BLG553E Algorithm Engineering Course Final Project Report

Şafak TÜLEMEZ

January 2021

1 Introduction

This report aims to give information about the study on SkipList Data Structure. Here we explain modeling, implementation and experiment phases. Modeling phase includes problem definition and solution approach. Implementation phase represents that how the data structure functions are implemented. Finally, we analyze practically and theoretically performance of our functions.

2 Modeling

In this section firstly we will define the problem. Assume that there is an ordered list of integers. We aims to represents this integers in the memory of a computer. For this purpose we can use arrays.

When we define an array, all integers placed into memory sequentially and also we can access any element with index of array in O(1) time. When we wanted to add or delete an element we must find position and shift all elements after this position. This costs O(n) time. If we want to find an element in array it costs O(logn) with binary search.

An other data structure which is used for this is linked list data structure. Each element of linked list includes integer and a pointer that is pointed to the next element. We can access any element traversing from the head of linked list. However, we can add a new element O(1) time with changing pointers. If we want to find an element in linked list it costs O(n) time.

There is a data structure named skiplist data structure for doing this operation in logaritmic time. Here we will give a model for this data structure. Skiplist data structure consist of two or more randomly generated series of lists. A skiplist data structure must be provide conditions can be seen below.

- Base list S_0 stores all integers and also $-\infty$ to $+\infty$ in the data structure.

- For i = 1,...,h-1 and h is total number of levels in skiplist, list Si contains a randomly generated subset of S_{i-1}. Also each list from from $-\infty$ to $+\infty$.

- List S_h is highest list in the skiplist and only consist of $-\infty$ and $+\infty$ elements.

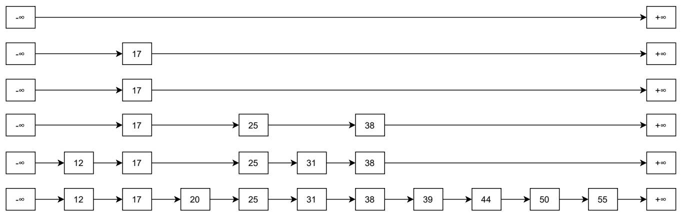

Figure 1: An example skiplist data structure consist of 10 integer elements

In this section we firstly defined our problem. Then we created a model for our data structure represents a solution for this problem.

3 Implementation

In this section we will determine how we will implement this model and what we will use while we are doing this. While we are implementing this data structure we used C++ programming language. This language is an object oriented programming language and high efficient memory access.

A class is defined for skiplist elements and functions. This class consist of height of skip list. This is number of levels in the skiplist. start_pos node pointer is top-left most position node in the skiplist. total_items stores the number of nodes. Functions will be explained on the next sections.

```
class skiplist
{
private:
    /* private variable declerations */
    int height; /* height of the list */
    node *start_pos; /* top and left most position */
    int total_items; /* total node counts */

    /* private function declerations */
    bool randomize();
    node *insert(node *pos, node *q, int key, int value);

public:
    /* public function declerations */
    skiplist();
    ~skiplist();
    node *skip_search(int key);
    node *skip_insert(int key, int value);
    int skip_get_item_index(int key);
    int skip_remove(int value);
    void print(void);
};
```

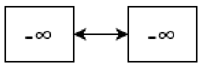

Figure 2: Initialized Skip List

3.1 Node Structure

Nodes are entities in the skiplist data structure. Each node structure stores integer value and four different pointer.

- "above" pointer stores the position of above node.
- "below" pointer stores the position of below node.
- "after" pointer stores the position of after node.
- "before" pointer stores the position of before node.

S_0 list's values stores the integers in skiplist sequentially. Each node's after pointer points to next node that has next bigger integer. Also before pointer points to node that has previous lesser integer. This structure is like a classical doubly linked list. Other lists are random subset of low level list.

Each node's above pointer stores the position of the higher level's node with same value. Also each node's below pointer stores the position of lower level's node with same value as well.

```
struct node
{
    int value;          /* value of node */
    node *below;        /* below node pointer */
    node *above;        /* above node pointer */
    node *before;       /* before node pointer */
    node *after;        /* after node pointer */
};
```

3.2 Initialization

skiplist() construction function is used for initialization of an object. This function create first level. this level only consists of -∞ and +∞ values. After initialization skiplist structure can be shown below.

3.3 Randomization

Randomization is provided by using randomize() function. This function calls the standard C++ library function rand(). this function generates a random integer, then if for odd value returns true otherwise false.

3.4 Printing of Skip List

print() function prints value from the higher level left most position of skip list to the last item of skiplist. Start position is updated every insertion.

3.5 Searching Node in Skip List

Two different search functions are implemented in this study. skip_search() function finds and returns skip list node has value equal or lesser than given key value. Psuodo code can be shown below.

```
Algorithm skip_search(key):
    pos <- start
    while pos.below != NULL:
        pos <- pos.below
        while key >= pos.after.value
            pos <- pos.after
    return pos
```

Another function for searching is skip_get_item_index() function. This function finds firstly trying to find given key value in list S0 . If it is failed then function returns -1. After that, this function finds S0 index value and returns this.

```
Algorithm skip_get_item_index(key):
    index <- 0
    pos <- skip_search(key)
    if pos.value != key:
        return -1
    while pos.before != NULL:
        pos <- pos.before
        index <- index+1
    return index
```

3.6 Insertion Node to Skip List

skip_insert() fonction is used for insertion to the skiplist. Psoudo code of insertion algorihm can be shown below. In addition insert(p,q,key,value) function is used for insertion of a new item after given p and above given q.

```
Algorithm skip_insert(key, value):
    i <- -1
    pos <- skip_search(key)
    q <- NULL
    repeat:
        i <- i+1
        if i >= h:
            h <- h+1
            after_start <- start.after
            start <- insert(NULL, start, -INT_MAX, -INT_MAX)
            insert(start, after_start, +INT_MAX, +INT_MAX)
        q <- insert(pos, q, key, value)
        while pos == pos.above:
            pos <- pos.before
        pos <- pos.above
    until true == randomize()
    return q
```

3.7 Removal from Skip List

Removal function is a trivial function for skip list. skip_remove() function remove the tower owns to given key value. Also it returns -1 when if the value is inappropriate or can not be found.

```
Algorithm skip_remove(value):
    if value == -INT_MAX or +INT_MAX:
        return -1
    pos <- skip_search(key)
    if pos.value != value:
        return -1
    repeat:
        pos.before.after <- pos.after
        pos.after.before <- pos.before
        pos <- pos.above
    until pos != NULL
    return q
```

4 Experimental Results

In this study we analyze insertion and searching algorithm performance. Firstly we made an asymptotic analysis. In worst case, randomize function can always return true and cause infinite loop. This case will continue until running out of memory. Average performance of insertion algorithm are expected O(logn) time. This probabilistic approach explains in [1]. Since the height h of S is O(logn) with high probability, the number of drop-down steps is O(logn) with high probability [1].

Another important performans factor of an algorithm is memory usage. As we said before infinite memory usage can be possible in worst case. Expected space requirement is O(n) for skiplist data structure [1].

After asymptotic analysis we calculated execution time and memory usage of searching and insertion algorithms. We evaluated these experiments for from 10 numbers to 100 numbers of data and represented on graphic.

Time probes are placed before and after function call of skip_insert() and skip_search_() functions. C++ Chrono library is used deals with time. Memory usage is calculated from number of nodes.

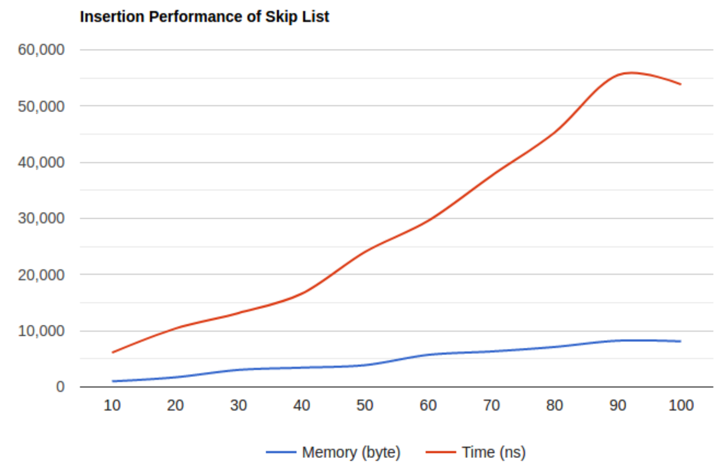

Figure 3: Insertion Performance of Skip List

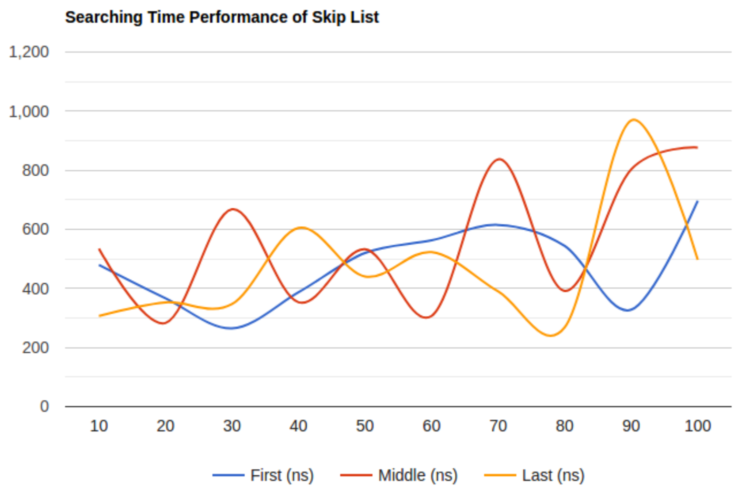

Figure 4: Searching Performance of Skip List

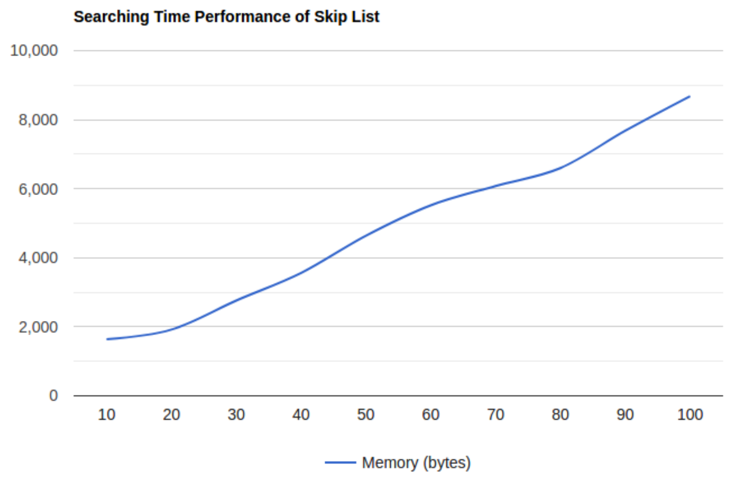

Figure 5: Searching Memory Usage of Skip List

4.1 Insertion Performance

Insertion experiment evaluated for 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 elements. All numbers randomly generated between 0 and 1000 by using rand(). Memory and execution time graphic can be shown in Figure 5.

4.2 Searching Performance

Searching experiment evaluated for 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 elements. All numbers randomly generated between 0 and 1000 by using rand(). After creation of skip list, first, middle and last element of this skip list are searched in the list and total execution times are calculated. Results can be shown in Figure 5. Also another graphic Figure 5 shows memory usage during searching experiment.

5 Conclusion

In conclusion, we create a model for skip list data structure for stores ordered integer numbers in memory. Then we implemented insertion, deletion and searching algorithms of this data structure with C++ and finally we evaluated performance of insertion and searching algorithms. Experimental result shows that memory usage is propotional with number of integers. Insertion algorithm graph behaves logaritmic.

References

[1] Goodrich, T. M., Algorithms and Data Structures in C++, 2nd ed.