浙江大学

Advanced Data Structures and Algorithm Analysis

Laboratory Projects

Skip Lists

Group 15

聂俊哲 张琦 张溢弛

Date: 2020-05-22

Content

Chapter 1: Introduction

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.

The report is to introduce the skip list and the problem is how to implement insertion, deletion, and searching in skip lists. We also need a formal proof to show that the expected time for the skip list operations is O(log N). We also generate test cases of different sizes to illustrate the time bound.

Chapter 2: Data Structure / Algorithm Specification

2.1 Description of the Skip List

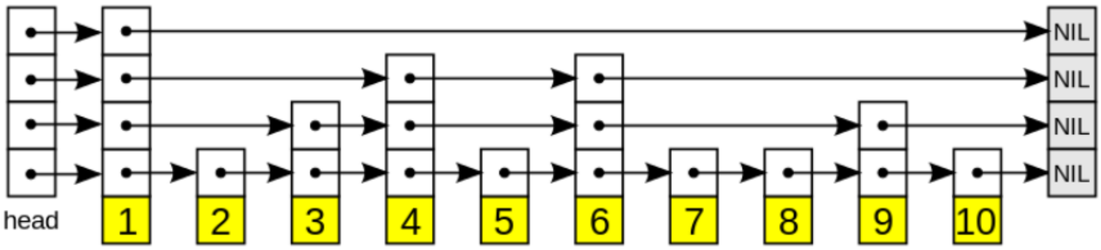
Skip Lists were developed around 1989 by William Pugh[1] of the University of Maryland. Professor Pugh sees Skip Lists as a viable alternative to balanced trees such as AVL trees or to self-adjusting trees such as splay trees. In simple terms, Skip Lists are sorted linked lists with two differences:

1. the nodes in an ordinary list have one 'next' reference. The nodes in a Skip List have many 'next' references (called *forward* references).
2. the number of forward references for a given node is determined probabilistically.

And there are several properties of the skip list:

1. A skip list have several levels.
2. The node in ith level have a probability to be chosen into the next level.
3. The first level of the skip list contain all the element.
4. Every level of the skip list is a normal list.
5. An element in ith level always points to another element that have the same key according *down* pointer.

6. In every level, there is a head node with MIN key and a tail node with MAX key.



An example implementation of a skip list [1]

To implement the skip list, we represent its data structure as following:

- **list node**

  Every node has a value for store, and a sequence to store the node forward.

```
struct SNode
{
    int key;
    SNode *forword[MAXN_LEVEL];
};
```

- **skip list**

  Every list has a head node, and set a level value to show the level.

```
struct SkipList
{
    int nowLevel;
    SNode *head;
};
```

2.2 Description of the Algorithm

This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The Search operation returns the contents of the value associated with the desired key or failure if the key is not present. The Insert operation associates a specified key with a new value (inserting the key if it had not already been present). The Delete operation deletes the specified key.

- **Search**

The input to this function is a search key, `key` . The output of this function is a position, `p`, such that the value at this position is the largest that is less than or equal to `key`.

```
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**

The input for insertion is a `key`. The output is the topmost position, `p`, at which the input is inserted. Note that we are using the `Search` method from above. We use a function called `Random()` that mimics a fair coin and returns either heads or tails. Finally, the function `insertAfter(a, b)` simply inserts the node `a` after the node `b`.

```
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 Random() == 'Tails'
    n = n + 1
    return q
```

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.

- **Deletion**

  Deletion takes advantage of the `Search` operation and is simpler than the `Insertion` operation. We will save space by writing the pseudocode more verbosely.

```
Delete(key)
    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
```

  Delete can be implemented in many ways. 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.

Chapter 3: Testing Results

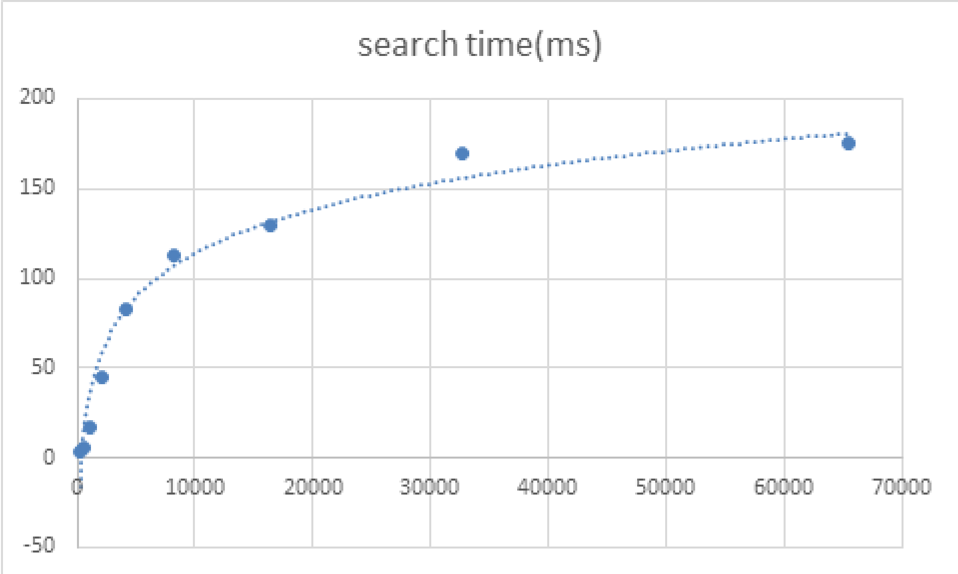
For time complexity test, in this project, we use random method to create a series of test cases. While testing data of small input sizes, we should use different test cases in only one test instance, so we use the random method for several times in one test instance.

We use the power of 2, from 256 to 65536 as the size of data input, testing the time complexity. The test data is including in our appendix. In these test cases, the data is generate randomly so one key may be the same with another key.

Here is the run time table, we test the performance of searching, inserting and deleting, and here is the result:

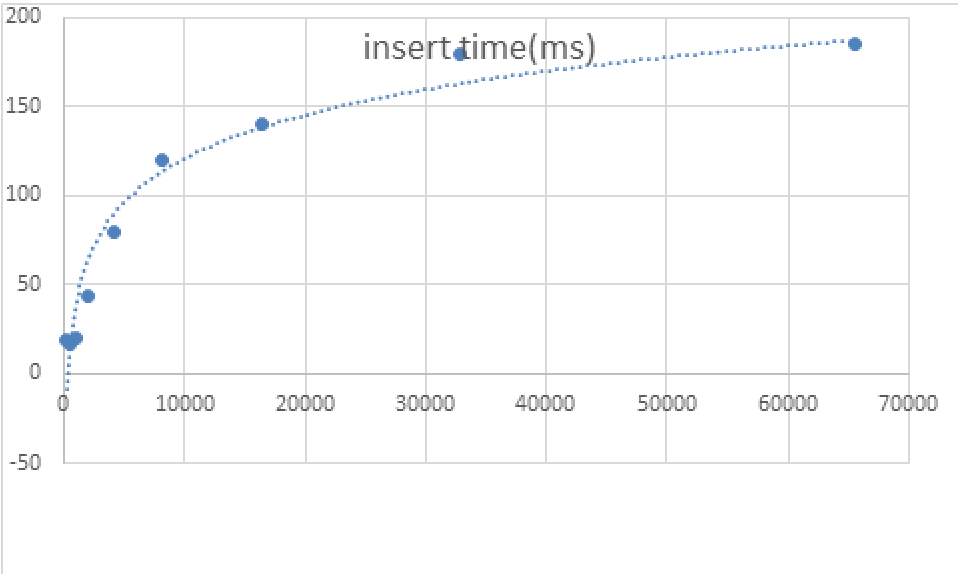
- **Search**

N	256	512	1024	2048	4096	8192	16384	32768	65536
time(ms)	3	5	14	48	83	117	131	167	175



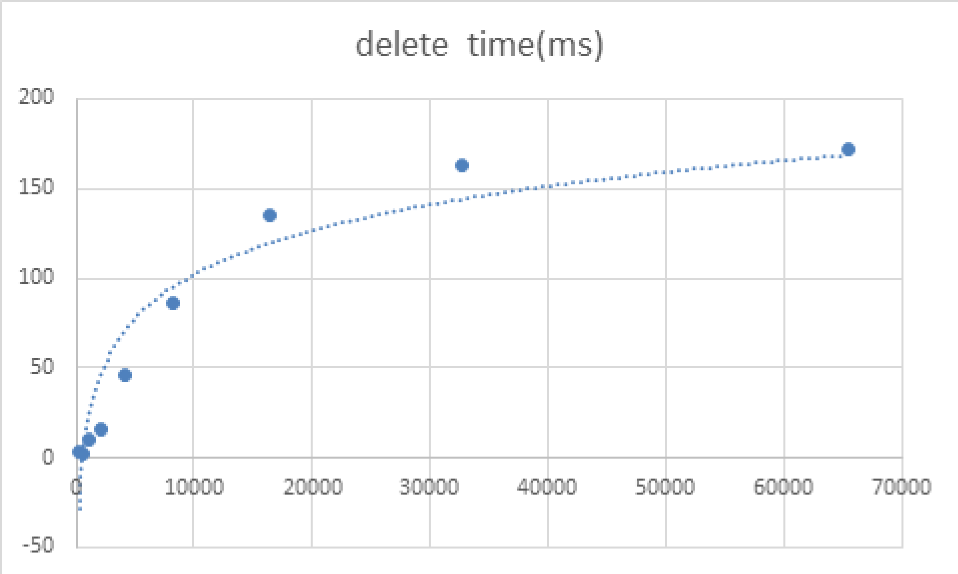
- **Insert**

N	256	512	1024	2048	4096	8192	16384	32768	65536
time(ms)	15	17	20	43	81	124	141	176	185



- **Delete**

N	256	512	1024	2048	4096	8192	16384	32768	65536
time(ms)	3	4	11	14	43	87	134	167	172



As we can see, the time complexity of all there operations is approximately O(log n).

Chapter 4: Analysis and Comments

4.1 Time complexity

The time required to execute the Search, Delete and Insert operations is dominated by the time required to search for appropriate element. For the Insert and Delete operations, there is an additional cost proportional 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 appear as we traverse the list.

4.1.1 Choosing a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level i+1 pointers. We say that a fraction p of the nodes with level i pointers also have level i+1 pointers. (for our original discussion, p = 1/2).

We can get that the probability of a node having a pointer at level 1 is p^0 =1, the probability of having a pointer at level 2 is p^1, and the probability of having a pointer at level L is p^{L-1}.

So, the probability that at *least one* of the n elements gets to level i is

$$E(P_i) = n * p^{i-1}$$

4.1.2 Max Level L(n)

Ideally we expect there are 1/p nodes in level L. Then we have

$$\frac{1}{p} = E(P_i) = n * p^{i-1}$$
$$L = \log_{1/p} n$$

When we are at the i-th forward pointer of a node x and we have no knowledge about the levels of node to the left of x or about the level of x, other than that the level of x must be at least i. The probability that the level of x is equal to i is 1-p, and the probability that the level of x is greater than i is p. Each time the level of x is greater than i, we climb up a level. Let C(L) = the expected cost (i.e, length) of a search path that climbs up L levels in an infinite list:

$$C(0) = 0$$
$$C(L) = (1-p)(cost\ in\ situation\ b) + p(cost\ in\ situation\ c)$$

By substituting and simplifying, we get:

$$C(L) = (1-p)(1 + C(L)) + p(1 + C(L-1))$$
$$C(L) = 1/p + C(L-1)$$
$$C(L) = L/p$$

So total expected cost to climb out of a list of n elements <= L(n)/p+1 , which is O(log n).

4.2 Space Complexity

Space is a little easier to reason about. Suppose we have the total number of positions in our skip list equal to

$$n \sum_{i=0}^{h} \frac{1}{2^i}$$

That is equal to

$$n * (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) = 2n$$

because of the infinite summation. Therefore, our expected space utilization is simply

$$Space - O(n)$$

Appendix:

Source Code

```
#include<iostream>
#include<ctime>
using namespace std;

/* you can change the MAXN_LEVEL and the PRABABILITY */
const int PRABABILITY = 0.5;
const int MAXN_LEVEL = 10;

/* list node */
struct SNode
{
    int key;
    SNode *forword[MAXN_LEVEL];
};

/* skip list */
struct SkipList
{
    int nowLevel;
    SNode *head;
};

/*************************************************
Parameter: max is the highest level
Function: generate random level
*************************************************/
int Random(int max)
{
    int r = 0;    //every level has probability p to ascend upwards

    while(rand() % 100 < 100 * PRABABILITY)
    {
        r++;
        if(r >= max)
        return max;
    }
    return r;
}

/*************************************************
Parameter: myList is a pointer to the head of the jump table
Function: Initialize jump table
*************************************************/
void InitSkipList(SkipList *& myList)
{
```

```
    myList=new SkipList;
    myList->nowLevel=0;
    myList->head=new SNode;
    for(int i=0;i<MAXN_LEVEL;i++)
        myList->head->forword[i]=NULL;
}

/**************************************************
Parameter: myList is a pointer to the head of the skip list, x is the element to be inserted
Function: Insert element x in skip list
**************************************************/
bool InsertSkipList(SkipList *myList,int val)
{
    if(NULL==myList)
        return false;

    int k=myList->nowLevel;
    SNode *q,*p=myList->head;
    SNode *upDateNode[MAXN_LEVEL];  // save the next node of different levels

    while(k>=0)
    {
        q=p->forword[k];
        while(NULL!=q&&q->key<val)  //scan forward
        {
            p=q;
            q=p->forword[k];
        }

        // if the element x had been inserted in the skip list
        if(NULL!=q&&q->key==val)
            return false;
        upDateNode[k]=p;  // save the next node of level k
        --k;
    }

    // Generate a random level value between 1 ~ MAX_LEVEL as the level of the node
    k=Random(MAXN_LEVEL-1);

    // if it is higher than present level of the list, head pointers should also be updated
    if(k>myList->nowLevel)
    {
        k=++myList->nowLevel;
        upDateNode[k]=myList->head;
    }

    p=new SNode;
    p->key=val;

    // update the list of the closest nodes to x in each level, which is below x's level
    for(int i=0;i<=k;i++)
    {
        q=upDateNode[i];
        p->forword[i]=q->forword[i];
        q->forword[i]=p;
    }
```

```
    //for(int i=k+1;i<=myList->nowLevel;i++)
    for(int i=k+1;i<MAXN_LEVEL;i++)
        p->forword[i]=NULL;
    return true;
}

/*************************************************
Parameter: myList is a pointer to the head of the skip list,
    x is the element to be searched, countRet is the number of search.
Function: Find if there is an element x in the skip list.
*************************************************/
SNode* FindSkipList(SkipList * myList,int val)
{
    if(myList==NULL)
        return NULL;

    int k=myList->nowLevel;
    SNode *q,*p=myList->head;

    while(k>=0)
    {
        q=p->forword[k];
        while(NULL!=q&&q->key<val)  //scan forward
        {
            p=q;
            q=p->forword[k];
        }
        if(NULL!=q&&q->key==val)  //if we find the element
            return q;
        --k;    //scan down
    }
    return NULL;
}

/*************************************************
Parameter: myList is a pointer to the head of the skip list, x is the element to be deleted
Function: delete element x in skip list
*************************************************/
bool DeleteSkipList(SkipList * myList,int val)
{
    SNode *ret=FindSkipList(myList,val);
    if(NULL==ret)
        return false;

    int k=myList->nowLevel;
    SNode *q,*p=myList->head;
    SNode *upDateNode[MAXN_LEVEL];  // save the next node of different levels

    for(int i=0;i<MAXN_LEVEL;i++)
        upDateNode[i]=NULL;

    while(k>=0)
    {
        q=p->forword[k];
        while(NULL!=q&&q->key<val)  //scan forward
        {
            p=q;
            q=p->forword[k];
```

```
        }
        if(NULL!=q&&q->key==val)
            upDateNode[k]=p;  // save the next node of level k
        --k;
    }

    //next is the exactly node to delete, update the pointers in each level, which is below x's level
    for(int i=0;i<=myList->nowLevel;i++)
    {
        q=upDateNode[i];
        if(NULL!=q&&q->forword[i]==ret)
            q->forword[i]=ret->forword[i];
    }
    delete ret; //delete the node x
    return true;
}

int main()
{
    int N;
    int val[100000];
    SkipList *  myList;
    InitSkipList(myList); // initialize the skip list

    scanf("%d", &N);  // input the size of the test data
    float cpu_time_used;
    printf("Size: %d\n", N);
    for(int i=0; i<N; i++)  // input the test data
        cin >> val[i];

    clock_t start, end;
    unsigned long sum = 0;

    /*calculate the time taken for inserted*/
    start = clock();
    for(int i=0; i<N; i++)
    {
        InsertSkipList(myList, val[i]);
    }
    end = clock();
    sum = end - start;
    cpu_time_used = ((double) sum)/CLOCKS_PER_SEC;
    printf("Insert: %lf\n", cpu_time_used);

    /*calculate the time taken for find the element*/
    start = clock();
    for(int i=0; i<N; i++)
    {
        FindSkipList(myList, val[i]);
    }
    end = clock();
    sum = end - start;
    cpu_time_used = ((double) sum)/CLOCKS_PER_SEC;
    printf("Find: %lf\n", cpu_time_used);

    /*calculate the time taken for deleted*/
    start = clock();
    for(int i=0; i<N; i++)
    {
        DeleteSkipList(myList, val[i]);
```

```
    }
    end = clock();
    sum = end - start;
    cpu_time_used = ((double) sum)/CLOCKS_PER_SEC;
    printf("Delete: %lf\n", cpu_time_used);

    system("Pause");
}


```

Test Code

```
#include<fstream>
#include<iostream>
#include<ctime>
using namespace std;
int main(){
    int N,var;
    ofstream OutputFile;                 //Define the output file
    cin >> N;
    srand((int)time(0));
    OutputFile.open("d:\\myfile.txt"); //open the file
    for(int i = 0; i < N;i++){
        var =rand() % (1000001);       //generate a number between 0 and 1000001
        OutputFile<<var<<endl;
    }
    OutputFile.close();                  //close the file
    return 0;
 }
```

References

[1] Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Skip_listc++

[2] William Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", 1990

Author List

- Zhang Yichi 3180103772
- Zhang Qi 3180103162
- Nie Junzhe 3180103501

Declaration

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

Signatures

聂俊哲

張琦

张溢驰