Promblem_A Deterministic Skip List

Huige Cheng

August 20, 2009

Abstract

This document is written for the implementation of 1-2-3 skip list based on the paper "Deterministic Skip Lists".

1 Program Flow

```
create an empty skip_list_tree
    open input_file
    open output_file
while(can read input_file )
    read one line from input_file, parse it to get command
    switch(command)
        case: INS value
            skip_list_tree.Insert(value)
            if (can not allocate memory during Insert)
                goto end
        case: FIND value
            skip_list_tree.Search(value)
        case: REM value
            skip_list_tree.Remove(value)
write result to output_file
end:
clear skip_list_tree
close input_file
close output_file
```

2 Code Explanation

In the file named skip_list.h, I declared an abstruct class SkipList which serves as an interface for differnt types of skiplists, here I only implemented the 1-2-3 skip list, other possible implementations can be added later.

```
template <typename ElemType>
```

```
class SkipList {
    public:
        virtual void Clear() = 0;
        virtual bool Search(const ElemType &value) const = 0;
        virtual bool Insert(const ElemType &value) = 0;
        virtual bool Remove(const ElemType &value) = 0;
        virtual void Print() const = 0;
};
```

The following two classes are used for implementing the deterministic skip list (actually the 1-2-3 deterministic skip list).All the two classes SkipNode and DeterminSkipList are written in determine_skip_list.h.

2.1 SkipNode

```
template <typename ElemType>
class SkipNode {
    public:
        template <typename T>  friend class DeterminSkipList;
        SkipNode();
        SkipNode(const ElemType &key_input,
                 SkipNode *right_input = NULL,
                 SkipNode *down_input = NULL);
    private:
        DISALLOW_COPY_AND_ASSIGN(SkipNode);
        ElemType key;
        SkipNode *right;
        SkipNode *down;
};
```

SkipNode defines the node type for the skip list.
ElemType **key**:
The data member **key** is used for user to search.
SkipNode $*$**right**:
The data member **right** is a pointer pointing to the right SkipNode.
SkipNode $*$**down**:
The data member **down** is a pointer pointingg to the down side SkipNode.
Below are some explanations of pointer **right** and **down**. Say for one node which p points to is on level h, there is another skip node which q points to:
If $q = p- > right$ node $*q$ is on the same level h and is the successor of $*p$.
If $q = p- > down$ Let $p = p\_left- > right$, $q = q\_left- > right$. Node $*q$ is on the lower level h-1 and is the successor of $*q\_left$, and $q\_left- > key$ equals to $p\_left- > key$.

Node $*q$ is on the lower level h-1 and is the successor of $*q\_left$, and $q\_left->key$ equals to $p\_left->key$.(Please refer to the Figure 2 below) So if you go down you get to the right position of the level below.

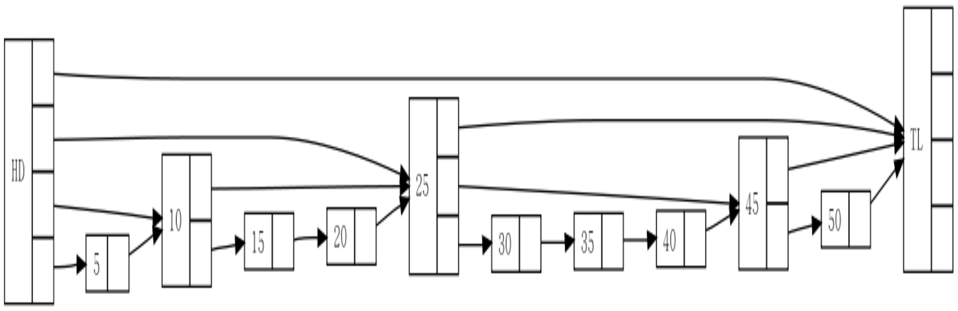


Figure 1: 1-2-3 Skip List (Abstract)

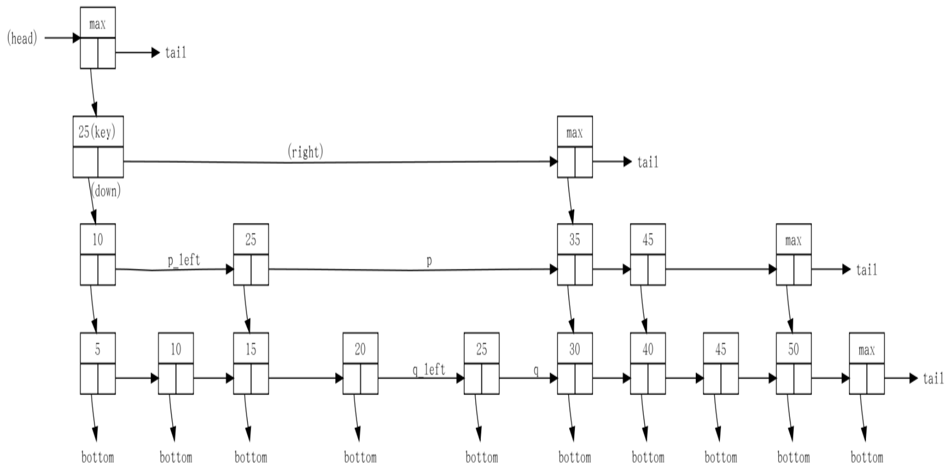


Figure 2: 1-2-3 Skip List (Implemetation)

Actually $*p\_left$ and $*q\_left$ represent the same node(in abstract sense the node with key 25) in different levels($*p\_left$ represent the node with key 25 of level 2, and $*q\_left$ represent level 1). If a node in abstract sense has max height h,then in our skip list we will have h skip nodes representing it, each skip node corresponding to one unique level.

The SkipNode class is invisible to the user, so I let all the data member (key, right, down) be private and for code simplicity let DeterminSkipList be its friend class so DeterminSkipList can visit its private data. Another possible implementation could be that let SkipNode be a nested class inside DeterminSkipList, but the complier seems not supporting this feature well for templated class so currently I still let the SkipNode class be an independent class.

2.2 DeterminSkipList

```
template <typename ElemType>
class DeterminSkipList :public SkipList<ElemType>{
    public:
        DeterminSkipList(const ElemType &max,
                         const ElemType &max_1);
        ~DeterminSkipList();
        void Clear();
        bool Search(const ElemType &value) const;
        bool Insert(const ElemType &value);
        bool Remove(const ElemType &value);
        void Print() const;
        bool IsValid() const;

    private:
        DISALLOW_COPY_AND_ASSIGN(DeterminSkipList);
        void Init();
        void ClearHelp(SkipNode<ElemType> *current);
        void FindAndModifyRemoveHelp(const ElemType &value,
                  const ElemType &other_value);
        void LowerHeadRemoveHelp();

    private:
        SkipNode<ElemType> *head;
        SkipNode<ElemType> *bottom;
        SkipNode<ElemType> *tail;
        const ElemType Max;
        const ElemType Max_1;
};
```

Currently, to use DeterminSkipList, the client must provide one value for Max, another value for Max_1 and make sure Max_1 ¿ Max ¿ any input key value. Since for code simplicity as the paper stated we need those two values Max and Max_1 as lookout.

For example if the key type is integer. The client could use it like below:

```
const int Max_1 = std::numeric_limits<int>::max();
const int Max = Max_1 - 1;
DeterminSkipList<int> skip_list(Max, Max_1);
```

If the user want to use std::string as key type and for any input string each character's ascii value < 128. Then client could use like:

```
char a[2];
a[0] = char(254); a[1] = '\0';
```

```
const string Max(a);
a[0] = char(255);
const string Max_1(a);
DeterminSkipList<string> skip_list(Max, Max_1);
```

For the user, he(she) can use the functions like Insert,Search,Remove and Clear. In order to help debug I have also added fuction Print and IsValid. Print will simply print the current skip list tree structure, not have good output effect but enough for the user to see the internal list structure eg.Max value is 100, Max_1 value is set to 101. So the head with its key 100 right pointer pointing to tail and down pointer pointing to the element whose key is 25.

```
100(25) 101[tail]
25(10) 100(45) 101[tail]
10(5) 25(15) 45(35) 100(50) 101[tail]
5(10) 10(10) 15(10) 20(10) 25(10) ... 50(10) 100(10) 101[tail]
```

IsValid will tell if the current skip list tree structure is correct, eg. if it finds one gap whose size > 4 it will return false.

3 Algorithm Summarize

3.1 Search

The key concept of searching is :

```
Step1: Start from head compare the current skip node key
       with the given value to search.
Step2: If the current skip node key > given value,then
       go to right else go down.
Repeat step 2 until come to the level 1(current skip node
p,p->down == bottom).
Compare current node key with the given value,
if equal than
    return true(find).
else
    return false(could not find).
```

3.2 Insert

The insert process is similar to search it uses the same flow except in order to keep the list as 1-2-3 skip list, we have to make sure no more than 3 elements of level h-1 between any elements of level h. To solve this problem, when going down from one level if we find the gap is 3 we will first raise the middle element of the 3 h-1 elements before we going the the lower level.

By doing this we can guarantee we always keep the correct 1-2-3 skip list structure. You can find detailed algorithm and one example of insertion figure from the paper,

Here one more thing to mention is there is a minor but fatal error in the code the paper presents. Say the current head is at level h. We may need to raise the head to level h + 1 if we need to raise one element of the level h - 1. If that happened ,even if we find the value v when we at the level 1, we do not need to insert but we still need to raise the head to keep our structure consistent. But the code the paper given shows that it immediately returned false after knowing the value has already exists at level 1. That will destroy the list structure and make subsequent operations to the tree resulting in wrong result. The fix is easy. See below:

```
int Insert(v)
{
    ....
    bool can_insert = true;
    while(x!= bottom) {
        ....
        if ((x->d == bottom) && (v == x->key)) {
            can_insert = false;
            break;
        }
        ....
    }
    if (head->r != tail) {
        ....
    }
    return can_insert;
}
```

Another thing to mention is I used $x- > key > x- > down- > right- > right- > key$ as the expression to judge if the gapsize is 3 instead of $x- > key == x- > down- > right- > right- > right- > key$, the effect is the same, but should run faster some how.

3.3 Remove

The remove process also uses the same flow as Search , now the problem occurs when we are facing a 1 element gap. The idea here is if we find we will go down when facing a gap whose size is 1. We will have to at first enlarge this gap. Neither by borrowing elements from neighboring gap or just merge with neighboring gap. (Actually we first try to merge with neighboring gap and if we find after mergence the gap size is 3 or less that's fine, otherwise we will have to raise one element to make our gap size equal to 2).

I used exactly the same strategy as the paper stated , on each level two pointers are needed this time, SkipNode $*previous$ and SkipNode $*current$. One for the current node and the other points to the one on the left side of current node if we have advanced toward right on this level.

On each level we advance toward right until we must stop and should go down. We see if we face a gap with its gap size == 1 and if so judging whether this is the first gap on this level. (Here first means when we come to this level we have not advanced toward right before we should go down.) If it is not the first gap we will merge with the gap on our right side, by lowering separator node, if the gap size after merge $>$ 3 we will have to raise the first element of the right gap. If it is the first gap then we will merge with the gap on our left side, by lowering separator node. If the gap size after merge $>$ 3 we will have to raise the last element of the right gap. Notice if after mergence the gap size is 1 or 2 or 3, we will have to delete one node, otherwise since we have to raise one element also we just need to swap value. When coming down to the level 1, we can safely delete the node if it is a height 1 node, otherwise we can swap it's key with its neighbor (left neighbor actually, suppose it has $key = x$) and delete it's neighbor.To implement this on our skip list data structure, we have go from head to search all the node with the key we want to delete and set its key to x. At last we will check if we need to lower the head pointer so to keep structure consistent. Below is all the related code for Remove operation of 1-2-3 skip list.

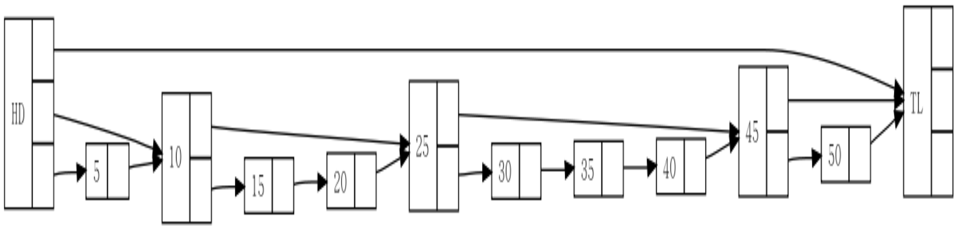
```
template <typename ElemType>
void DeterminSkipList<ElemType>::FindAndModifyRemoveHelp(
                                  const ElemType &value,
                                  const ElemType &other_value)
{
    SkipNode<ElemType> *current;
    current = head;
    while (current != bottom) {
        while (value > current->key)
            current = current->right;
        if (value == current->key)
            current->key = other_value;
        current = current->down;
    }

}
template <typename ElemType>
void DeterminSkipList<ElemType>::LowerHeadRemoveHelp()
{
    SkipNode<ElemType> *temp;
    if (head->down->right == tail) {
        temp = head->down;
        head->down = temp->down;
        delete temp;
    }
}
template <typename ElemType>
bool DeterminSkipList<ElemType>::Remove(const ElemType &value)
{
    if (head->down == bottom)
      return false;
    SkipNode<ElemType> *current, *previous, *next, *temp;
    current = head->down;
    int visit_num = 0;
    bool can_remove = true;
    for(;;) {
      previous = NULL;
      while (value > current->key) {
          previous = current;
          current = current->right;
      }
      if (value == current->key)
          visit_num++;
      if (current->down == bottom) {
          if (visit_num == 0) {
              LowerHeadRemoveHelp();
              can_remove = false;
              return false;
          }
          else {
              if(visit_num == 1) {
                  temp = current->right;
                  current->key = current->right->key;
                  current->right = temp->right;
                  delete temp;
                  LowerHeadRemoveHelp();
              } else {
                  LowerHeadRemoveHelp();
                  FindAndModifyRemoveHelp(current->key,
                                          previous->key);
                  previous->right = current->right;
                  delete current;
              }
              return true;
```

```
                }
            }
            next = current->down;
            if (current->key == current->down->right->key) {
                if (previous == NULL) {
                    if (current->right->key >
                        current->right->down->right->key) {
                        current->key = current->right->down->key;
                        current->right->down = current->right
                                                ->down->right;
                    } else {
                        temp = current->right;
                        current->key = temp->key;
                        current->right = temp->right;
                        delete temp;
                    }
                } else {
                    if (previous->key > previous->down
                                                 ->right->key) {
                        temp = previous->down->right;
                        if (temp->right->key != previous->key)
                            temp = temp->right;
                        previous->key = temp->key;
                        current->down = temp->right;
                    } else {
                        previous->key = current->key;
                        previous->right = current->right;
                        delete current;
                    }
                }
            }
            current = next;
        }
    }
```

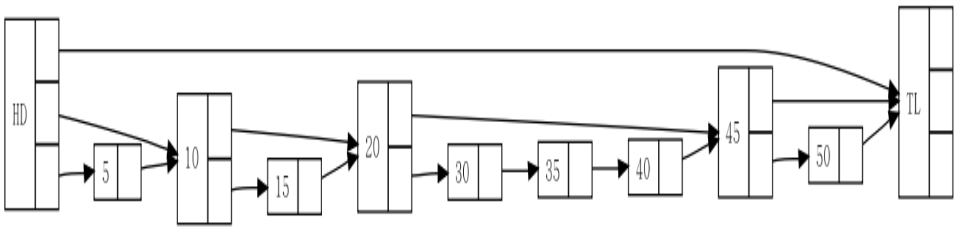
In the paper, the author has given one example of deleting element, now I will illustrate another one which delete one element whose *height* > 1, suppose we are going to delte the element whose key is 25 of the 1-2-3 skip list as figure 1 shows.

First, at level 3, we will go down in the first gap, ant it's gap size is only 1, so consider the first gap and the gap on the right.Since after merge the gap size will be 3. So we just need to lower down the element whose key is 25, Pay attention to the fact that we also need to lower the head. The figure below shows the result,lowering the elment with key 25 as well as the head.



Next on level 2, we are at the element a whose key is 10 and then go right to the element b whose key is 25. This time the gap size between a and b is 2, so we can go down safely.

Then we come to the final level and starting from the element whose key is 15 until get to the element whose key is 25, then we will have to delete this element. But it's height is 2, so we will swap key with the left element whose key is 20 before deleting the neigbour.The figure below illustrates the final result of deleting the element with key 25.



4 Test and Performance

The correctness of the algorithm can be tested by comparing the result of skip list with std::set. To test the performance I used random insert (3332964 times), search(3332964 times) and remove(3330534 times) of random data(integer, random generated ranging from 0 to 21747483645), experiments showed 1-2-3 skip list performed as well as std::set, the skip list used 66.17s while the latter used 67.72s. (I am running on my virtual machine ,so quite slow and the result may change a lot for different running, but it did show skip list operations are of the same time complexity as std::set, $O(log(n))$).