Advanced Data Structures and Algorithm Analysis

Project6 Skip Lists
Group 25
Date:2020-05-29

Chapter 1: Introduction

1. Problem Description

Similar to balanced tree structure, *Skip list* is a data structure that supports searching ,insertion and deletion in $O(logN)$ expected time. But *skip list* bases on list stucture, to improve efficiency, the algorithm design many lists like subway routes. We always search on the fastest route which is possible to get the destination.

In this project we **introduced the skip lists**, and implement **insertion, deletion, and searching** in skip lists. **A formal proof** was given to show that the expected time for the skip list operations is $O(logN)$. We also **generated test cases** of different sizes to illustrate the time bound.

Balancing a data structure probabilistically is easier than explicitly maintaining the balance, which simplifies the implementation and reduces the space cost.

2. Main Idea

When a item inserting into skip lists, we keep tossing a coin until getting TAIL. And the number of HEADs will be the level of the item. Items with different levels construct the skip lists.

To search an item, we begin at the top level and scan-forward until the ***key*** of the next item is bigger. Then we go to next level and scan-forward until we found the item.

3. Our Work

We implemented the algorithm and verified the correctness and time complexity in both theory and practice.

Chapter 2: Algorithm Specification

Our code can successfully solve the problem. And we are going to specify it with the following aspect.

1. *Skip Lists* General Introduction

Skip lists are a probabilistic alternative to balanced trees, which are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, they have great average-case performance. This will be shown in the following proofs.

When a item inserting into skip lists, we keep tossing a coin until getting TAIL. And the number of HEADs will be the level of the item. Items with different levels construct the skip lists.

2. Sketch of the Program

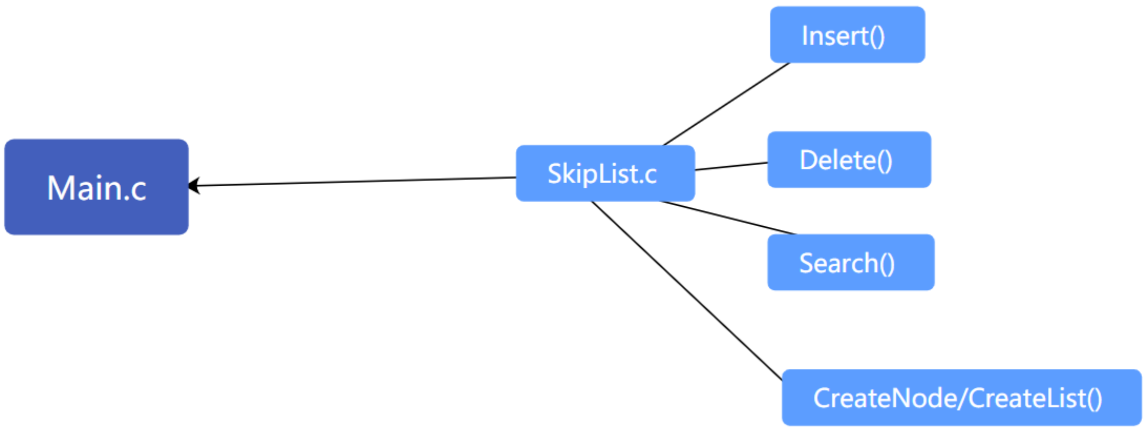


Figure 1

Figure 1 lists the main functions and the program structure.

The data structures are as follows:

```
typedef struct SkipNode* PtrtoNode;
struct SkipNode
{
    KeyType key;        // key: act as index
    ValType value;      // value: act as storage
    PtrtoNode forward[];// Dynamic array
};

typedef struct SkipList *List;
struct SkipList
{
    int level;          // Current level
    PtrtoNode head;     // Beginning
};
```

$SkipNode$:Each level is arranged according to key. And the massege is stored in $value$. $forward[i]$ points at the next node at level i.

$SkipList$:$level$ means the max level of the skip lists. $head$ is the head node.

3. Basic Operation

- **Initialization**

  Create a head node with a key less than any legal key. It has max number of levels.

  Create a tail node(NIL).

```
CreateList()
    create a sklist;
    create a headnode;
    sklist->head = headnode;
    for i = 0 to MAXLEVEL do
        link headnode to NIL
    return sklist;
```

- ***Search***

  Beginning at highest level, traverse forward pointers that do not overshoot the node containing the key being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level.

```
Search(sklist, searchKey)
    current begin at head on sklist->level;
    while not found && next is not NIL on level 0
        if next->key == searchKey
            return next;                // Found
        if next->key < searchKey
            current = next;
        else Go to Next Level;
    return INF;                         // Not Found
```

- ***Insertion***

  Use a method like ***Search*** to find the position to insert, and store the nodes might be updated. If the key is in the *skip lists* then just update the value. If it is a new key, we should create a new node(with random level) and insert it into the list.

  Function ***RandomLevel()*** return a integer x with a probability of p^{-x}. $(x <= MaxLevel)$ We choose $p = 2$.

```
Insert(sklist, searchKey, newValue)
    current = sklist->head;
    next = current->next;
    for i = sklist->level downto 0
        while(next is not NIL && next->key < searchKey)
            current = next;
        update[i] = current;    // Store the nodes that may need to change

    if (next is not NIL && next->key == searchKey)
        next->value = newValue;     // Update value
        return;
    newlevel = RandomLevel();
    if(newlevel > sklist->level)
        for i = sklist->level+1 to newlevel do
            update[i] = sklist->head;
        sklist->level = newlevel;
    create a newnode
    for i = 0 to newlevel do
        newnode->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = newnode;
    return;
```

- ***Deletion***

  Similar to ***Insertion***.

```
Delete(sklist, key)
    update[MAXLEVEL];
    current = sklist->head;
    next = current->next;

    for i = sklist->level downto 0 do:
        while(next is not NIL && next->key < key)
            current = next;
        update[i] = current;    // Store the nodes that may need to change

    if (next is NIL || next->key != key)
        return 0;               // Not found

    for i = sklist->level downto 0 do
        if (update[i]->forward[i] == next)
            update[i]->forward[i] = next->forward[i];
            if(sklist->head->forward[i] is NIL)
                sklist->level--;

    return 1;                   // Found and deleted
```

4. Picture Presentation

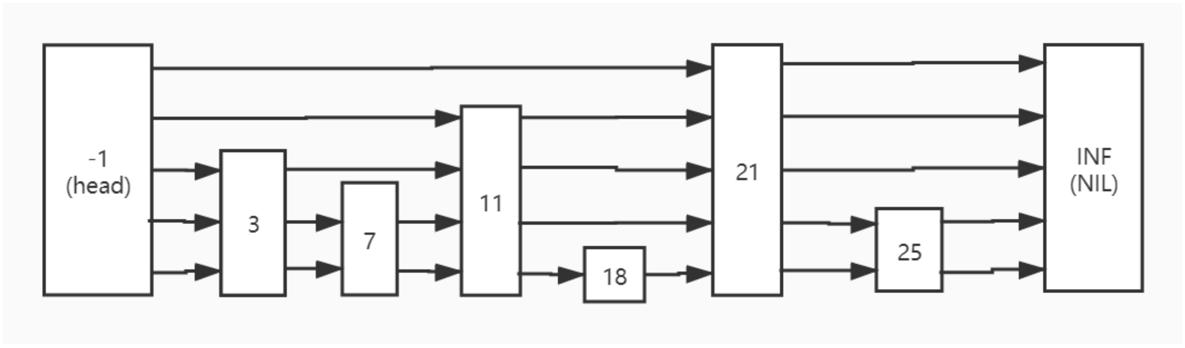


Figure 2 Origin List

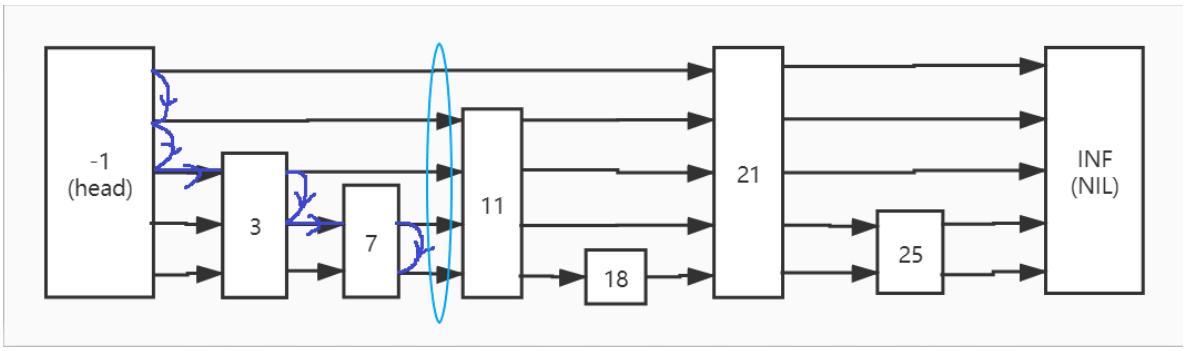


Figure 3 Search for Insert key = 10

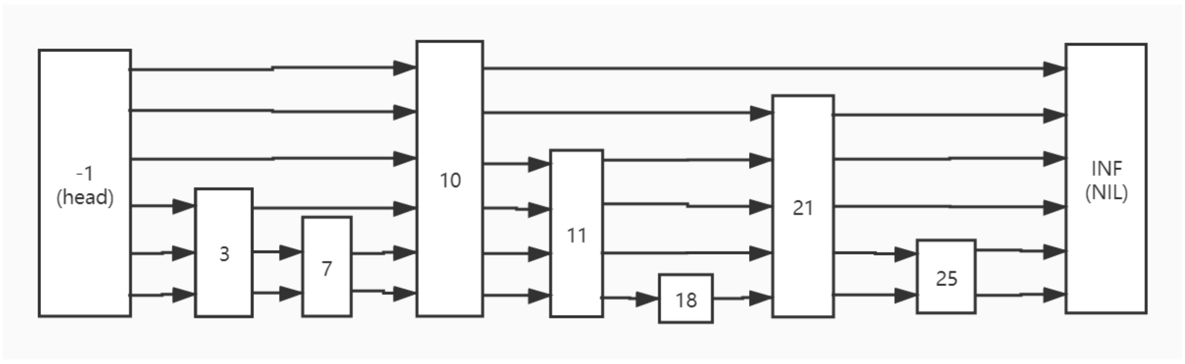


Figure 4 Inserte key = 10 with level = 6

Delete key = 10 from Figure 4, we get Figure 3.

5. Correctness Proof

Items must be in the bottom level, and we will go to next level if the next key is too large. So we can always find them.

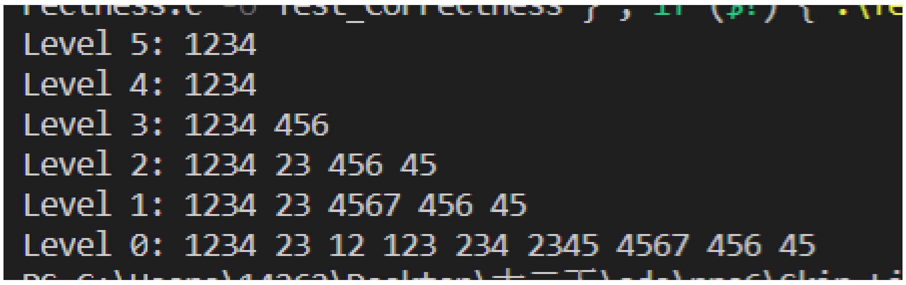
Chapter 3: Testing Results

1. Test for correctness

Program: *Test_Correctness.c*
Input(data.txt):

```
9
3 12
4 123
1 1234
2 23
5 234
6 2345
9 45
8 456
7 4567
```

Output:



```
rectness.c -o Test_Correctness } ; if ($?) { .\Te
Level 5: 1234
Level 4: 1234
Level 3: 1234 456
Level 2: 1234 23 456 45
Level 1: 1234 23 4567 456 45
Level 0: 1234 23 12 123 234 2345 4567 456 45
[illegible]
```

Conclusion: The output meet the expectation of a typical skip list, thus confirm the correctness of the skip list.

Purpose:
The test case here is formed by a typical input with rather small N.
After implementing the insertion algorithm, the result skip list would be displayed.
And we'll compare the result skip list with all the possible skip list that we draw manually to testify the correctness of the program.

2. Time testing

Purpose:

The test cases here aims to measure the time complexity of search operation.
Since the search operation is the most commonly used operation, we'll cover the most of the potentially-used input size(from 100 to 200000).
To reduce error, we make repetition large enough, typically when input size is small, so the general time would be more than 2 seconds.

case	Input Size	Data	Repetition	Total time(s)	Average Time($\times 10^{-6} s$)
0	100	data0.txt	100000000	3.35100	0.03351
1	1000	data1.txt	50000000	5.97900	0.11958
2	3000	data2.txt	30000000	4.66100	0.15537
3	5000	data3.txt	20000000	3.46500	0.17325
4	8000	data4.txt	16000000	3.12500	0.19531
5	10000	data5.txt	10000000	2.13400	0.21340
6	20000	data6.txt	10000000	2.40200	0.24020
7	50000	data7.txt	10000000	2.81400	0.28140
8	100000	data8.txt	10000000	3.28500	0.32850
9	200000	data9.txt	10000000	3.64500	0.36450

Figure 5 Time testing for Search operation

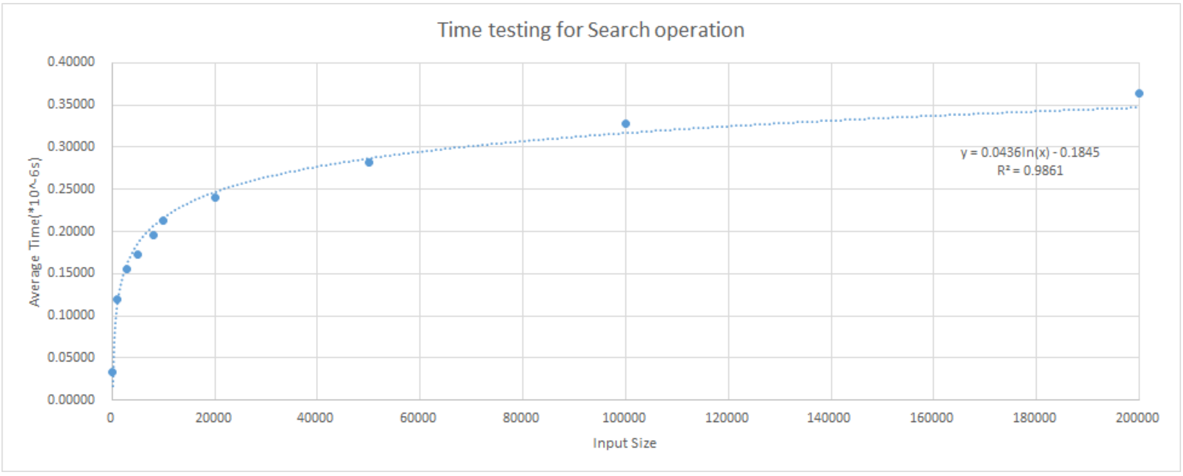


Figure 6 Time testing for Search operation

Purpose:

The test cases here aims to measure the time complexity of insertion and deletion.

Since the repeated insertion or deletion doesn't make any sense, and the small N could only cost extremely little time, we use rather large scale of input size(from 100000 to 10000000).

case	Input Size	Repetition	Total Time	Average Time($\times 10^{-6} s$)
0	100000	100000	0.037000	0.370000
1	500000	500000	0.365000	0.730000
2	1000000	1000000	0.936000	0.936000
3	3000000	3000000	3.559000	1.186333
4	5000000	5000000	6.406000	1.281200
5	8000000	8000000	11.088000	1.386000
6	10000000	10000000	15.165000	1.516500

Figure 7 Time testing for Insertion

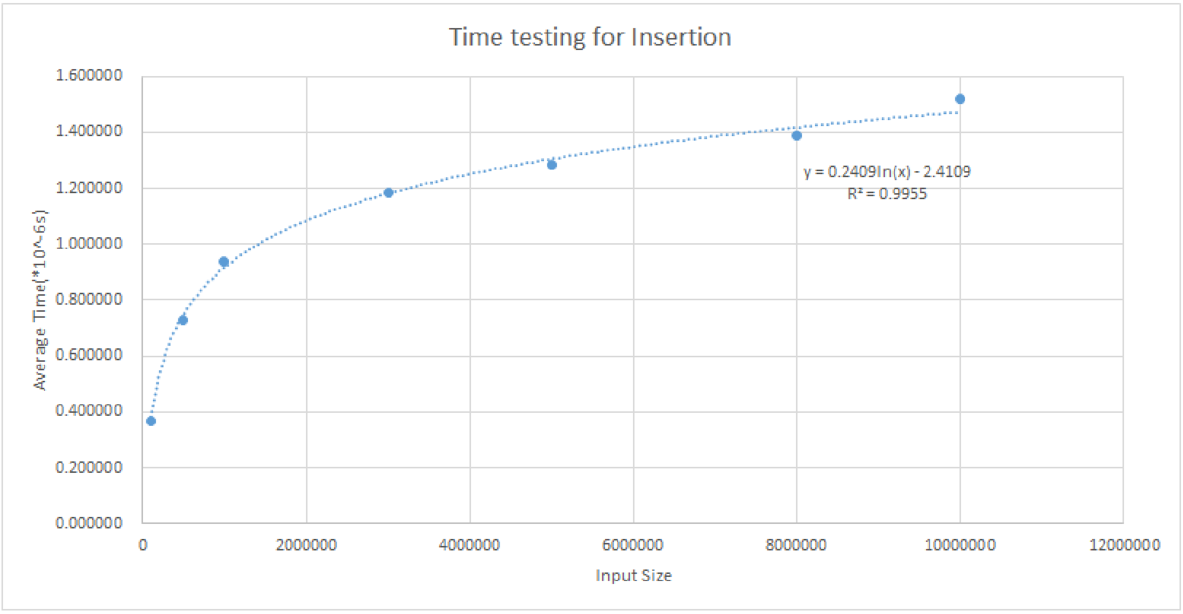


Figure 8 Time testing for Insertion

case	Input Size	Repetition	Total Time	Average Time(*10^-6s)
0	100000	100000	0.030000	0.300000
1	500000	500000	0.208000	0.416000
2	1000000	1000000	0.523000	0.523000
3	3000000	3000000	2.332000	0.777333
4	5000000	5000000	4.377000	0.875400
5	8000000	8000000	7.801000	0.975125
6	10000000	10000000	10.275000	1.027500

Figure 9 Time testing for Deletion

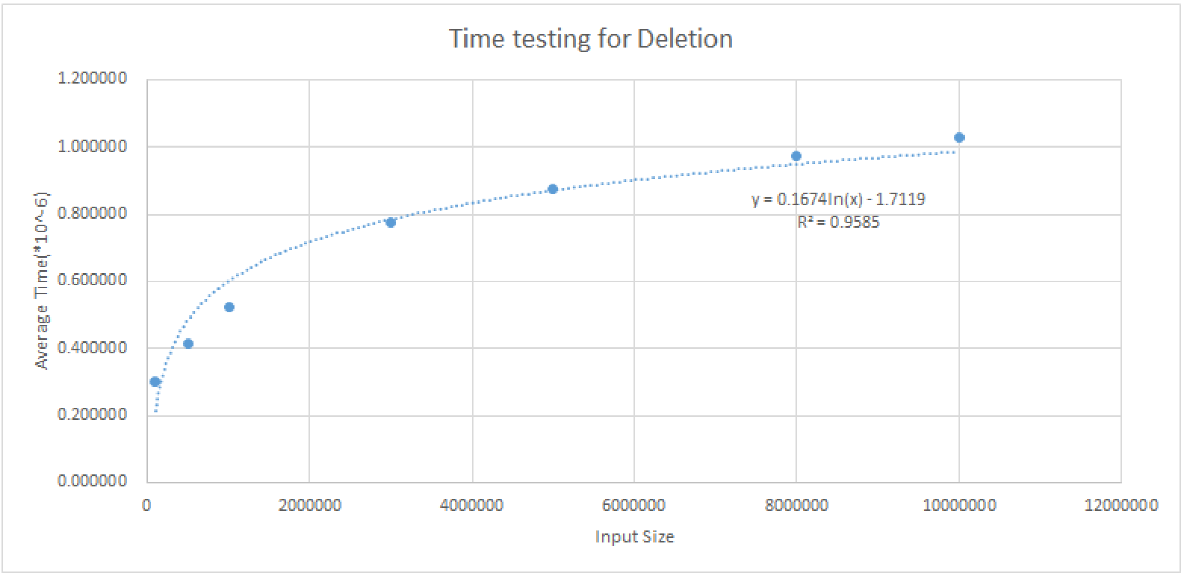


Figure 10 Time testing for Deletion

Note: The time testing for Insertion and Deletion cannot base on large amount of repetition if the Input Size is small. Consequently, we choose rather large data to estimate the time complexity, the txt file could be too large so we'll not upload it.

Conclusion:

Time testing shows that the time complexity of three operations is $O(logN)$. And in **Chapter 4**, we will prove it.

Chapter 4: Analysis and Comments

1. Space Complexity: $O(N)$

The space used by the skip list is determined by the insertion algorithm which add nodes randomly.

Since the probability of a certain node staying on level i is 2^{-i} , the total number of node , with the input size of N, would be calculated with the following formula:

$$N\sum_{i=0}^{h}2^{-i}$$

It’s clear that the geometric progression is smaller than 2. Therefore we could conclude that:

$$N\sum_{i=0}^{h}2^{-i} < 2N$$

So the space complexity is O(N).

2. Time complexity

From the Figure 6,8,10, we could see that all three curves of three operations matches with the trend of logarithmic function. Therefore, we could tentatively draw the following conclusion:

Search: $O(logN)$

Insertion: $O(logN)$

Deletion: $O(logN)$

- **With High Probability (w.h.p)**
  - Parameterized event E_α occurs **with high probability** if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which E_α occurs with probability at least $1 - c_\alpha/n^\alpha$
- **We’ll prove a lemma first: The height of a skip list is $O(logN)$ with high probability.**
  - *Fact 1: If each of n events has probability p, the probability that at least one event occurs is at most np*
  - Because $P(h_x \geq a) = 2^{-a}$, the probability of level a having node is at most $2^{-a}n$, according to *Fact 1*.
  - Pick $a = clog_2n, (c \geq 3)$. The probability of level a having node is at most n^{1-c}
  - A skip list with n item has height at most $clog_2n$ with probability at least $1 - n^{1-c}$.
  - So height of skip list is $O(logN)$ with high probability.
- **The Time Complexity of Search is $O(logN)$ with high probability**
  - Analyze search backwards(from destination to root).
  - Search starts[ends] at destination.
  - At each node visited:
    - If node wasn't promoted higher, then we go left.
    - If node was promoted higher, then we go up.
  - Search stops at the root.
  - Number of 'up' moves $<$ heights of skip list $= O(logn)$ with high probability. So with high probability, total number of moves is at most the number of times we need to flip a conin to get $clog_2n$ HEADs.

- **Claim**: Number of coin flips until $clogn$ HEADs = $\Theta(logn)$with high probability.
    - **Proof**: Obviously $\Omega(logn)$:at least $clogn$.

      Assume we make $10clog_2n$ flips. Set $A=$ {exactly $clog_2n$ HEADs}. $B=${at most $clog_2n$ HEADs}

      $P(A)=\binom{10clog_2n}{clog_2n}(\frac{1}{2})^{clog_2n}(\frac{1}{2})^{9clog_2n}$

      $P(B)\le\binom{10clog_2n}{clog_2n}(\frac{1}{2})^{9clog_2n}$

      Use bounds on $\binom{y}{x}$:$(\frac{y}{x})^x\le\binom{y}{x}\le(e\frac{y}{x})^x$

      $P(B)\le(e\frac{10clog_2n}{clog_2n})^{clog_2n}(\frac{1}{2})^{9clog_2n}$

      $=(10e)^{clog_2n}2^{-9clog_2n}$

      $=2^{[log_2(10e)-9]clog_2n}$

      $=1/n^{\alpha}$ for $\alpha=[9-log_2(10e)]c$

      Key Property:$\alpha\to\infty$ as $10\to\infty$, for any c.

      So set 10,i.e., constant in $O(logn)$ bound, large enough to meet desired α.
  - The time complexity of search is $O(logN)$.
- **The Time Complexity of Insertion is $O(logN)$ with high probability**
  - The insertion process is search operation + inserting items.
  - Each level of skip list is equal to a normal linked list. The time cost for a insertion of a linked list is $O(1)$. Since the height is $O(logN)$ with high probability, inserting an item itself requires inserting $O(logN)$ items which costs $O(logN)$ time.
  - And the search operation itself also costs $O(logN)$ time.
  - Consequently, the time complexity is $O(logN)+O(logN)=O(logN)$
- **The Time Complexity of Deletion is $O(logN)$ with high probability**
  - The deletion process is search operation + deleting items. It's almost the same as Insertion. We could also conclude that the time complexity is $O(logN)+O(logN)=O(logN)$

3. Comment

Although both of insertion and deletion are with the time-bound O(logN), from the testing data, we could observe that the deletion is much quicker than insertion. This might resulted from the function Randomize(), since the randomization itself requires a certain amount of time, it makes insertion much slower, typically when the input size is large. Therefore, the time cost for randomization itself should not be ignored when the data structure is applied into practices.

In this project, we apply the idea of skip list. And the time testing clearly reveal the time bound for searching is O(logN). Generally speaking, the skip list enables the linked list to perform a relatively similar idea of binary search. However, it also comes with the uncertainty of the data structure which makes it difficult to examine and debug if we apply the data structure to real project. Yet, Google's LevelDB, which is a database based on LSM tree, applies skip list as an internal data structure for the MemTable considering its high efficiency in insertion, selection and deletion.

Appendix

1. References

Pugh W . Skip Lists: A Probabilistic Alternative to Balanced Trees[C]// Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings. 1989.

2. Declaration

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

3. Duty Assignments

Programmer: Implement Skip list algorithm. Write a test of performance program. All the codes must be sufficiently commented. **王子腾**

Tester: Provide the necessary inputs for testing and give the run time table. Plot the run times vs. input sizes for illustration. Write analysis and comments. **李想**

Report Writer: Chapter 1, Chapter 2, and finally a complete report which forms overall style of documentation. **张童晨**

4. Code

SkipList.h

```c
/**
 * FileName: SkipList.h
 * Author: Ziteng Wang
 * Description:
 *   1. Type Assignment
 *   2. Data Structure Definition
 *   3. Functions Declaration
 * Date:2020-05-19
*/

#ifndef SkipList__H
#define SkipList__H

#define MAXLEVEL 16     // 2^16 nodes approximately
#define NIL NULL        // Tail node
#define INF 0xffffffff  // INFINITY

typedef int KeyType;    // Key is of integer typoe
typedef int ValType;    // So is Value
typedef int bool;       // Logic Type

// Node Definition
typedef struct SkipNode* PtrtoNode;
struct SkipNode
{
    KeyType key;        // key: act as index
    ValType value;      // value: act as storage
    PtrtoNode forward[];// Dynamic array
};
```

```
// List Definition
typedef struct SkipList *List;
struct SkipList
{
    int level;          // Current level
    PtrtoNode head;     // Beginning
};

// Function Declaration

// Create type
List CreateList();
PtrtoNode CreateNode(int level, KeyType key, ValType value);
// Operation
ValType Search(List skl, KeyType key);
bool Insert(List skl, KeyType key, ValType value);
bool Delete(List skl, KeyType key);
// Assistance
int RandomLevel();
void FreeList(List skl);


#endif
```

SkipList.c

```
/**
 * FileName: SkipList.c
 * Author: Ziteng Wang
 * Description: Functions Implement
 * Date:2020-05-19
*/

#include"SkipList.h"
// #include<stdlib.h>

/**
 * Create and Return a SkipList
 * @return [List][result skiplist]
*/
List CreateList()
{
    List sklist = (List)malloc(sizeof(struct SkipList));    // Memory Allocate
    if(!sklist)     // No Space Allocated for SkipList
        return NULL;

    sklist->level = 0;      // Begin with Empty Node

    PtrtoNode h = CreateNode(MAXLEVEL, -1, -1);     // Head
    if(!h)          // No Space Allocated for Head Node
    {
        free(sklist);       // Release before return
        return NULL;
    }
    sklist->head = h;

    for(int i = 0; i <= MAXLEVEL; i++)
```

```
        h->forward[i] = NIL;    // Connect to Tail

    return sklist;
}



/**
 * Create a Node with given information
 * @param [int][newNode level]
 * @param [KeyType][Assigned Key]
 * @param [ValType][Assigned Value]
 * @return [PtrtoNode][result node]
*/
PtrtoNode CreateNode(int level, KeyType key, ValType value)
{
    // Allocate 'level' units space for Dynamically Array
    PtrtoNode node = (PtrtoNode)malloc(sizeof(struct SkipNode) + (level+1) *
sizeof(PtrtoNode));
    if(!node)           // Allocation Error
        return NULL;

    node->key = key;    // Key & Value assignment
    node->value = value;

    for(int i = 0; i <= level; i++)
        node->forward[i] = NIL;     // Connect to Tail

    return node;
}


/**
 * Search by Key target and return stored Value
 * @param [List][SkipList to be searched]
 * @param [KeyType][target Key]
 * @return [Valtype][searched Value]
*/
ValType Search(List skl, KeyType key)
{
    PtrtoNode current, next;    // Pointers to current and next Node
    current = skl->head;        // Assignment
    next = NULL;

    /**
     * for: from 'top' to 'bottom' level
     * while: moving along one level
     *        break when meeting stop sign
     * return: find the target
    */
    for (int i = skl->level; i >= 0; i--)
    {
        while((next = current->forward[i]) && next->key < key)
            current = next;
        if(next && next->key==key)      // Not end but find
            return next->value;
    }
    return INF;             // Not Found
```

```
}


/**
 * Insert a Node with given information
 * @param [List][SkipList to be inserted]
 * @param [KeyType][given Key]
 * @param [Valtype][given Value]
 * @return [bool][Success or Fail]
*/
bool Insert(List skl, KeyType key, ValType value)
{
    PtrtoNode update[MAXLEVEL]; // Pointers saving update node
                                // [Index] corresponds to each level
    PtrtoNode current, next;    // Pointers to current and next Node
    current = skl->head;        // Assignment
    next = NULL;

    // Similar to Search
    for (int i = skl->level; i >= 0; i--)
    {
        while((next = current->forward[i]) && next->key < key)
            current = next;
        update[i] = current;    // Store the nodes that may need to change
    }

    // Find existing key
    if (next&&next->key == key)
    {
        next->value = value;    // Update value
        return 1;               // Success
    }

    // Generate a randomized level
    int newlevel = RandomLevel();

    // Beyond current highest level
    if(newlevel > skl->level)
    {
        /**
         * for: from current-top to new-top
         *  expand update[]
        */
        for (int i = skl->level+1; i <= newlevel; i++)
        {
            update[i] = skl->head;
        }
        skl->level = newlevel;  // Update current level
    }

    // Create Node
    PtrtoNode newnode = CreateNode(newlevel, key, value);
    if(!newnode)    // Space Allocated Error
        return 0;

    // Update nodes needed to be updated
    // i: level
    for (int i = 0; i <= newlevel; i++)
    {
        newnode->forward[i] = update[i]->forward[i];    // Insert
        update[i]->forward[i] = newnode;
    }

    return 1;       // Success
}


/**
 * Delete a Node with given target
 * @param [List][SkipList to be operated]
 * @param [KeyType][given Key]
 * @return [bool][Success of Fail]
*/
bool Delete(List skl, KeyType key)
{
    PtrtoNode update[MAXLEVEL]; // Pointers saving update node
                                // [Index] corresponds to each level
    PtrtoNode current, next;    // Pointers to current and next Node
    current = skl->head;        // Assignment
    next = NULL;

    // Similar to Search
    for (int i = skl->level; i >= 0; i--)
    {
        while((next = current->forward[i]) && next->key < key)
            current = next;
        update[i] = current;    // Store the nodes that may need to change
    }

    // Not Found
    if (!next||(next->key != key))
        return 0;               // Fail

    /**
     * for: from bottom to top
     * outer if: reconnect
     * inner if: update level
    */
    for (int i = skl->level; i >= 0; i--)
        if (update[i]->forward[i] == next)      // existing height
        {
            update[i]->forward[i] = next->forward[i];
            if(skl->head->forward[i] == NIL)    // The only highest Node
                skl->level--;                   // resize level
        }

    free(next);                 // Release Space

    return 1;
}


/**
 * Generate a Randomized level
 * @return [int][randomized level]
*/
```

```c
int RandomLevel()
{
    int level = 0;  // Start from bottom

    while(rand()%2) // 0.5 possibility each time
        level++;    // Go higher

    level = (MAXLEVEL<level) ? MAXLEVEL : level;    // Ceiling process

    return level;
}


/**
 * Insert a Node with given information
 * @param [List][SkipList to be freed]
*/
void FreeList(List skl)
{
    if(!skl)        // No such SkipList
        return;

    PtrtoNode current, next;    // Similar to former
    current = skl->head;
    next = NULL;

    while (current != NIL)      // Haven't met Tail
    {
        next = current->forward[0];
        free(current);          // Release Node
        current = next;
    }

    free(skl);                  // Release SkipList
}
```

Main.c

```c
/**
 * FileName: Main.c
 * Author: Ziteng Wang
 * Description: Main & Test
 * Date:2020-05-19
*/
#define _CRT_SECURE_NO_WARNINGS
#include"SkipList.h"
#include<stdio.h>
#include<time.h>

#define MaxNode 50000   // Max Nodes Number

clock_t start, stop;    // Time Stamps
double duration;        // Time Cost
int Node[MaxNode][2];   // Node[][0]: Key, Node[][1]: Value


/**
```

```
 * Main Integration & Testing part
 * @exception [Input][Node's Key and Value are supposed to be non-negative]
 * @return [int][0: Success -1: Fail]
*/
int main()
{
    srand((unsigned)time(NULL));    // Prepare for randomization

    int N;              // Input scale
    scanf("%d", &N);
    if(N > MaxNode)     // Beyond bound
        return -1;

    for (int i = 0; i < N; i++)
        scanf("%d%d", Node[i], Node[i]+1);  // Read Key&Value(>= 0)

    List skiplist = CreateList();   // Create a SkipList

    /**
     * Functional Test and Speed Performance
     * Order:
     *  1. Insert
     *  2. Search
     *  3. Delete
    */
    // Insertion Test
    start = clock();                // Start timing
    for (size_t i = 0; i < N; i++)
        if(!Insert(skiplist, Node[i][0], Node[i][1]))       // Insertion & Check
            return -1;      //ERROR
    stop = clock();                 // Stop timing
    duration = (double)(stop-start)/CLK_TCK;
    printf("%d Insertions Cost: %lfs\n", N, duration);

    // Search Test
    start = clock();
    for (size_t i = 0; i < N; i++)
        if (Search(skiplist, Node[i][0]) == INF)               // Searching &
Check
            return -1;      //ERROR
    stop = clock();
    duration = (double)(stop-start)/CLK_TCK;
    printf("%d Searches Cost: %lfs\n", N, duration);

    // Deletion Test
    start = clock();
    for (size_t i = 0; i < N; i++)
        Delete(skiplist, Node[i][0]);                       // Deletion
    stop = clock();
    duration = (double)(stop-start)/CLK_TCK;
    printf("%d Deletions Cost: %lfs\n", N, duration);

    system("pause");        // Pause
    return 0;
}
```