AVL & SPLAY TREE

GROUP13

BY SHUTING GUO, RUIJIA ZHANG, ZIYUE TANG

ADS PROJECT 1, MARCH 2019

PART I

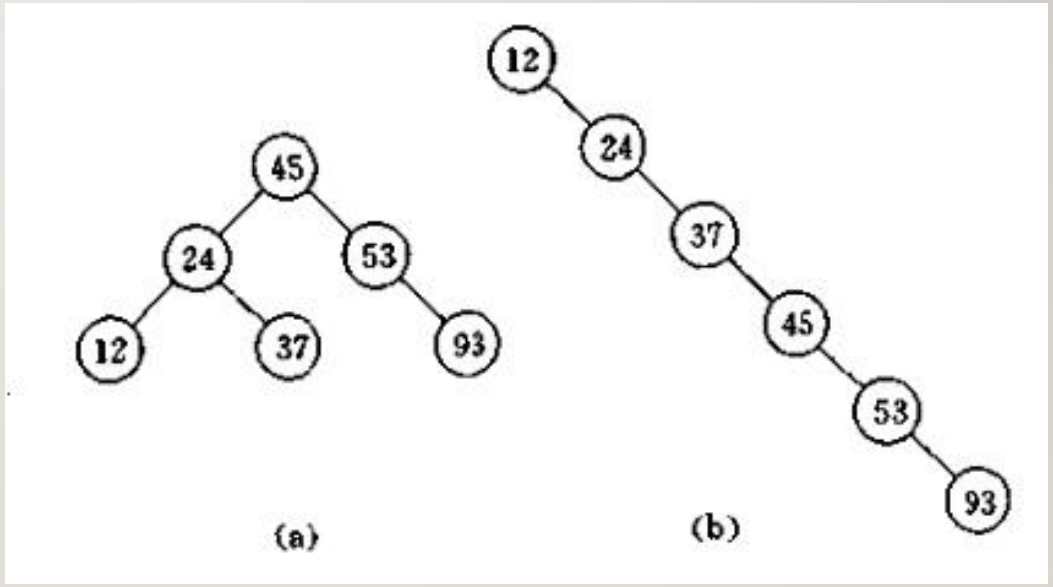
INTRODUCTION OF ALGORITHM

MAIN TASK

- This lab project requires a Splay and an AVL tree implement programmed by C/C++. And then, test the performance on several cases of data inputs and analyze the complexity of time and space, and compare with basic non-balanced BST.

MAIN IDEA - BALANCED BINARY TREE

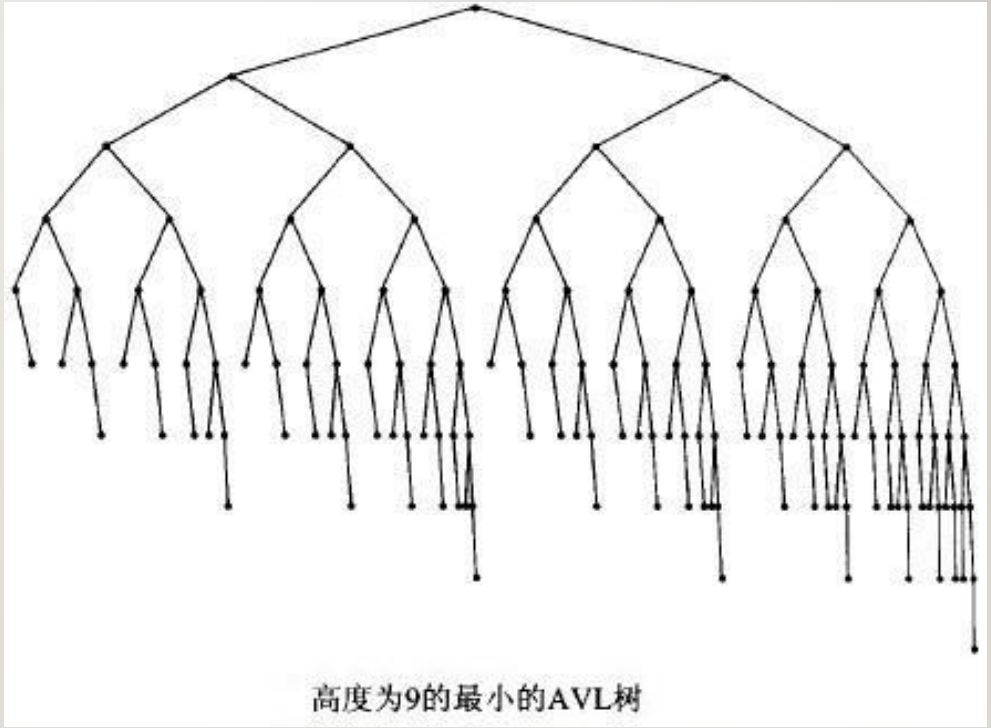
- Sometimes, we use a BST to keep data, so that we can find an element by comparing with elements in a binary tree.
- However, usually the shape of a BST is not as good as we expected. Figure (b) is an example, for finding an element in such a tree, is no better than in an array.
- And when this happens, we want to optimize the shape of a tree and balanced BST structures were designed.



MAIN IDEA - BALANCED BINARY TREE – AVL

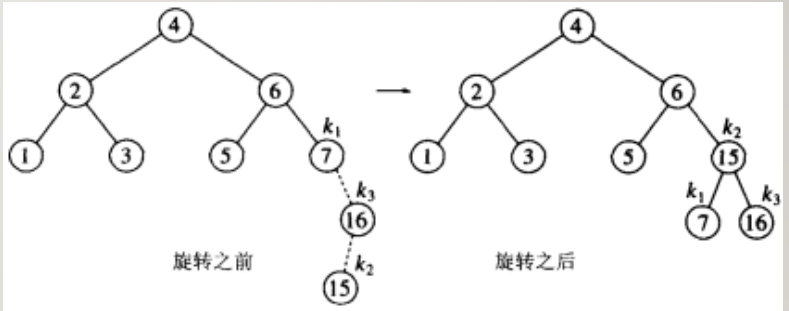
- There are many ways to make a BST balanced, which means, all operations in a tree could be done in O(logN) time.

- The most direct way to keep a BST in balance is – keep balanced all the time. And this is what structure called AVL tree does.



MAIN IDEA - BALANCED BINARY TREE – AVL

- AVL keeps the following property:
- *For each node in the tree, difference of the height of two children is no more than 1.*
- To keep this property, once an element is inserted, or deleted, the height will change, and AVL fix the difference of height by rotation. Because rotate the higher subtree to the lower side will reduce the difference of height.



MAIN IDEA - BALANCED BINARY TREE – AVL

- It can be proved from the property we talked about above, that the height of the tree is O(logN). And we know that if we insert a node into the tree, the subtrees' height will grow by O(1), also, if we delete an element, the height will decrease by O(1), and each rotation, reduce the difference of height by O(1).
- Therefore, only O(n) rotation will be performed on the tree, during insert and delete.

MAIN IDEA - BALANCED BINARY TREE – AVL

01

The advantage of AVL is clear now. Because once an AVL is built, no more operation is necessary while making query. So if there is a large amount of query, benefit form the strict height limit, AVL is very fast.

02

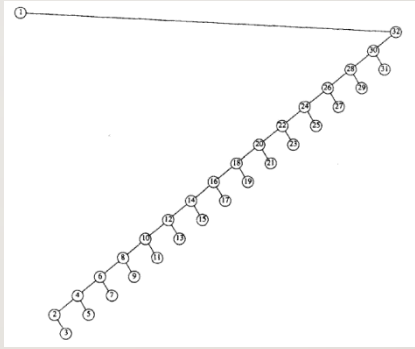
The disadvantage of AVL is that, AVL is not flexible enough to deal with complexed operations such as rotate, split, merge, interval operations and so on. Because the shape of AVL that contains specified data is unique.

MAIN IDEA - BALANCED BINARY TREE – SPLAY

- We've talked about AVL before. The strategy of AVL to keep balance is, keeping balance all the time.
- On the opposite, splay use a entirely different strategy.
    - Getting better on each operation.
- Splay is not a strictly balanced BST, what splay does is shorten a long chain to half on each operation by a core technology called 'splay'.

MAIN IDEA - BALANCED BINARY TREE – SPLAY

- However, Splay rotates the node with every operation. The reason why Splay can do this while keeping balance is, in Splay's rotate operation, the order of rotation is decided on double nodes.
- This double-rotation guarantees **the height will not be worse** during the large amount of rotation.
- So, when the amount of operation become huge, each operation is implied with rotate operation, means that the shape of Splay is getting more and more balanced. Splay can be rotated arbitrarily, so splay is more flexible. But it needs more time to rotate, so splay can be a little bit slower than other balanced trees.



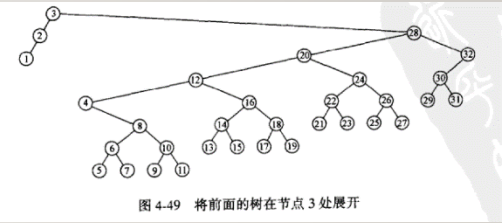


图 4-49　将前面的树在节点 3 处展开

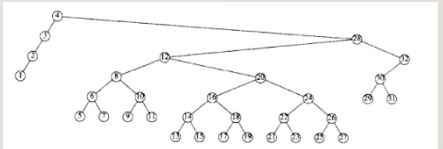


图 4-50　将前面的树在节点 4 处展开

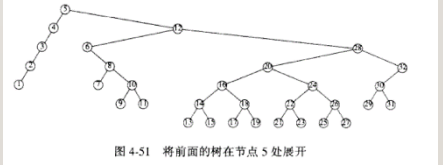


图 4-51　将前面的树在节点 5 处展开

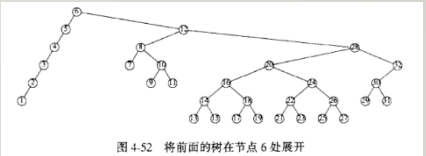


图 4-52　将前面的树在节点 6 处展开

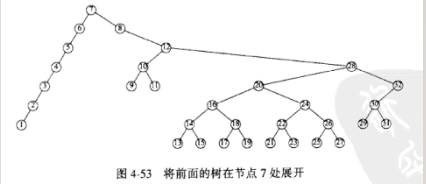


图 4-53　将前面的树在节点 7 处展开

MAIN IDEA - BALANCED BINARY TREE – SPLAY

The advantage of splay is, the shape of the tree is free, you rotate splay as you want. Because the double-rotation tech. guarantees the amortization complexity is O(logN). This suggests that splay is very flexible.

The disadvantage is, the height of splay is not strict. And each time we access data needs a set of rotation, so this makes splay slower than other balanced BST.

PART 2

STRUCTURE IMPLEMENT

STRUCTURE IMPLEMENT– AVL OPERATIONS

- **Insert**
- Insert a node recursively, until a NULL node is met, allocate a new node and replace it. While tracing back, update the height of each node and check if a subtree is still balanced. If not, fix it by rotation.
- **Delete**
- First, find the node to delete. If it's a leaf node, delete it directly. Else, replace it by its precursor or subsequence in left or right subtree (always choose the higher subtree), and delete its precursor or subsequence recursively. Update the node to keep balance by the same way as insert.
- **Update node**
- Let $height = \max\{LeftHight\,, RightHeight\} + 1$.
- If $|LeftHeight - RightHeight| > 1$, check the direction of edges that link nodes, and always rotate the higher subtree to lower side. (Use LL RR LR RL.)

STRUCTURE IMPLEMENT– AVL CORE CODE

```
// Fix Height after insert or delete
void update_node(node *& t)
{
    // Update height
    t->h=max(geth(t->l),geth(t->r))+1;
    // Unbalanced
    if(AVL_abs(geth(t->l) - geth(t->r)) >  1)
    {
        if(geth(t->l) > geth(t->r))
            if(geth(t->l->l) > geth(t->l->r)) RotateR(t);
            else RotateL(t->l), RotateR(t);
        else
            if(geth(t->r->l) < geth(t->r->r)) RotateL(t);
            else RotateR(t->r), RotateL(t);
    }
}

void Insert(node *& t,const _tp & x)
{
    if(!t) { t=ALLOC(x); t->h=1; return ; }
    if(x==t->val) return ;
    if(x<t->val) Insert(t->l,x);
    else Insert(t->r,x);
    update_node(t);
    return ;
}

node * Find(node * t,const _tp & x)
{
    while(t && x!=t->val)
        if(x < t->val) t=t->l;
        else if(x > t->val) t=t->r;
    return t;
}
```

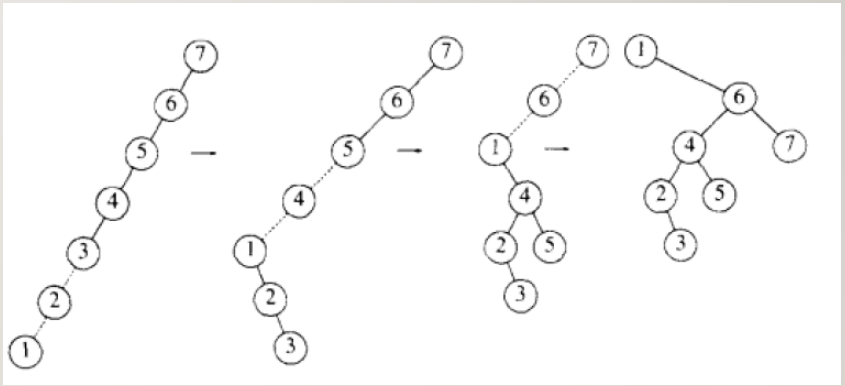
```
void Delete(node *& t,const _tp & x)
{
    if(!t) { return ; }
    // Found x, delete it
    if(x==t->val)
        if(!t->l && !t->r) RECYCLE(t),t=NULL;
        else if(t->l && !t->r) RECYCLE(t),t=t->l;
        else if(t->r && !t->l) RECYCLE(t),t=t->r;
        // replace t with min node 'P' in left subtree
        //   or max node 'P' in right subtree
        // Delete 'P' recursively
        else
            if(geth(t->l) > geth(t->r))
                t->val=Find_Max(t->l)->val,
                Delete(t->l,t->val);
            else
                t->val=Find_Min(t->r)->val,
                Delete(t->r,t->val);
    else if(x<t->val) Delete(t->l,x);
    else              Delete(t->r,x);

    // keep balance
    if(t) update_node(t);
    return ;
}
```

STRUCTURE IMPLEMENT– SPLAY

- The rotation in splay considers two nodes rather than one. The main purpose to do this is shorten the path to root.
- There are two implements of 'splay'.
- The first way is find a node and rotate it up to root.
- The second way is to split the tree from root and then merge them.

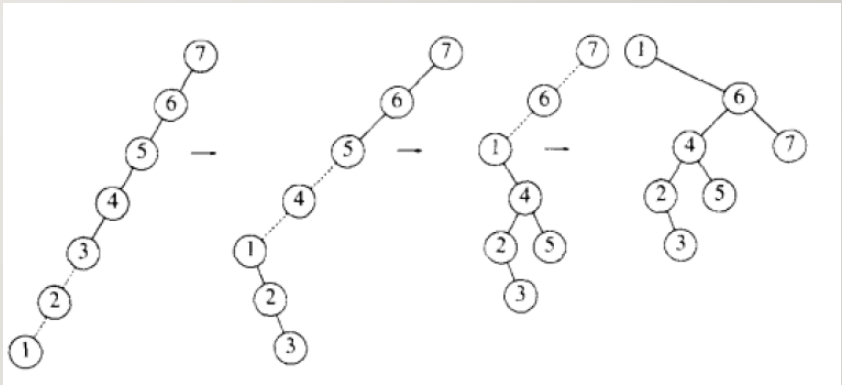
STRUCTURE IMPLEMENT– SPLAY - METHOD 1, ROTATE TO ROOT (FROM LEAF UP TO ROOT)

- **Insert**
- 1. For normal cases, splay precursor to root, splay subsequence to root->r, then the node should be linked to root->r->l.
- 2. If the new node X is min(max) element in the tree, splay min element to root and link root to the right(left) side of X, then replace the root with X. That is, after insertion, the old root should be right(left) son or new root X.
- 3. If the tree is empty, use X as root.



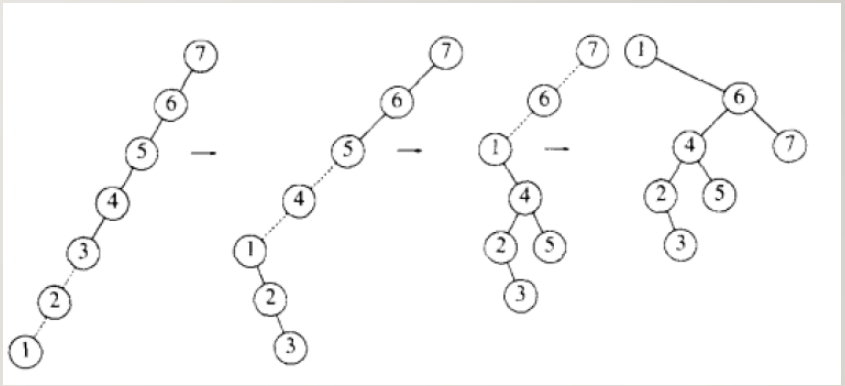
STRUCTURE IMPLEMENT– SPLAY - METHOD 1, ROTATE TO ROOT (FROM LEAF UP TO ROOT)

- **Delete**
- 1. For normal cases, splay precursor to root, splay subsequence to root->r, then the node should be deleted is root->r->l.
- 2. If the element is min(max) element in the tree, splay it to root, use root->r(root->l) as root and delete the old root.
- 3. If the element the only element int the tree, delete it directly, and the tree becomes empty.



STRUCTURE IMPLEMENT– SPLAY - METHOD 1, ROTATE TO ROOT (FROM LEAF UP TO ROOT)

- **Rotate**
- Rotate a node to shallower position.
- **Splay**
- Use Zig-Zig, Zig-Zag, Zag-Zig, Zag-Zag, rotate a node up, until its father node is target. (If target is NULL, the node will be rotated to root.)
- **Find**
- After node is found or not, splay the last-accessed node to root. (This should be well noticed, because if you forget to splay especially when the node is not found, the time complexity can increase to O(N).)



STRUCTURE IMPLEMENT– SPLAY - METHOD 1, ROTATE TO ROOT (FROM LEAF UP TO ROOT)

```
void Rotate(node * t) // rotate up, height(t) reduces
{
    node * gfa=t->fa->fa;
    // t is right node of t->fa, rotateL
    t->getlr()?t->link(0,t->fa->link(1,t->s[0])):
    // t is left node of t->fa, rotateR
        t->link(1,t->fa->link(0,t->s[1]));
    // fix pointers
    if(gfa) gfa->link(gfa->s[1]==t->fa,t);
    else t->fa=0,Root=t;
}

// rotate t up until t->fa==NULL
// splay(t) means rotate t to root
node * splay(node * t,node * tar=NULL)
{
    // double rotate
    while(t->fa!=tar && t->fa->fa!=tar)
        t->getlr()==t->fa->getlr()?
        // same direction
        (Rotate(t->fa),Rotate(t)):
        // differnt direction
        (Rotate(t),    Rotate(t));
    // no grandfather node, sigle rotate
    if(t->fa!=tar) Rotate(t);
    return t;
}
```

```
void Insert(node *& t,const _tp & x)
{
    if(Find(x)) return ;
    node * templ, * tempr, * tempx;
    templ=Find_less(Root,x); // precursor position
    tempr=Find_greater(Root,x); // subsequence position
    tempx=ALLOC(x); // newnode
    // insert to root->r-l
    if(templ && tempr)       splay(tempr,splay(templ))->link(0,tempx);
    // x is max
    else if(templ && !tempr) Root=tempx->link(0,splay(templ));
    // x is min
    else if(!templ && tempr) Root=tempx->link(1,splay(tempr));
    // empty tree
    else                     Root=tempx;
}

node * Find(node * t,const _tp & x)
{
    node * last=0;
    while(t && x!=t->val)
    {
        last=t;
        if(x < t->val) t=t->s[0];
        else if(x > t->val) t=t->s[1];
    }
    if(t) splay(t);
    // splay a node no matter x is found or not
    //   if not do so, complexity of Find(x) BOOMBOOM!
    else if(last) splay(last);
    return t;
}
```

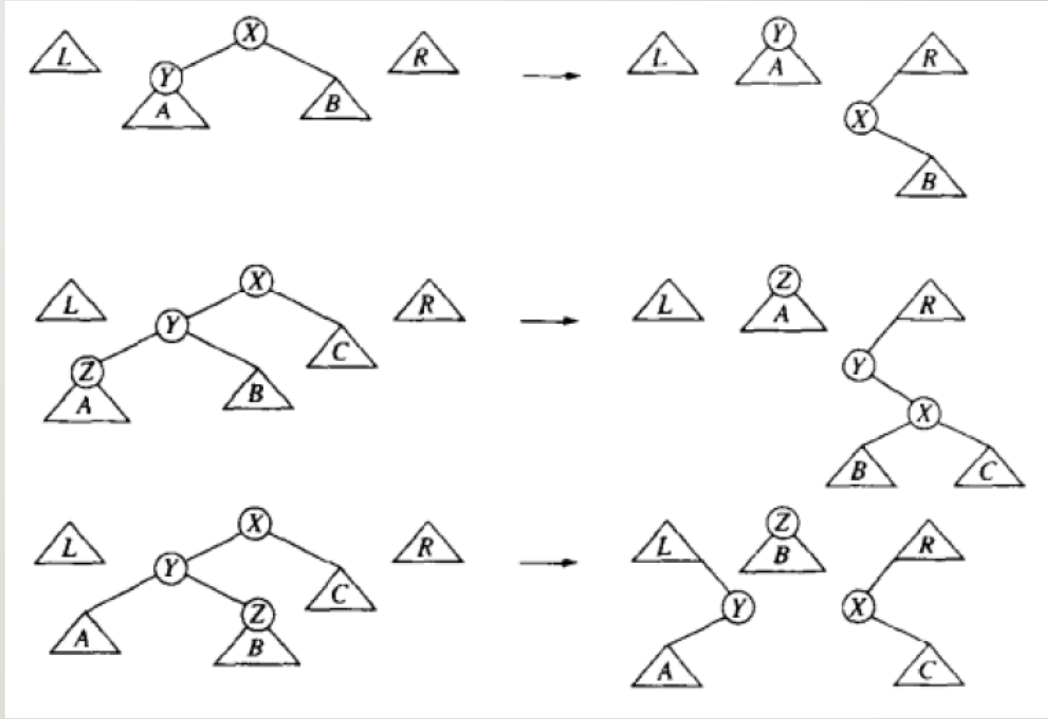
```
void Delete(node *& t,const _tp & x)
{
    node * tempx, * templ, * tempr;
    tempx=Find(Root,x); // find x
    if(!tempx) return ; // if no such node, return
    templ=Find_less(Root,x); // precursor position
    tempr=Find_greater(Root,x); // subsequence position
    // delete root->r->l
    if(templ && tempr)       splay(tempr,splay(templ))->s[0]=0;
    // x is max
    else if(templ && !tempr) splay(templ)->s[1]=0;
    // x is min
    else if(!templ && tempr) splay(tempr)->s[0]=0;
    // empty tree
    else                     Root=0;
    RECYCLE(tempx); // recycle memory
}
```

```
node * Find_greater(node * t, _tp x)
{
    node * temp=0, * last=0;
    while(t)
    {
        last=t;
        if(x<t->val) temp=t,t=t->s[0];
        else t=t->s[1];
    }
    if(temp) splay(temp);
    else if(last) splay(last);
    return temp;
}

node * Find_less(node * t, _tp x)
{
    node * temp=0, * last=0;
    while(t)
    {
        last=t;
        if(x<=t->val) t=t->s[0];
        else temp=t,t=t->s[1];
    }
    if(temp) splay(temp);
    else if(last) splay(last);
    return temp;
}

void To_vector(node * t, std::vector<_tp> & vec)
    // In order to avoid recursion, this function is O(NlogN)
{
    if(!t) return ;
    _tp temp=Find_Min(Root)->val;
    for(int i=0;i<size;++i)
        vec.push_back(temp),temp=Find_greater(temp);
}
```

STRUCTURE IMPLEMENT– SPLAY
- METHOD 1, ROTATE TO ROOT (FROM LEAF UP TO ROOT)

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- As we see in the figure, suppose we are looking for node 'Z'. Each time we found a node, consider whether it's less or greater than pivot 'Z'. If less, link to tree 'L' where every node is less than 'Z'. And in tree 'R', every node is greater than 'Z'.



STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Zig:
- When ‘Y’ have no left child or ‘Y’ is the key we are looking for, let ‘Y’ be the root of main tree and link X to Right-Tree.

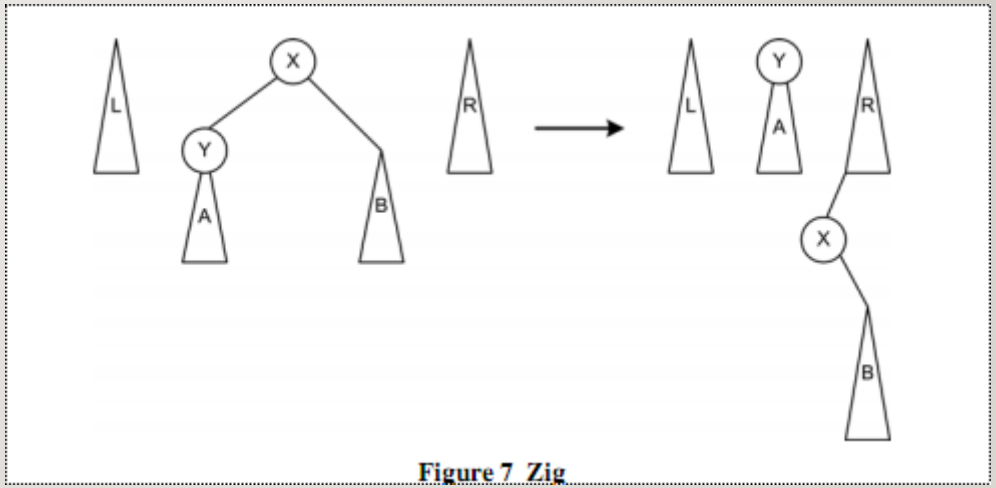


Figure 7 Zig

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Zig-Zig:
- This situation, we find that the Key we want to find is in Z and its SubTree. So we first single rotation Y and then right link it.

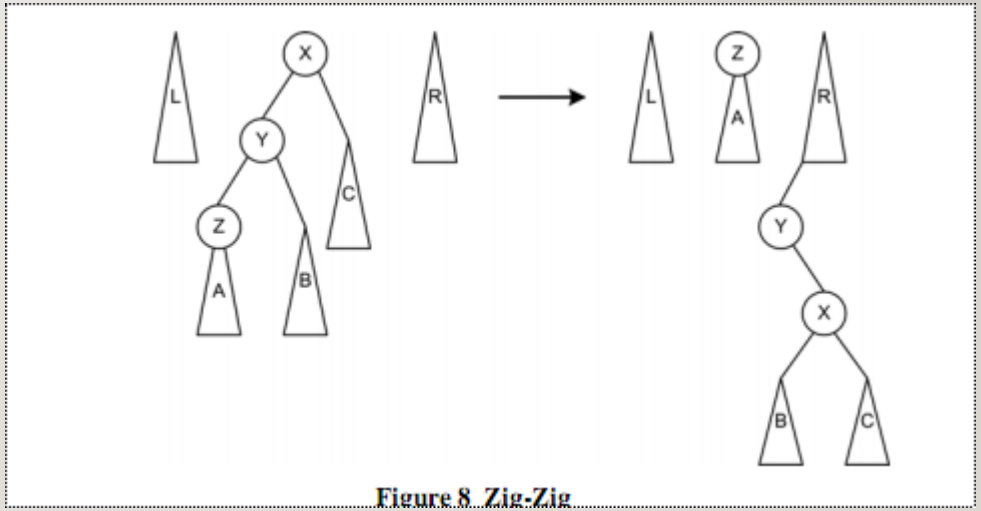
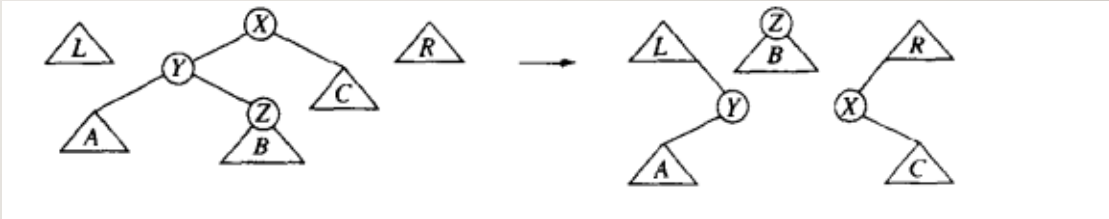


Figure 8. Zig-Zig

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Zig-Zag:
- If Key is smaller than X,and lager than Y ,We can right link X and Left link Y ,then make Z be the root of middle Tree. But this may be a little difficult when we implement it by code,so we try to simplify Zig-Zag situation by this.



STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Zig-Zag(Simplify):
- We only solve it by one floor (right link X, and let Y be root), so
- that we can solve Y later (and the result is not changed because we
- left link Y). And Zig-Zag situation can be implemented same as Zig

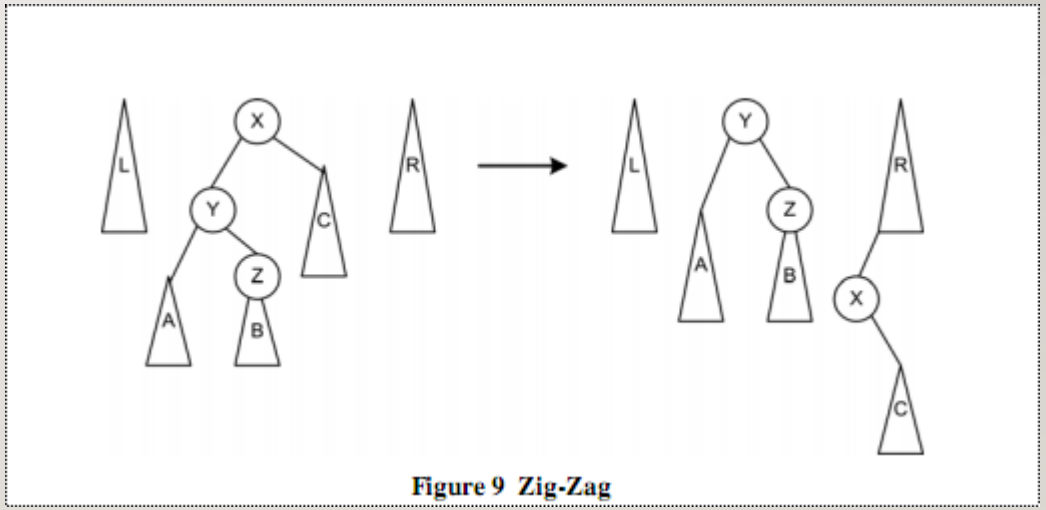


Figure 9 Zig-Zag

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Combine
- We we finally find X(X is the root of middle tree).We combine the left middle and right tree.We right link X's right sub tree and left link X's left sub tree.Then we let right tree be X's right sub tree and left tree be X's left sub tree.
- Then we build a new tree with the root X.
- (And if X is not in the tree,we let the key which is the most near X be the root.)

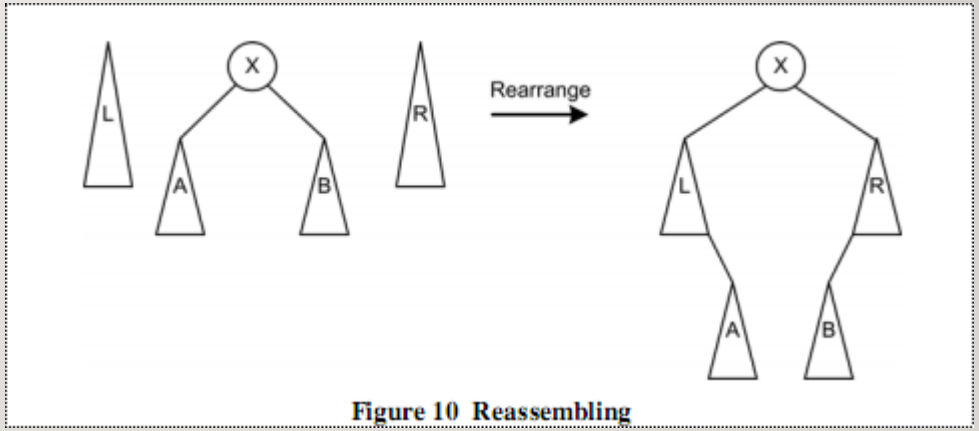
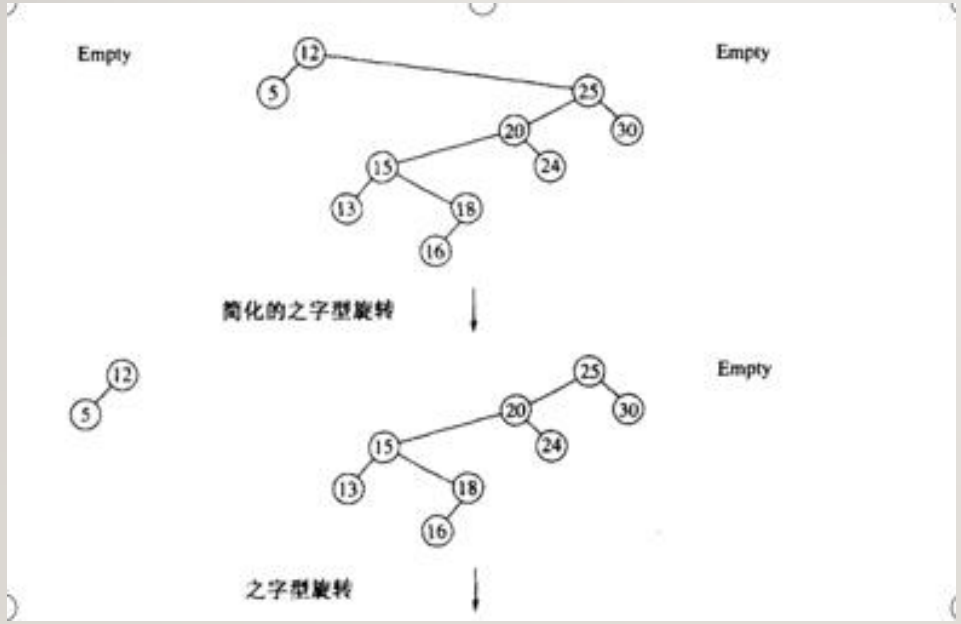
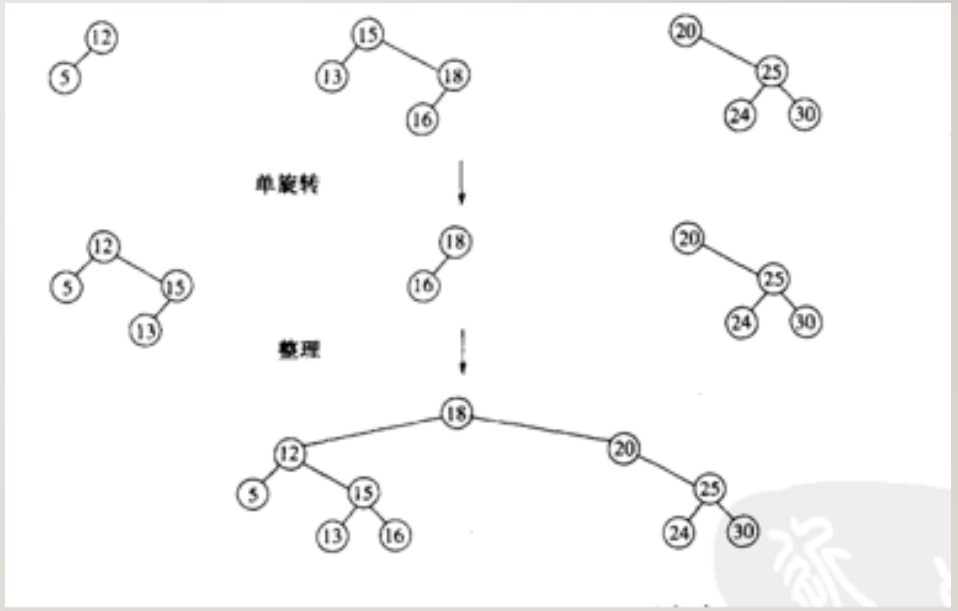


Figure 10 Reassembling

STRUCTURE IMPLEMENT– SPLAY
- METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- An Example
- Find 19





STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Insert
- If T is empty tree,let X be the root and return T.
- Otherwise,Splay the Tree by X.If X is not in the tree,then the root must be a number nearly X.Compare X with it,and link it depend on the situation.

```
16.     else
17.     {
18.         T = Splay(T, X);
19.         if(X < T->Key)
20.         {
21.             NewNode->Left = T->Left;
22.             NewNode->Right = T;
23.             T->Left = NullNode;
24.             T = NewNode;
25.         }
26.         else if(X > T->Key)
27.         {
28.             NewNode->Right = T->Right;
29.             NewNode->Left = T;
30.             T->Right = NullNode;
31.             T = NewNode;
32.         }
33.         else return T;
34.     }
35.
36.     NewNode = NULL;
37.     return T;
38. }
```

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

- Delete
- First,Splay the Tree by X. And if the root is X,we can know that X is in the Tree.If X doesn't have left child,let New T = T->Right,then we delete X.And if X have two child.The we splay X's left sub tree by X,which means that the root of new tree doesn't have right sub tree.So we link X's right sub tree to the new tree.

```
1.  ST Delete(ST T, int X)
2.  {
3.      ST NewTree;
4.      if(T != NullNode)
5.      {
6.          T = Splay(T, X);
7.          if(X == T->Key)
8.          {
9.              if(T->Left == NullNode)
10.                 NewTree = T->Right;
11.             else
12.             {
13.                 NewTree = T->Left;
14.                 NewTree = Splay(NewTree, X);
15.                 NewTree->Right = T->Right;
16.             }
17.             delete T;
18.             T = NewTree;
19.         }
20.     }
21.
22.     return T;
23. }
```

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

```
ST Splay(ST T,int X)
{
    static struct SplayTree Header;
    ST LeftTreeMax,  RightTreeMin;

    Header.Left = Header.Right = NullNode;
    LeftTreeMax = RightTreeMin = &Header;
    NullNode->Key = X;

    while(X != T->Key)
    {
        if(X < T->Key)
        {
            if(X < T->Left->Key)
                T = RR(T);
            if(T->Left == NullNode)
                break;

            RightTreeMin->Left = T;
            RightTreeMin = T;
            T = T->Left;
        }
        else
        {
            if(X > T->Right->Key)
                T = LL(T);
            if(T->Right == NullNode)
                break;

            LeftTreeMax->Right = T;
            LeftTreeMax = T;
            T = T->Right;
        }
    }

    LeftTreeMax->Right = T->Left;
    RightTreeMin->Left = T->Right;
    T->Left = Header.Right;
    T->Right = Header.Left;

    return T;
}
```

```
ST Insert(ST T,int X)
{
    static ST NewNode = NULL;

    if(NewNode == NULL)
    {
        NewNode = new SplayTree;
    }
    NewNode->Key = X;

    if(T == NullNode)
    {
        NewNode->Left = NewNode->Right = NullNode;
        T = NewNode;
    }
    else
    {
        T = Splay(T, X);
        if(X < T->Key)
        {
            NewNode->Left = T->Left;
            NewNode->Right = T;
            T->Left = NullNode;
            T = NewNode;
        }
        else if(X > T->Key)
        {
            NewNode->Right = T->Right;
            NewNode->Left = T;
            T->Right = NullNode;
            T = NewNode;
        }
        else return T;
    }

    NewNode = NULL;
    return T;
}
```

STRUCTURE IMPLEMENT– SPLAY - METHOD 2, SPLIT & MERGE (FROM ROOT DOWN TO LEAF)

```
ST Delete(ST T, int X)
{
    ST NewTree;
    if(T != NullNode)
    {
        T = Splay(T, X);
        if(X == T->Key)
        {
            if(T->Left == NullNode)
                NewTree = T->Right;
            else
            {
                NewTree = T->Left;
                NewTree = Splay(NewTree, X);
                NewTree->Right = T->Right;
            }
            delete T;
            T = NewTree;
        }
    }

    return T;
}
```

STRUCTURE IMPLEMENT– SPLAY - METHOD 1 & 2, COMPARISON AND ANALYSIS

- We tested the performance of two implements of SPLAY, as we expected, method 1 is a little bit slower than method 2.
- Consider why this happens?
- In method 1, each operation needs to find and then splay(rotate) a node to root. But in method 2, it doesn't needs to do so, for while finding, it splits the tree in the same time. Reduce of iteration speed method 2 up.
- However, though method 2 is faster, method 1 is more easy to write and develop extensions.

PART 3

TEST OF CORRECTNESS
& PERFORMANCE

CORRECTNESS TEST

- The correctness is checked by comparing with std::set<int>. Data is generated by random generator. To ensure the correctness of algorithm, we write 5 codes, each use STL std::set<int>, AVL, two Splays and normal unbalanced BST.
- Input *N* and *N* commands. Commands are input as following way.
- `i x , insert x, output the number of elements in the tree`
- `d x , delete x, output the number of elements in the tree`
- `f x , find x, output True if x is found, otherwise output False`
- `l x , find the element less than x, and output it`
- `g x , find the element greater than x, and output it.`
- After all operations, output all elements in the tree in order.
- ***For each random data, for codes run together, then check if four output are same.***

PERFORMANCE TEST

- We have tested these codes in five situations, and measure the time cost on each, result is shown as follows.

SITUATION 1

- The same as correctness testing.
- In this case, input is random, so that in normal BST, the shape of tree is tend to be balanced. Meanwhile, splay is slower because its frequent rotation.

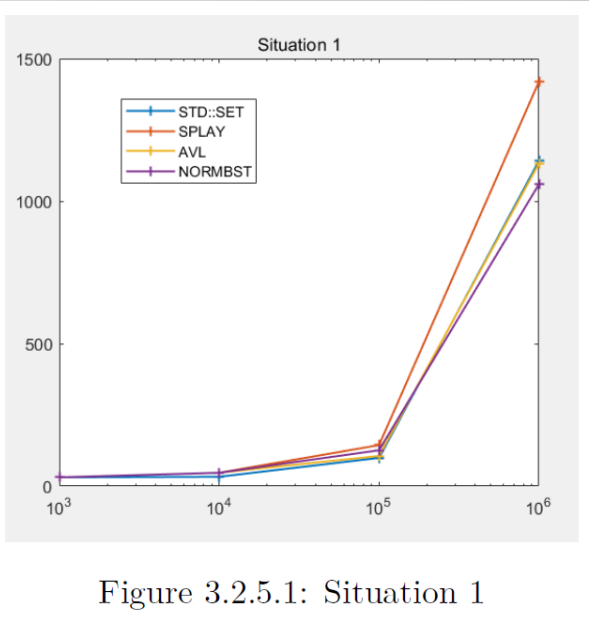


Figure 3.2.5.1: Situation 1

Algorithm	1000	1e4	1e5	1e6
std::set	31ms	33ms	99ms	1141ms
Splay	31ms	47ms	144ms	1419ms
AVL	31ms	47ms	106ms	1132ms
Normal BST	31ms	47ms	126ms	1059ms

Table 3.2.1.1: Situation 1

SITUATION 2

- Insert 1-N in order and delete in the same order.
- In this case, AVL needs very frequent rotation to keep balance. And normal BST is grown up to *O*(*N*2) and the time cost become unacceptable.

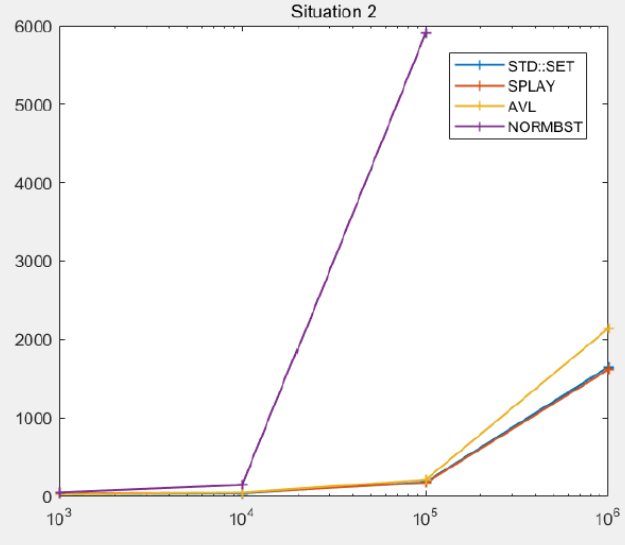


Figure 3.2.5.2: Situation 2

Algorithm	1000	1e4	1e5	1e6
std::set	28ms	38ms	178ms	1647ms
Splay	35ms	45ms	169ms	1612ms
AVL	28ms	46ms	209ms	2138ms
Normal BST	47ms	141ms	9503ms	-

Table 3.2.2.1: Situation 2

SITUATION 3

- Insert 1-N in order and delete in reversed order.
- The same as Situation 2. But for Splay, in this case, it doesn't needs to do any rotation, just link nodes one by one to the root, and delete one by one from root.

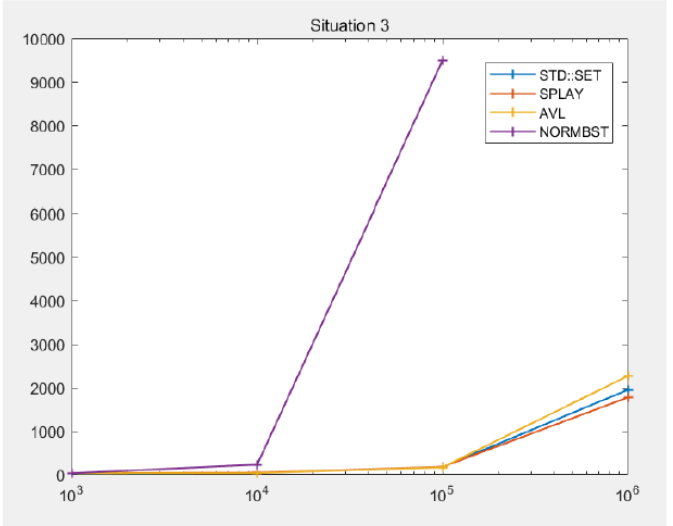


Figure 3.2.5.3: Situation 3

Algorithm	1000	1e4	1e5	1e6
std::set	28ms	40ms	177ms	1952ms
Splay	39ms	47ms	179ms	1776ms
AVL	35ms	38ms	196ms	2269ms
Normal BST	35ms	231ms	9503ms	-

Table 3.2.3.1: Situation 3

SITUATION 4

- Number in range [0,N), insert and delete in random order.
- The same as Situation 1.

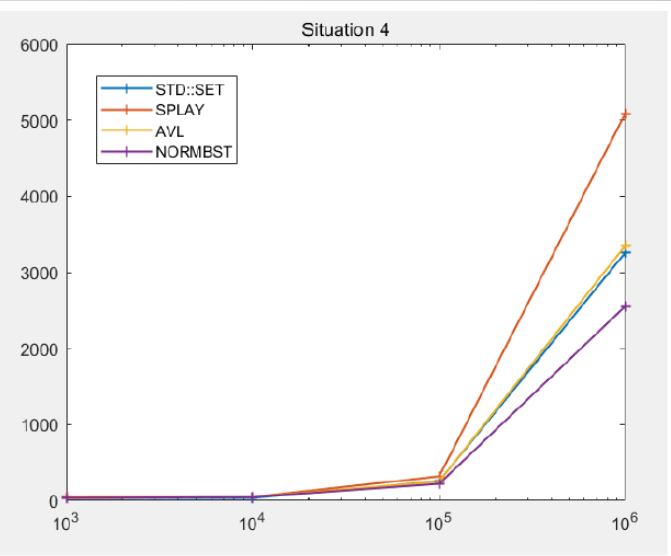


Figure 3.2.5.4: Situation 4

Algorithm	1000	1e4	1e5	1e6
std::set	32ms	34ms	258ms	3260ms
Splay	44ms	47ms	319ms	5089ms
AVL	31ms	46ms	253ms	3352ms
Normal BST	31ms	47ms	211ms	2556ms

Table 3.2.4.1: Situation 4

SITUATION 5

- Insert 1-N in order, then *find* 0 for N times, *find_less* N+1 for N times, *find_greater* 0 for N times.
- This case is to test whether complexity of query operations are correct.

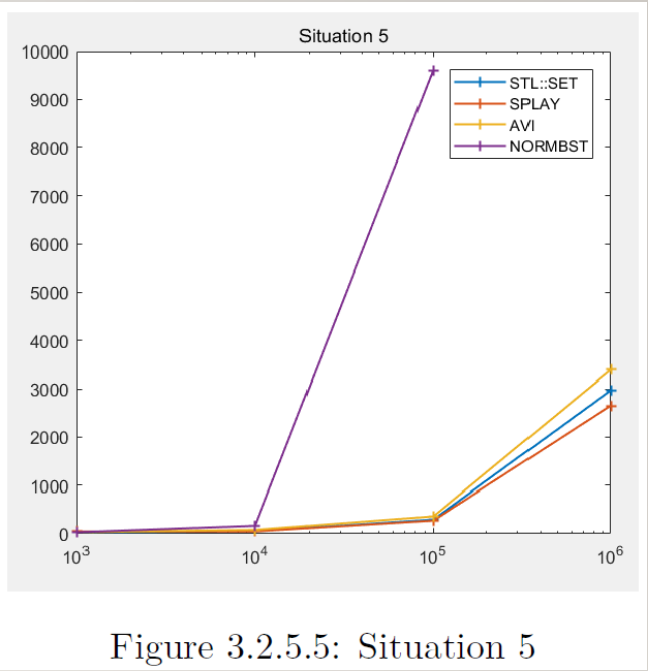


Figure 3.2.5.5: Situation 5

Algorithm	1000	1e4	1e5	1e6
std::set	32ms	48ms	291ms	2971ms
Splay	46ms	47ms	271ms	2656ms
AVL	32ms	79ms	357ms	3394ms
Normal BST	31ms	163ms	9611ms	-

Table 3.2.5.1: Situation 5

- Q & A

- *There are many data structures and algorithms to solve a problem.*
- *But none of is just a knowledge point in our text book.*
- *Data structures and algorithms are whatever you think.*
- *They are alive.*

- ***THANK YOU!***