, 0);
    srand(time(0));
    return s0;
}

int genlevel()
{
    int level = 1;
    while(rand()%4==0)
    {
        level++;
        if(level == MAXLEVEL) return MAXLEVEL;
    }
    return level;
}


int main()
{
    srand((int)time(0));
    double duration;
    clock_t start, stop;
    int n, m;
    cout << "Enter the number of nodes you want to insert:" ;
```

```
    cin >> n;
    cout << "Enter the number of nodes you want to delete:" ;
    cin >> m;

    skiplist* s_list = ini();
    /* test for insertion */
    start = clock();
    for(int i = 1; i < n; i++)
        s_list->insert(i, i+50);
    stop = clock();
    duration  = (double)(stop - start)/CLK_TCK;
    cout << endl;
    cout << "insert cost for "<< n << " nodes is: "<< duration << endl;
    /* test for deletion */
    start = clock();
    for(int i = 60;i < 60+m; i++)
        s_list->dele(i);
    stop = clock();
    duration  = (double)(stop - start)/CLK_TCK;
    cout << endl;
    cout << "deletion cost for "<< m << " nodes is: "<< duration <<
endl;
   // s_list->print();


    s_list->search(80);

    start = clock();
    s_list->search(91);
    stop = clock();
    duration  = (double)(stop - start)/CLK_TCK;
    cout << "search cost is: "<< duration << endl;
    return 0;
}
```

References:

[1] Pugh W . Skip lists: a probabilistic alternative to balanced trees[C]// Workshop on Algorithms & Data Structures. 1990.

Author List:

Name	Task
Wang Kedi	Time bound analysis & Chap 4
Shen Yunfeng	Code of Skip List & Chap1,2
Zhao Yilei	Test & Analysis & Plot & Chap 3

Declarations:

We hereby declare that all the work done in this project titled "pj7" is of out independent effort as a group.

Signatures:

Each author must sign his/her name here: