浙江大學

Binary Search Trees

Advanced Data Structures and Algorithm Analysis
Research Project 1

Wang Zhongwei 王中伟
Lin Jiafeng 林家丰

March 9, 2019

Contents

Chapter 1

Introduction

1.1 Problem Description

We have just learned AVL tree and splay tree, they are the upgrade versions of unbalanced binary search tree. Because they are more efficient in theory. In this project, we want to test and verify their efficiency in practice through a series of test cases.
Then we will analyse their space and time complexity and give the mathematical proof of their efficiency.

1.2 Background of Data Structures

In computer science, a tree is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Chapter 2

Algorithm Specification

2.1 Binary Search Tree

2.1.1 Definition

A binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:

1. The left subtree of a node contains only nodes with elements less than the node's element.
2. The right subtree of a node contains only nodes with elements greater than the node's element.
3. The left and right subtree each must also be a binary search tree.
4. There must be no duplicate nodes.

Binary-search-tree property

Let x be a node in a binary tree. If y is a node in the left subtree of x, then

$$y.elements \leq x.elements$$

If y is a node in the right subtree of x, then

$$y.elements \geq x.elements.$$

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their elements rather than any part of their associated records.
The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.
Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

2.1.2 Operations

Searching

Insertion

To insert an element k in the BST, first you have to find the proper node y which meets one of following :

1. if $k <$element(y), then append element on the left subtree of y
2. if $k >$element(y), then append element on the right subtree of y
3. if y is empty, then append the element on the root of the tree

Deletion

To delete a node z, which node is actually removed is depend on the number of childs that node z has, and there are 3 cases:

1. if z has no child, we just remove z
2. if z has 1 child, we remove z and make the only child of z to be the child of parent(z)
3. if z has 2 children, then we find z' s successor y,which must be in z' s right subtree, and have y take the position of z.The rest of z' s original right subtree becomes y' s new right subtree, and z' s left subtree becomes y' s new left subtree.

When performing deletion, these 3 cases can be concluded into 4 cases:

1. if z has no left child, then replace z with its right child.(if right-child(z) is empty, then we deal with z has no child; if right-child(z) is not empty, then we deal with z has only right child.)
2. if z has only a left child, we simply replace z with left-child(z)
3. Otherwise, we find z' s successor y, that lies in the right subtree of z
   - if right-child(z) = y, then replace z by y,and do not change y' s right child.
   - Otherwise, if y lies in the right subtree of z , but right-child(z) is not equal y, we first replace y by its own right child, then replace z by y.

2.2 AVL Tree

2.2.1 Definition

AVL tree is a self-balanced binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. If they differ by more than one anytime, rebalancing will be done to restore this property. [1]
To simplize the problem, we define the ”Balance factor” of all nodes in a tree.

Balance factor

The balance factor of a node N is defined to be the height difference of its two child subtrees.

$$\text{BF}^{2}(N) = \text{Height(LeftSubtree}(N)) - \text{Height(RightSubtree}(N))$$

2.2.2 Operations

Searching

If a tree T is an AVL tree, a searching operation of T will not change the balance factor of any node in T. Thus, the way to search an element in AVL Tree is the same as in normal binary search tree.

Insertion

When inserting an element into an AVL tree, we initially follow the same process as inserting into a binary search tree. However, the balance factor will be changed because of the insertion. It is necessary to check each of the node’s ancestors.
Without loss of generality, we assume the inserted node is on the right of tree T. The changed tree can be classified into 6 cases:

[1] Wikipedia, ”AVL tree”, en.wikipedia.org/wiki/AVL_tree
[2] BF means Balance Factor

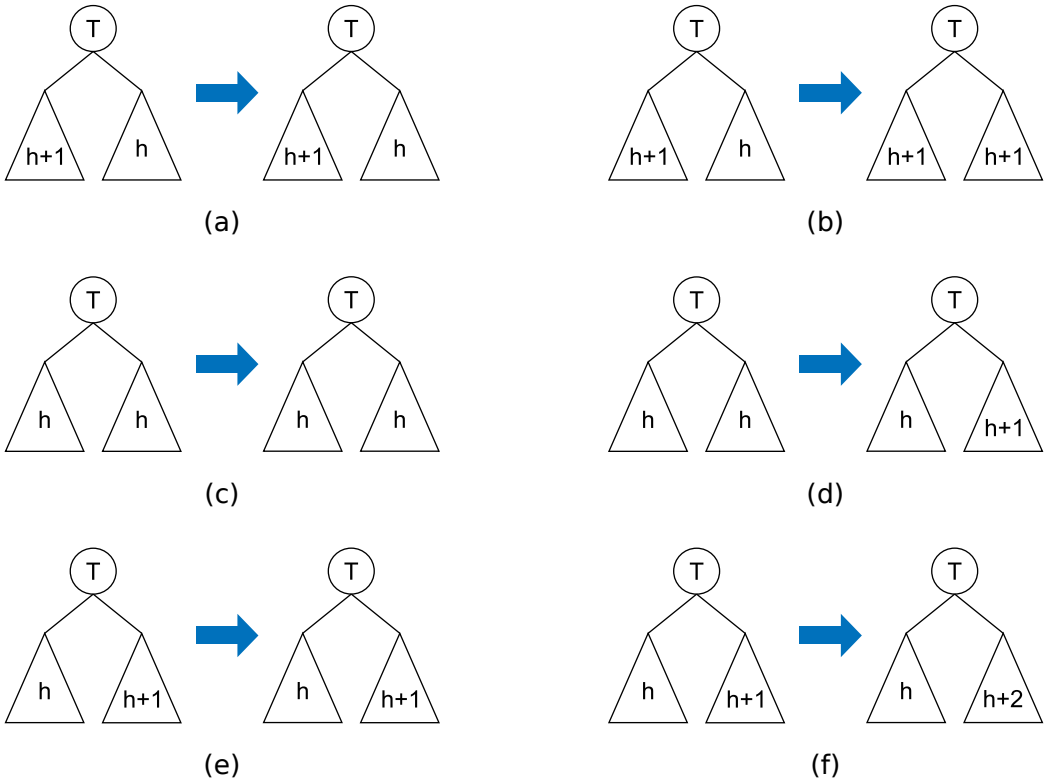


Figure 2.1: 6 cases of changed trees

The cases (a) to (e) are still a AVL tree, but the balance factor of case (f) is (-2). It is necessary to change the structure of the tree T of (f). The unbalanced tree like third case could be more clearly classify into 3 cases:

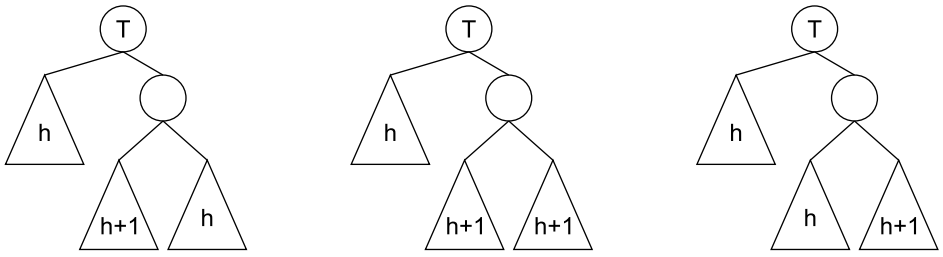


Figure 2.2: 3 cases of unbalanced trees

There is a way to change the structure of a tree. The way is the so-called tree rotations, because they move the elements only ”vertically”, so that the ”horizontal” in-order sequence of the elements is fully preserved (which is essential for a binary-search tree) [4]

[3]The case 2 will not appear when inserting a new data, because the case 2 must come from case 1 or 3. But before this insertion, the tree will be rebalanced to be a AVL tree.

[4]Knuth Donald E, Sorting and searching (2. ed) 458–481

Algorithm 1 Left rotation

```
1: function left-rotation(x)
2:     tree y := x.right-child
3:     x.right-child := y.left-child
4:     y.left-child := x
5:     update-information(y.left-child)
6:     update-information(y)
7:     return y
8: end function
```

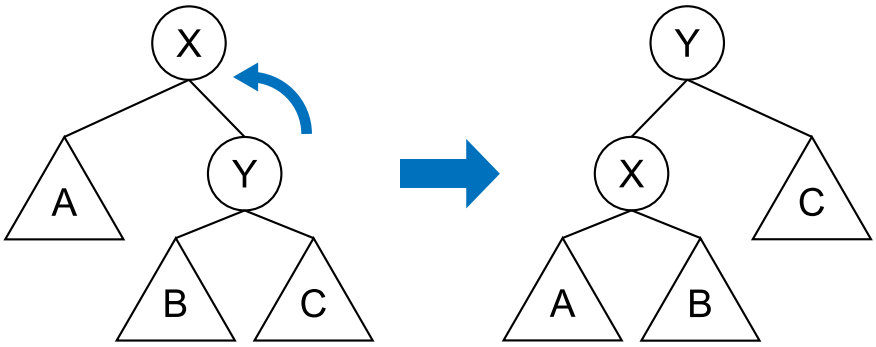


Figure 2.3: left rotation

Algorithm 2 Right rotation

```
1: function right-rotation(x)
2:     tree y := x.left-child
3:     x.left-child := y.right-child
4:     y.right-child := x
5:     update-information(y.right-child)
6:     update-information(y)
7:     return y
8: end function
```

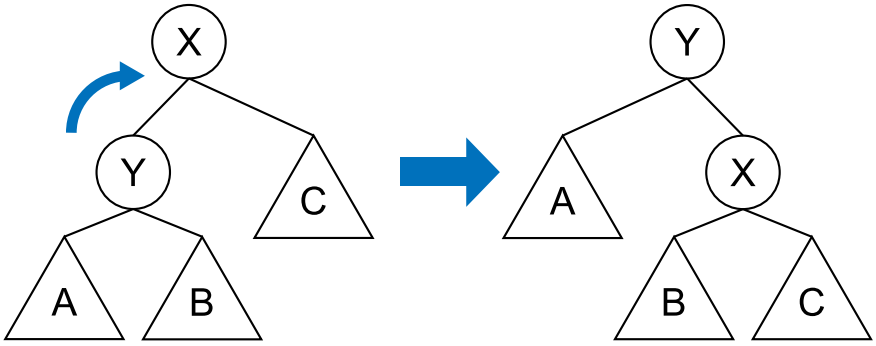


Figure 2.4: right rotation

By rotating the tree, the unbalanced tree will be changed into an AVL tree.

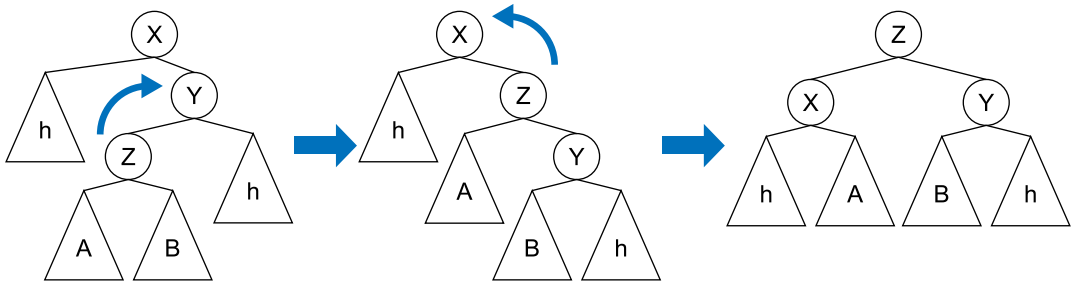
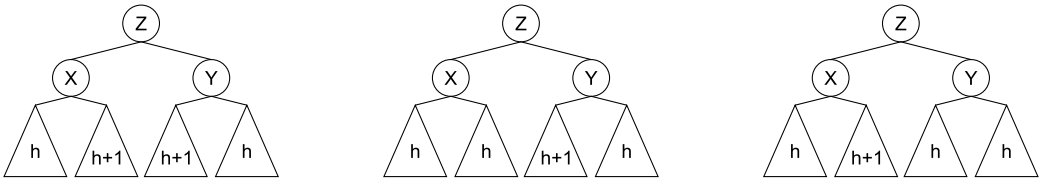


Figure 2.5: case 1(double rotation)



(a) case 1-1 (b) case 1-2 (c) case 1-3

Figure 2.6: all results of case 1

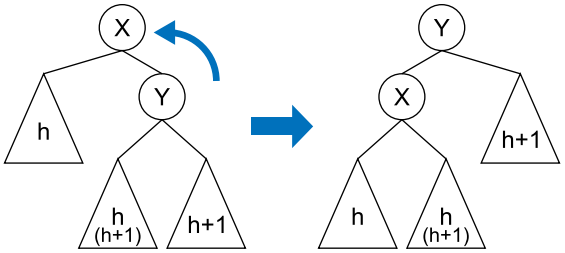


Figure 2.7: case 2, 3(single rotation)

Finally, the tree should be rotated recursively from inserted node to the root of tree. After that, the unbalanced tree will be changed back to an AVL tree.

Algorithm 3 insert

```
1: function insert(T, N)
2:     if T is empty then
3:         T := N
4:     else
5:         if N.element < T.element then
6:             insert(T.left-child, N)
7:         else
8:             insert(T.right-child, N)
9:         end if
10:        rebalance(T)
11:    end if
12: end function
```

Deletion

The deletion in AVL tree is same as normal binary search tree. But after deleting an element in AVL tree, the balance factor will be changed. We have to rebalance the tree like what we do after insert an element into the tree. Without loss of generality, we assume the deleted node is on the left of tree T. The changed tree can be classified into 6 cases:

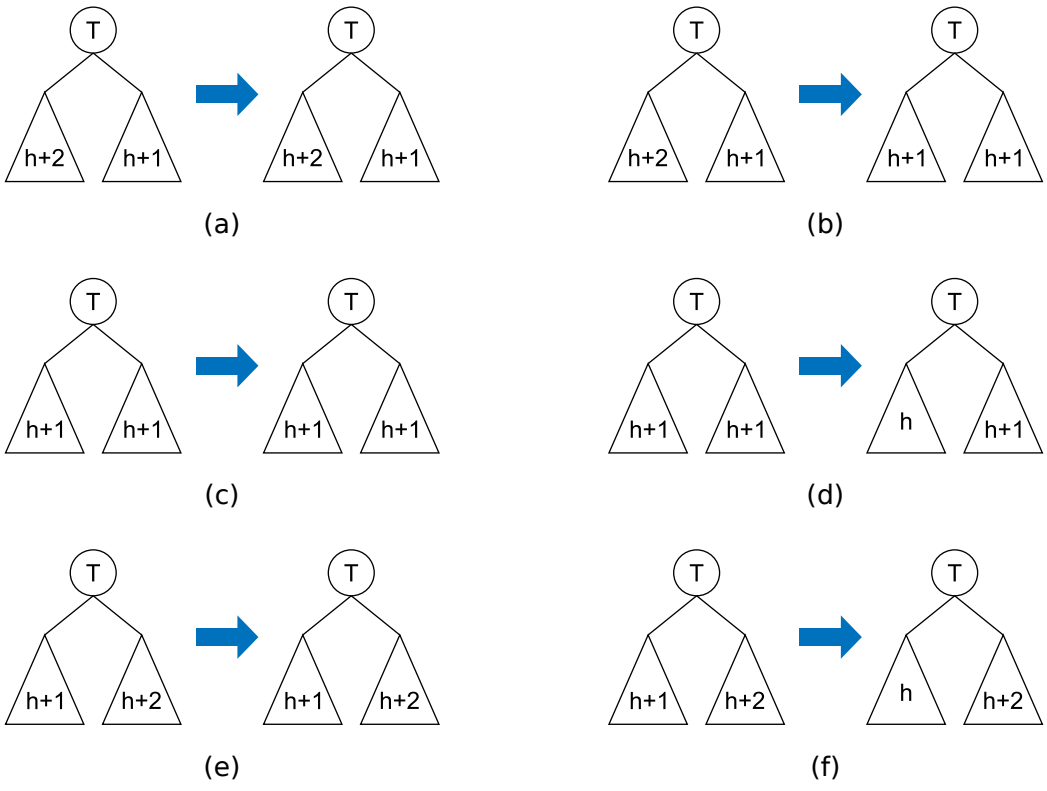


Figure 2.8: 6 cases of changed trees

As we can see, the case (a) to (e) is still an AVL tree. The strategy to rebalance the tree of case (f) is totally same as mentioned above.

Algorithm 4 delete

```
1: function delete(T, N)
2:     if N.element= T.element then
3:         if T.left-child is empty then
4:             k := T
5:             T := T.right-child
6:             delete-from-memory(k)
7:         else if T.right-child is empty then
8:             k := T
9:             T := T.left-child
10:            delete-from-memory(k)
11:        else
12:            k :=find-min(T.right-child)
13:            T.element := k.element
14:            delete(T.right-child, N)
15:        end if
16:    else
17:        if N.element < T.element then
18:            delete(T.left-child, N)
19:        else
20:            delete(T.right-child, N)
21:        end if
22:    end ifrebalance(T)
23: end function
```

2.3 Splay Tree

2.3.1 Definition

The basic idea of the splay tree is that after a node is accessed, it is pushed to the root by a series of AVL tree rotations. If a node is deep, there are many nodes on the path that are also relatively deep, and by restructuring we can make future accesses cheaper on all these nodes. Thus we can get a good time bound in theory.

2.3.2 Operations

Splaying

This operation can push any node to the root through a series of tree rotations which are similar to the rotations in the AVL Tree method. In general we divide the rotations into two main types: single rotation and double rotation. They are not exactly the same as those in the AVL

method. Within the double rotation type, we have 2 cases: zig-zig and zig-zag. They are shown by the graphs below.

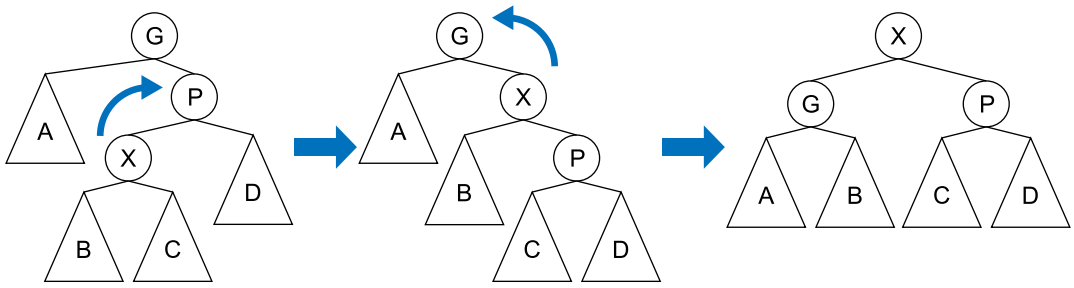


Figure 2.9: zig-zag

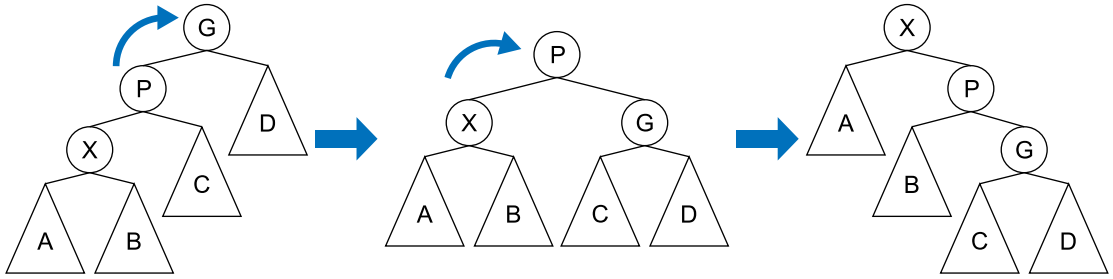


Figure 2.10: zig-zig

While implementing this operation we can use a loop to push the node all the way up through these rotations until the node reaches the root.

Searching

This operation finds the position of the node that contains the specific element and push it to the root. Its implementation is mostly the same as that in binary search tree, except that it calls the splay function at the end.

Insertion

We initially follow the same process as inserting into a binary search tree. After the element is successfully inserted, we perform a splay on the node. If the elements is already in the tree, we will push the node to the root.

Deletion

We can perform deletion by using the $Find$ operation to push it to the root. Then we apply FindMax on its left subtree. We can finish the

deletion by freeing the root and combining the left subtree with the right subtree.

Chapter 3

Testing Results

3.1 Unbalanced Binary Search Tree

3.1.1 Case 1

N	Time	TotalTime
1000	571	0.571
1250	861	0.861
1500	1281	1.281
1750	1730	1.730
2000	2397	2.397
2500	3657	3.657
3000	5523	5.523
3500	7627	7.267
4000	9668	9.668
4500	12082	12.082
5000	15167	15.167
5500	18250	18.250
6000	21654	21.654
6500	26042	26.042
7000	30053	30.053
8000	39165	39.165
9000	49227	49.227
10000	63018	63.018

Figure 3.1: testing result of case 1

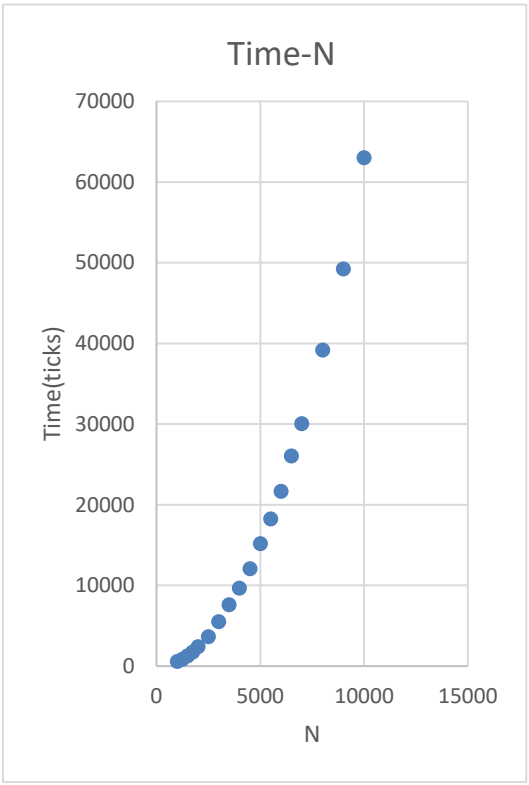


Figure 3.2: plot of case 1

3.1.2 Case 2

N	Time	TotalTime
1000	1102	1.102
1250	1750	1.750
1500	2572	2.752
1750	3464	3.464
2000	4681	4.681
2500	7231	7.231
3000	10710	10.710
3500	14764	14.764
4000	19237	19.237
4500	24177	24.177
5000	30414	30.414
5500	36474	36.474
6000	43682	43.682
6500	51246	51.246
7000	60153	60.153
8000	78644	78.644
9000	102572	102.572
10000	124861	124.681

Figure 3.3: testing result of case 2

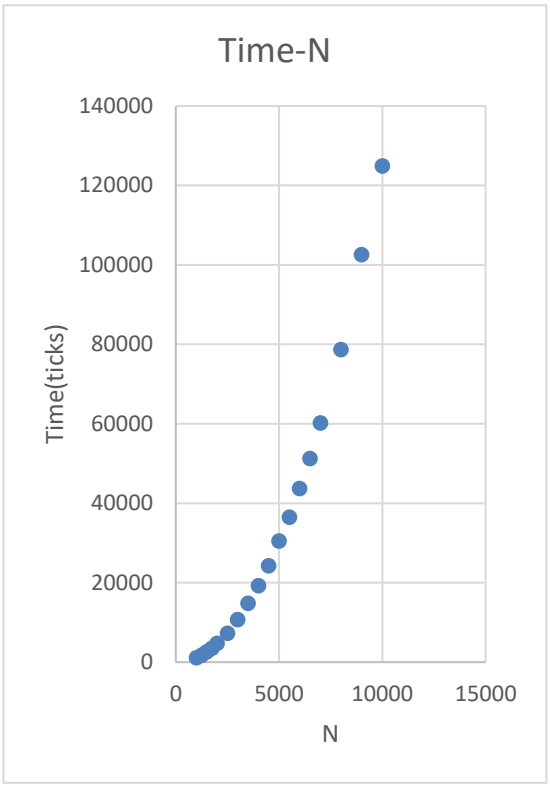


Figure 3.4: plot of case 2

3.1.3 Case 3

N	Time	TotalTime
1000	63	0.063
1250	81	0.081
1500	97	0.097
1750	118	0.118
2000	136	0.136
2500	173	0.173
3000	211	0.211
3500	262	0.262
4000	300	0.300
4500	390	0.390
5000	388	0.388
5500	422	0.422
6000	467	0.467
6500	517	0.517
7000	570	0.570
8000	662	0.662
9000	784	0.784
10000	852	0.852

Figure 3.5: testing result of case 3

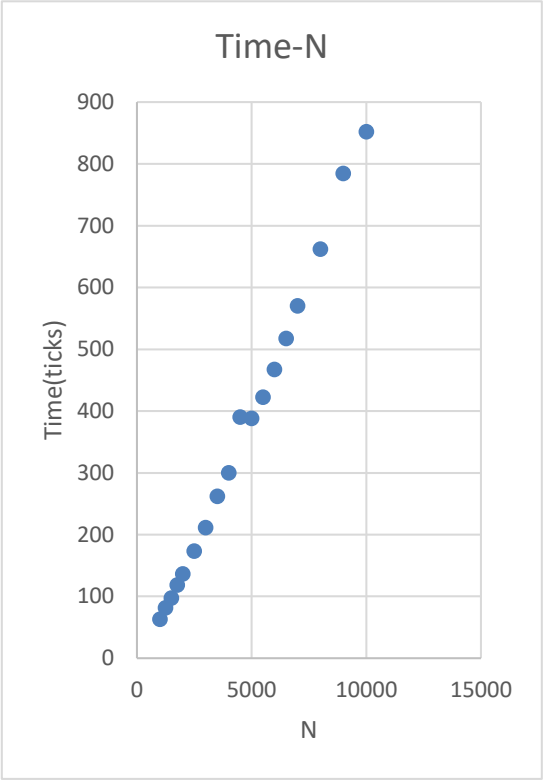


Figure 3.6: plot of case 3

3.2 AVL Tree

3.2.1 Case 1

N	Time	TotalTime
1000	82	0.082
1250	937	0.093
1500	112	0.112
1750	130	0.130
2000	152	0.152
2500	198	0.198
3000	238	0.238
3500	299	0.299
4000	332	0.332
4500	379	0.379
5000	416	0.416
5500	458	0.458
6000	511	0.511
6500	556	0.556
7000	595	0.595
8000	677	0.677
9000	773	0.773
10000	859	0.859

Figure 3.7: testing result of case 1

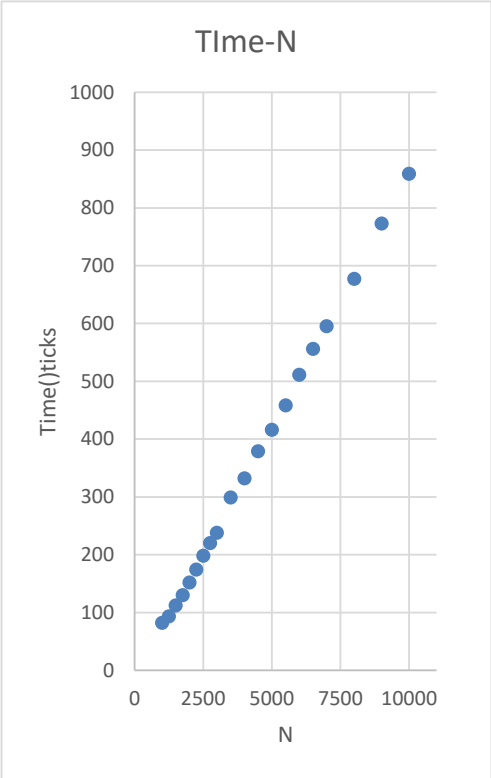


Figure 3.8: plot of case 1

3.2.2 Case 2

N	Time	TotalTime
1000	76	0.076
1250	97	0.097
1500	116	0.116
1750	139	0.139
2000	155	0.155
2500	203	0.203
3000	250	0.250
3500	294	0.294
4000	336	0.336
4500	384	0.384
5000	436	0.436
5500	487	0.487
6000	527	0.527
6500	572	0.572
7000	619	0.619
8000	714	0.714
9000	823	0.823
10000	912	0.912

Figure 3.9: testing result of case 2

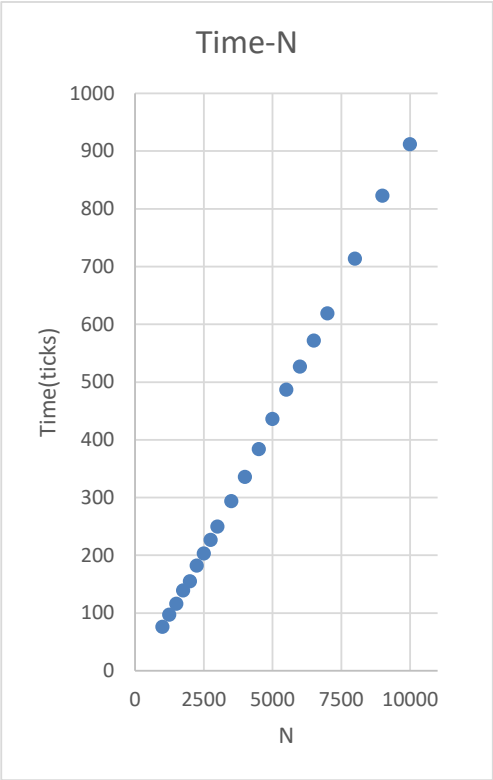


Figure 3.10: plot of case 2

3.2.3 Case 3

N	Time	TotalTime
1000	101	0.101
1250	130	0.130
1500	160	0.160
1750	190	0.190
2000	224	0.224
2500	284	0.284
3000	347	0.347
3500	414	0.414
4000	481	0.481
4500	545	0.545
5000	614	0.614
5500	686	0.686
6000	754	0.754
6500	798	0.798
7000	922	0.922
8000	1030	1.030
9000	1187	1.187
10000	1346	1.346

Figure 3.11: testing result of case 3

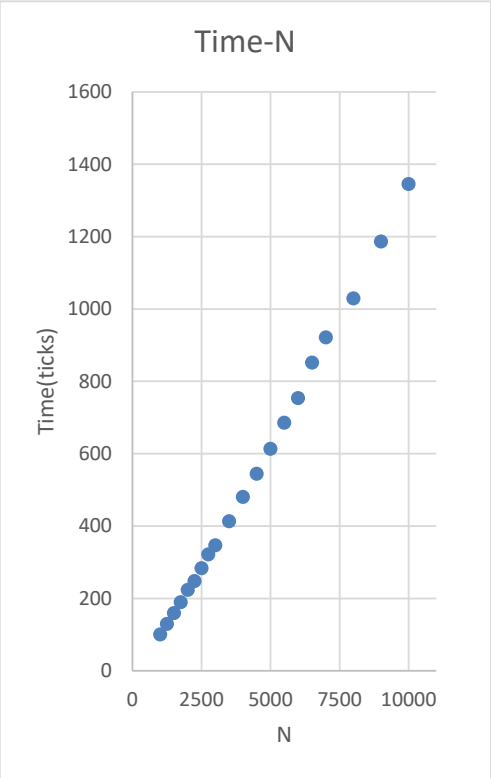


Figure 3.12: plot of case 3

3.3 Splay Tree

3.3.1 Case 1

N	Time	TotalTime
1000	43	0.043
1250	44	0.044
1500	53	0.053
1750	65	0.065
2000	72	0.072
2500	83	0.083
3000	104	0.104
3500	141	0.141
4000	140	0.140
4500	158	0.158
5000	170	0.170
5500	206	0.206
6000	202	0.202
6500	226	0.226
7000	238	0.238
8000	274	0.274
9000	303	0.303
10000	371	0.371

Figure 3.13: testing result of case 1

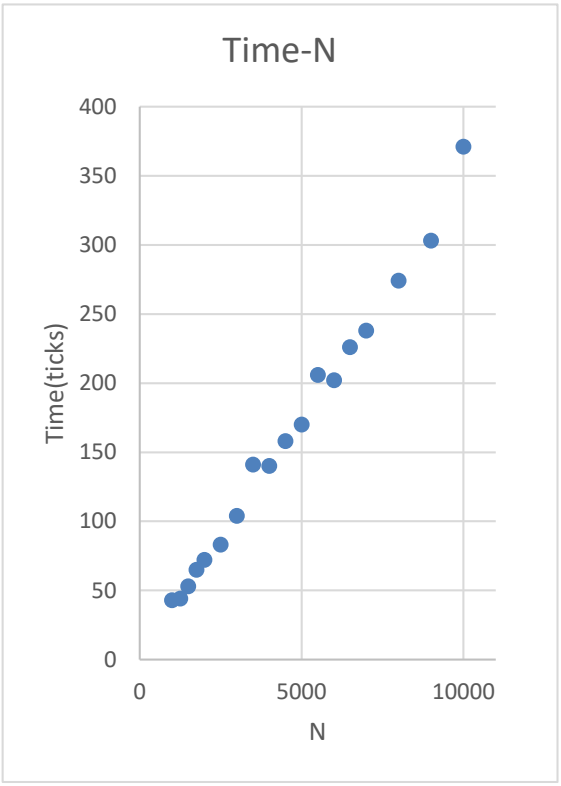


Figure 3.14: plot of case 1

3.3.2 Case 2

N	Time	TotalTime
1000	23	0.023
1250	31	0.031
1500	34	0.034
1750	41	0.041
2000	45	0.045
2500	56	0.056
3000	75	0.075
3500	79	0.079
4000	88	0.088
4500	100	0.100
5000	112	0.112
5500	127	0.127
6000	133	0.133
6500	159	0.159
7000	153	0.153
8000	187	0.187
9000	214	0.214
10000	219	0.219

Figure 3.15: testing result of case 2

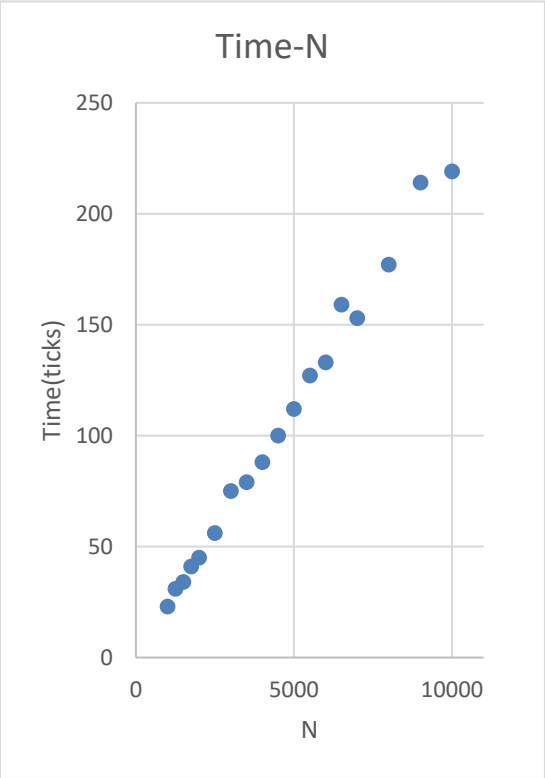


Figure 3.16: plot of case 2

3.3.3 Case 3

N	Time	TotalTime
1000	163	0.163
1250	227	0.227
1500	260	0.260
1750	308	0.308
2000	360	0.360
2500	465	0.465
3000	571	0.571
3500	698	0.698
4000	792	0.792
4500	937	0.937
5000	1033	1.033
5500	1158	1.158
6000	1317	1.317
6500	1411	1.411
7000	1516	1.516
8000	1779	1.779
9000	2040	2.040
10000	2290	2.290

Figure 3.17: testing result of case 3

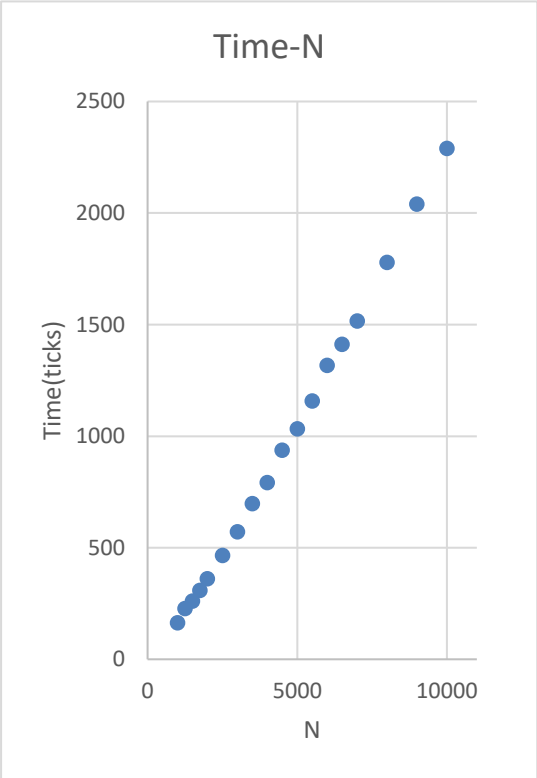


Figure 3.18: plot of case 3

Chapter 4

Analysis and Comments

4.1 AVL tree

4.1.1 the height of AVL tree

The time complexity of operation (searching, insertion and deletion) in a binary search tree always depend on the height of the tree. To know the time complexity, the most important thing is to analize the height of AVL tree.
Let N_i be the fewest number of nodes in a AVL tree T_i of height i. The children of T_{i+2} must be T_{i+1}, T_i; otherwise, the number of nodes in T_{i+2} will be more than $1 + N_{i+1} + N_i$. Thus, there is a relationship between N_{i+2}, N_{i+1}, N_i.

$$N_{i+2} = 1 + N_{i+1} + N_i, N_0 = 0, N_1 = 1$$

Add 1 at two sides of the equation.

$$(N_{i+2} + 1) = (N_{i+1} + 1) + (N_i + 1)$$

This equation is equalvant to Fibonacci numbers.

$$F_{i+2} = F_{i+1} + F_i$$

N_i can be expressed by a Fibonacci number.

$$N_i = F_{i+2} - 1$$

Fibonacci number theory gives that

$$F_i \simeq \frac{1}{\sqrt{5}}[\frac{1+\sqrt{5}}{2}]^i$$

The number of nodes N_h in tree T_h of height h

$$N_h \simeq \frac{1}{\sqrt{5}}[\frac{1+\sqrt{5}}{2}]^{h+2} - 1$$

$$\Rightarrow h = O(logN_h)$$

4.1.2 time complexity

Searching, inserting and deleting an element in AVL tree is same as it in normal binary search tree. The time complexity is $O(\text{height})$. The height of AVL tree is $O(logN)$ and the time complexity of rotation is $O(1)$, so the time complexity is $O(logN)$.

4.1.3 space complexity

Because the number of nodes in a tree is equal the number of the elements inserted, the space complexity of all binary tree is $O(N)$.

4.2 Splay tree

4.2.1 Amortized Analysis of Splay Trees

In this section we will give the general proof of the O(log N) amortized bound for the splaying step. If a tree rotation is being performed at node X, then prior to the rotation P is its parent and G is its grandparent (if X is not a child of the root). First we find a proper potential function φ:

$$\varphi(T) = \sum_{i \in T} logS(i)$$

S(i) represents the number of descendants of i (including i itself). To simplify the notation, we define:

$$R(i) = logS(i)$$

This makes:

$$\varphi(T) = \sum_{i \in T} R(i)$$

We call R(i) the rank of node i. The point is that while a rotation can change the heights of many nodes in the tree, only X, P and G can have their ranks changed.

lemma

If $a + b \leq c$, and a and b are both positive integers, then
$loga + logb \leq 2logc - 2$.

proof

$$\sqrt{ab} \leq \frac{a+b}{2} \Rightarrow \sqrt{ab} \leq \frac{c}{2} \Rightarrow ab \leq \frac{c^2}{4} \Rightarrow loga + logb \leq 2logc - 2$$

Let $R_i(X)$ and $S_i(X)$ be the rank and size of X before the splaying step, and let $R_f(X)$ and $S_f(X)$ be the rank and size of X immediately after

the splaying step. For the zig step, the actual time is 1 (for the single rotation), and the potential change is $R_f(X) + R_f(P) - R_i(X) - R_i(P)$. Thus:

$$AT_{zig} = 1 + R_f(X) + R_f(P) - R_i(X) - R_i(P)$$

Because $R_i(P) \geq R_f(P), R_f(X) \geq R_i(X)$

$$AT_{zig} \leq 1 + 3(R_f(X) - R_i(X))$$

Similarly for zig-zag step and zig-zig step, we get:

$$AT_{zig-zag} \leq 3(R_f(X) - R_i(X))$$

$$AT_{zig-zig} \leq 3(R_f(X) - R_i(X))$$

The amortized cost of an entire splay is the sum of the amortized costs of each splay step. We see that the total amortized cost to splay at node X is at most

$$1 + 3(R_f(X) - R_i(X))$$

Since the last splaying step leaves X at the root, we obtain an amortized bound of

$$1 + 3(R_f(T) - R_i(X))$$

which is O(log N). Because every operation on a splay tree requires a splay, the amortized cost of any operation is within a constant factor of the amortized cost of a splay. Thus, all splay tree operation take O(log N)amortized time. Proof complete. [1]

[1]Data Structure and Algorithm Analysis in C – Mark Allen Weiss

Appendices

Appendix A

Source Code (in C/C++)

BST.h

```
#ifndef _BST_H
#define _BST_H
typedef struct BstNode *Pos;
typedef struct BstNode *BST;

struct BstNode {
    int Element;
    BST Left;
    BST Right;
};

Pos FindMin(BST T);
//Find out the minimum and return the pointer.

Pos FindMax(BST T);
//Find out the maximum and return the pointer.

bool isExisted(int X, BST T);
//Element X is in the tree->return 1.

BST Init_Tree(BST T);
//Make an empty tree.

BST Insert(int X, BST T);
//Insert an element into the BST.

BST Delete(int X, BST T);
//Delete a certain element in the BST.

void Traverse(BST T);
//Print the preorder,inorder,postorder traversal.

#endif
```

BST.cpp

```cpp
#include "BST.h"
#include <iostream>
using namespace std;

//Functions of Traversal
void PrintNode(BST T) {
    cout << T->Element << " ";
}

void PreorderTraverse(BST T) {
    PrintNode(T);
    if (T->Left) PreorderTraverse(T->Left);
    if (T->Right) PreorderTraverse(T->Right);
}

void InorderTraverse(BST T) {
    if (T->Left) InorderTraverse(T->Left);
    PrintNode(T);
    if (T->Right) InorderTraverse(T->Right);
}

void PostorderTraverse(BST T) {
    if (T->Left) PostorderTraverse(T->Left);
    if (T->Right) PostorderTraverse(T->Right);
    PrintNode(T);
}

void Traverse(BST T) {
    if (T == NULL) {
    cout << "EMPTY TREE!" << endl;
    return;
}
    cout << "Preorder: " << endl;
    PreorderTraverse(T);
    cout << endl;
    cout << "Inorder: " << endl;
    InorderTraverse(T);
    cout << endl;
    cout << "Postorder: " << endl;
    PostorderTraverse(T);
    cout << endl;
}

bool isExisted(int X, BST T) {
    if (T == NULL) return 0;
    if (X == T->Element) return 1;
    if (X > T->Element) return isExisted(X, T->Right);
```

```
48      if (X < T->Element) return isExisted(X, T->Left);
49  }
50
51  Pos FindMin(BST T) {
52      if (T != NULL)
53          while (T->Left != NULL) T = T->Left;
54      return T;
55  }
56
57  Pos FindMax(BST T) {
58      if (T != NULL)
59          while (T->Right != NULL) T = T->Right;
60      return T;
61  }
62
63  BST Init_Tree(BST T) {
64      if (T != NULL) {
65          Init_Tree(T->Left);
66          Init_Tree(T->Right);
67          delete T;
68      }
69      return NULL;
70  }
71
72  //Function of insertion
73  BST Insert(int X, BST T) {
74  //Return a one-node tree.
75      if (T == NULL) {
76          T = new BstNode;
77          T->Element = X;
78          T->Left = T->Right = NULL;
79      }
80      else if (X < T->Element) {
81          T->Left = Insert(X, T->Left);//Recursive
                Insertion to the Left.
82      }
83      else if (X > T->Element) {
84          T->Right = Insert(X, T->Right);//Recursive
                Insertion to the Right.
85      }
86      return T;
87  }
88
89  //delete
90  BST Delete(int X, BST T) {
91      Pos tmp;
92      if (X < T->Element) T->Left = Delete(X, T->Left);
93      else if (X > T->Element) T->Right = Delete(X, T->
```

```
        Right);
    /*
    The element to be deleted has been found.
    case: Two children.
    */
    else if (T->Left&&T->Right) {
    /*Replace with smallest in right subtree*/
        tmp = FindMin(T->Right);
        T->Element = tmp->Element;
        T->Right = Delete(T->Element, T->Right);
    }
    /*case: One or zero children*/
    else {
        tmp = T;
        if (T->Left == NULL) {
            T = T->Right;
        }
        else if (T->Right == NULL) {
        T = T->Left;
        }
        delete tmp;
    }

    return T;
}
```

test_pfms_Bst.cpp

```
#include "BST.h"
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
const int iteration = 200;
using namespace std;

int main() {
    clock_t start, stop;
    BST T = NULL;
    cout << "Test performance of BST." << endl;
    cout << "Iteration: 200" << endl;
    cout << "Case one: Insert N integers in increasing
       order\
    and delete them in the same order." << endl;
    printf("%-10s", "Datasize");
    printf("%-7s", "Ticks");
    printf("%-14s", "TotalTime(s)");
    printf("%s\n", "Duration(s)");
    cout << "_____"
```

```
        << endl;
21      for (int n = 1000; n <= 10000; n += 250) {
22          int ticks;
23          start = clock();
24              for (int it = 0; it < iteration; it++) {
25                  for (int i = 1; i <= n; i++) {
26                      T = Insert(i, T);
27                  }
28                  for (int i = 1; i <= n; i++) {
29                      T = Delete(i, T);
30                  }
31              }
32          stop = clock();
33          printf("%-10d", n);
34          printf("%-7d", stop - start);
35          printf("%-14.6f", (double)(stop - start) /
               CLK_TCK);
36          printf("%.6f\n", (double)(stop - start) / CLK_TCK
               / iteration);
37      }
38      cout << endl;
39
40
41      cout << "Case two: Insert N integers in increasing
           order\
42      and delete them in the reverse order." << endl;
43      printf("%-10s", "Datasize");
44      printf("%-7s", "Ticks");
45      printf("%-14s", "TotalTime(s)");
46      printf("%s\n", "Duration(s)");
47      cout << "-------------------------------------------"
           << endl;
48      for (int n = 1000; n <= 10000; n += 250) {
49          int ticks;
50          start = clock();
51          for (int it = 0; it < iteration; it++) {
52              for (int i = 1; i <= n; i++) {
53                  T = Insert(i, T);
54              }
55              for (int i = n; i >= 1; i--) {
56                  T = Delete(i, T);
57              }
58          }
59          stop = clock();
60          printf("%-10d", n);
61          printf("%-7d", stop - start);
62          printf("%-14.6f", (double)(stop - start) /
               CLK_TCK);
```

```
63          printf("%.6f\n", (double)(stop - start) / CLK_TCK
                / iteration);
64      }
65
66      cout << endl;
67
68      cout << "Case three: Insert N integers in random
           order\
69      and delete them in random order." << endl;
70      printf("%-10s", "Datasize");
71      printf("%-7s", "Ticks");
72      printf("%-14s", "TotalTime(s)");
73      printf("%s\n", "Duration(s)");
74      cout << "-------------------------------------------------"
           << endl;
75      srand(unsigned(time(NULL)));
76      for (int n = 1000; n <= 10000; n += 250) {
77          //Generate n distinct, random ordered element.
78          int *p = new int[n];
79          //Initialize the array with 1->n.
80          for (int i = 0; i < n; i++) {
81              p[i] = i + 1;
82          }
83          //Randomize the array.
84          for (int i = 0; i < n; i++) {
85              int tmp;
86              int a = rand() % n;
87              int b = rand() % n;
88              tmp = p[a];
89              p[a] = p[b];
90              p[b] = tmp;
91          }
92
93      //Start processing
94      start = clock();
95      for (int it = 0; it < iteration; it++) {
96          for (int i = 1; i <= n; i++) {
97              T = Insert(p[i - 1], T);
98          }
99          //Randomize the array.
100         for (int i = 0; i < n; i++) {
101             int tmp;
102             int a = rand() % n;
103             int b = rand() % n;
104             tmp = p[a];
105             p[a] = p[b];
106             p[b] = tmp;
107         }
```

```
108         for (int i = 1; i <= n; i++) {
109             T = Delete(p[i - 1], T);
110         }
111     }
112     stop = clock();
113     printf("%-10d", n);
114     printf("%-7d", stop - start);
115     printf("%-14.6f", (double)(stop - start) / CLK_TCK);
116     printf("%.6f\n", (double)(stop - start) / CLK_TCK /
          iteration);
117     delete[]p;
118     }
119     system("pause");
120     return 0;
121 }
```

test_func_Bst.cpp

```
1  #include "BST.h"
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int functionID;
7      int tmp;
8
9      BST T = NULL;
10     cout << "Function Test of BST" << endl;
11     cout << "————————" << endl;
12     cout << "1.Insert" << endl
13     << "2.Delete" << endl
14     << "3.Find" << endl
15     << "4.Traverse" << endl
16     << "5.Initialize" << endl
17     << "0.Exit" << endl;
18     cout << "————————" << endl<<endl;
19     cout<< "Please input functionID:" << endl;
20
21     cin >> functionID;
22     cout << "————————" << endl;
23
24     while (functionID) {
25         switch (functionID)
26         {
27             case 1:
28                 cout << "Please input the element:"<<endl
                      ;
29                 cin >> tmp;
30                 if (isExisted(tmp, T)) {
```

```
31                      cout << "Already existed!" << endl <<
                            endl;
32                      break;
33                  }
34                  T = Insert(tmp, T);
35                  cout << "Insert element " << tmp << "
                       successfully!" << endl << endl;
36                  break;
37              case 2:
38                  cout << "Please input the element:" <<
                       endl;
39                  cin >> tmp;
40                  if (!isExisted(tmp, T)) {
41                      cout << "Not Existed!" << endl <<
                           endl;
42                      break;
43                  }
44                  else {
45                      T = Delete(tmp, T);
46                      cout << "Delete element " << tmp << "
                            successfully!" << endl << endl;
47                  }
48                  break;
49              case 3:
50                  cout << "Please input the element:" <<
                       endl;
51                  cin >> tmp;
52                  if (isExisted(tmp, T)) {
53                      cout << "Existed." << endl << endl;
54                  }
55                  else {
56                      cout << "Not Existed." << endl <<
                           endl;
57                  }
58                  break;
59              case 4:
60                  Traverse(T);
61                  cout << endl;
62                  break;
63              case 5:
64                  T = Init_Tree(T);
65                  cout << "Successfully initialized." <<
                       endl << endl;
66                  break;
67              default:
68                  cout << "Input Error" << endl << endl;
69                  break;
70          }
```

```
71          cout << "Please input functionID:" << endl;
72          cin >> functionID;
73          cout << "————————————" << endl;
74      }
75      return 0;
76  }
```

Avl_Tree.h

```
1  #ifndef _Avl_Tree_H
2  #define _Avl_Tree_H
3  typedef struct AvlNode *Pos;
4  typedef struct AvlNode *AvlTree;
5
6  struct AvlNode {
7      int Element;
8      AvlTree Left;
9      AvlTree Right;
10     int height;
11 };
12
13 Pos FindMin(AvlTree T);
14 //Find out the minimum and return the pointer.
15
16 Pos FindMax(AvlTree T);
17 //Find out the maximum and return the pointer.
18
19 AvlTree Init_Tree(AvlTree T);
20 //Make an empty tree.
21
22 bool isExisted(int X, AvlTree T);
23 //Element X is in the tree->return 1.
24
25 AvlTree Insert(int X, AvlTree T);
26 //Insert an element into the avl tree.
27
28 AvlTree Delete(int X, AvlTree T);
29 //Delete a certain element in the avl tree.
30
31 void Traverse(AvlTree T);
32 //Print the preorder,inorder,postorder traversal.
33
34 #endif
```

Avl_Tree.cpp

```
1  #include "Avl_Tree.h"
2  #include <iostream>
3  using namespace std;
```

```
4
5  AvlTree Init_Tree(AvlTree T) {
6      if (T != NULL) {
7          Init_Tree(T->Left);
8          Init_Tree(T->Right);
9          delete T;
10     }
11     return NULL;
12 }
13
14 //Functions of Insert
15 AvlTree rLeft(AvlTree T);//LL->SingleRotateWithLeft
16 AvlTree rRight(AvlTree T);//RR->SingleRotateWithRight
17 AvlTree dRLeft(AvlTree T);//LR->DoubleRotateWithLeft
18 AvlTree dRRight(AvlTree T);//RL->DoubleRotateWithRight
19 int getHeight(AvlTree T);//Get the height of a certain
      tree node.
20
21 AvlTree Insert(int X, AvlTree T) {
22 //Return a one-node tree.
23     if (T == NULL) {
24         T = new AvlNode;
25         T->Element = X;
26         T->height = 0;
27         T->Left = T->Right = NULL;
28     }
29     else if (X < T->Element) {
30         T->Left = Insert(X, T->Left);//Recursive
              Insertion to the Left.
31         if (getHeight(T->Left) - getHeight(T->Right) ==
              2) {
32             //LL
33             if (X < T->Left->Element) T = rLeft(T);
34             //LR
35             else T = dRLeft(T);
36         }
37     }
38     else if (X > T->Element) {
39         T->Right = Insert(X, T->Right);//Recursive
              Insertion to the Right.
40         if (getHeight(T->Right) - getHeight(T->Left) ==
              2) {
41             //RR
42             if (X > T->Right->Element) T = rRight(T);
43             //RL
44             else T = dRRight(T);
45         }
46     }
```

```
47      T->height = getHeight(T->Left) > getHeight(T->Right)
           ? getHeight(T->Left) + 1 : getHeight(T->Right) + 1;
48      return T;
49  }
50
51  AvlTree rLeft(AvlTree T) {
52      Pos p;
53
54      p = T->Left;
55      T->Left = p->Right;
56      p->Right = T;
57
58      T->height = getHeight(T->Left) > getHeight(T->Right)
           ? getHeight(T->Left) + 1 : getHeight(T->Right) + 1;
59      p->height = getHeight(p->Left) > getHeight(p->Right)
           ? getHeight(p->Left) + 1 : getHeight(p->Right) + 1;
60      return p;
61  }
62
63  AvlTree rRight(AvlTree T) {
64      Pos p;
65
66      p = T->Right;
67      T->Right = p->Left;
68      p->Left = T;
69
70      T->height = getHeight(T->Left) > getHeight(T->Right)
           ? getHeight(T->Left) + 1 : getHeight(T->Right) + 1;
71      p->height = getHeight(p->Left) > getHeight(p->Right)
           ? getHeight(p->Left) + 1 : getHeight(p->Right) + 1;
72
73      return p;
74  }
75
76  AvlTree dRLeft(AvlTree T) {
77      T->Left = rRight(T->Left);
78      return rLeft(T);
79  }
80
81  AvlTree dRRight(AvlTree T) {
82      T->Right = rLeft(T->Right);
83      return rRight(T);
84  }
85
86  int getHeight(AvlTree T)
87  {
88      if (T == NULL) return -1;
89      else return T->height;
```

```
90 }
91
92 //Functions of Traversal
93 void PrintNode(AvlTree T) {
94     cout << T->Element << " ";
95 }
96
97 void PreorderTraverse(AvlTree T) {
98     PrintNode(T);
99     if (T->Left) PreorderTraverse(T->Left);
100    if (T->Right) PreorderTraverse(T->Right);
101 }
102
103 void InorderTraverse(AvlTree T) {
104    if (T->Left) InorderTraverse(T->Left);
105    PrintNode(T);
106    if (T->Right) InorderTraverse(T->Right);
107 }
108
109 void PostorderTraverse(AvlTree T) {
110    if (T->Left) PostorderTraverse(T->Left);
111    if (T->Right) PostorderTraverse(T->Right);
112    PrintNode(T);
113 }
114
115 void Traverse(AvlTree T) {
116    if (T == NULL) {
117        cout << "EMPTY TREE!" << endl;
118        return;
119    }
120    cout << "Preorder: " << endl;
121    PreorderTraverse(T);
122    cout << endl;
123    cout << "Inorder: " << endl;
124    InorderTraverse(T);
125    cout << endl;
126    cout << "Postorder: " << endl;
127    PostorderTraverse(T);
128    cout << endl;
129 }
130
131 bool isExisted(int X, AvlTree T) {
132    if (T == NULL) return 0;
133    if (X == T->Element) return 1;
134    if (X > T->Element) return isExisted(X, T->Right);
135    if (X < T->Element) return isExisted(X, T->Left);
136 }
137
```

```
138 Pos FindMin(AvlTree T) {
139     if (T != NULL)
140         while (T->Left != NULL) T = T->Left;
141     return T;
142 }
143
144 Pos FindMax(AvlTree T) {
145     if (T != NULL)
146         while (T->Right != NULL) T = T->Right;
147     return T;
148 }
149
150 AvlTree Delete(int X, AvlTree T) {
151     Pos tmp;
152     if (X < T->Element) {
153         T->Left = Delete(X, T->Left);
154         //The deletion caused unbalance in subtree of
              root T.
155         if (getHeight(T->Right) - getHeight(T->Left) ==
              2) {
156             //RR
157             if (getHeight(T->Right->Right) > getHeight(T
                  ->Right->Left)) {
158                 T = rRight(T);
159             }
160             //RL
161             else {
162                 T = dRRight(T);
163             }
164         }
165     }
166     else if (X > T->Element) {
167         T->Right = Delete(X, T->Right);
168         //The deletion caused unbalance in subtree of
              root T.
169         if (getHeight(T->Left) - getHeight(T->Right) ==
              2) {
170         //LL
171             if (getHeight(T->Left->Left) > getHeight(T->
                  Left->Right)) {
172                 T = rLeft(T);
173             }
174             //LR
175             else {
176                 T = dRLeft(T);
177             }
178         }
179     }
```

```
180     /*
181     The element to be deleted has been found.
182     case: Two children.
183     */
184     else if (T->Left&&T->Right) {
185         /*Replace with smallest in right subtree*/
186         tmp = FindMin(T->Right);
187         T->Element = tmp->Element;
188         T->Right = Delete(T->Element, T->Right);
189     }
190     /*case: One or zero children*/
191     else {
192         tmp = T;
193         if (T->Left == NULL) {
194             T = T->Right;
195         }
196         else if (T->Right == NULL) {
197             T = T->Left;
198         }
199         delete tmp;
200     }
201     if (T != NULL)
202         T->height = getHeight(T->Left) > getHeight(T->
               Right) ? getHeight(T->Left) + 1 : getHeight(T->
               Right) + 1;
203     return T;
204 }
```

Splay_Tree.h

```
1  #include <iostream>
2  #ifndef _Splay_Tree_H
3  #define _Splay_Tree_H
4  typedef struct AvlNode *Pos;
5  typedef struct AvlNode *SplayTree;
6
7  struct AvlNode {
8      int Element;
9      SplayTree Left = NULL;
10     SplayTree Right = NULL;
11     SplayTree Father = NULL;
12     int height;
13 };
14
15 Pos AccessNode(int X, SplayTree T);
16 //Access one node and return its pointer.
17
18 Pos Access(int X, SplayTree T);
19 //Access one node and splay the tree.
```

```
20
21 Pos FindMin(SplayTree T);
22 //Find out the minimum and return the pointer.
23
24 Pos FindMax(SplayTree T);
25 //Find out the maximum and return the pointer.
26
27 SplayTree Init_Tree(SplayTree T);
28 //Make an empty tree.
29
30 bool isExisted(int X, SplayTree T);
31 //Element X is in the tree->return 1.
32
33 SplayTree Insert(int X, SplayTree T);
34 //Insert an element into the splay tree and splay it.
35
36 SplayTree InsertNode(int X, SplayTree T);
37 //Insert an element into the splay tree.
38
39 SplayTree Delete(int X, SplayTree T);
40 //Delete a certain element in the splay tree.
41
42 SplayTree Splay(Pos T);
43 //Splay a certain element to the root.
44
45 void Traverse(SplayTree T);
46 //Print the preorder, inorder, postorder traversal.
47
48 #endif
```

Splay_Tree.c

```
1  #include "Splay_Tree.h"
2  #include <iostream>
3  using namespace std;
4
5  SplayTree Init_Tree(SplayTree T) {
6      if (T != NULL) {
7          Init_Tree(T->Left);
8          Init_Tree(T->Right);
9          delete T;
10     }
11     return NULL;
12 }
13
14 //Functions of Traversal
15 void PrintNode(SplayTree T) {
16     cout << T->Element << " ";
17 }
```

```
18
19 void PreorderTraverse(SplayTree T) {
20     PrintNode(T);
21     if (T->Left) PreorderTraverse(T->Left);
22     if (T->Right) PreorderTraverse(T->Right);
23 }
24
25 void InorderTraverse(SplayTree T) {
26     if (T->Left) InorderTraverse(T->Left);
27     PrintNode(T);
28     if (T->Right) InorderTraverse(T->Right);
29 }
30
31 void PostorderTraverse(SplayTree T) {
32     if (T->Left) PostorderTraverse(T->Left);
33     if (T->Right) PostorderTraverse(T->Right);
34     PrintNode(T);
35 }
36
37 void Traverse(SplayTree T) {
38     if (T == NULL) {
39     cout << "EMPTY TREE!" << endl;
40         return;
41     }
42     cout << "Preorder: " << endl;
43     PreorderTraverse(T);
44     cout << endl;
45     cout << "Inorder: " << endl;
46     InorderTraverse(T);
47     cout << endl;
48     cout << "Postorder: " << endl;
49     PostorderTraverse(T);
50     cout << endl;
51 }
52
53 bool isExisted(int X, SplayTree T) {
54     if (T == NULL) return 0;
55     if (X == T->Element) return 1;
56     if (X > T->Element) return isExisted(X, T->Right);
57     if (X < T->Element) return isExisted(X, T->Left);
58 }
59
60 Pos FindMin(SplayTree T) {
61     if (T != NULL)
62     while (T->Left != NULL) T = T->Left;
63     return T;
64 }
65
```

```
66 Pos FindMax(SplayTree T) {
67     if (T != NULL)
68     while (T->Right != NULL) T = T->Right;
69     return T;
70 }
71
72 Pos p1;
73 SplayTree Insert(int X, SplayTree T) {
74     T = InsertNode(X, T);
75     return Splay(p1);
76 }
77
78 SplayTree InsertNode(int X, SplayTree T) {
79     //Return a one-node tree.
80     if (T == NULL) {
81         T = new SplayNode;
82         T->Element = X;
83         T->height = 0;
84         T->Left = T->Right = NULL;
85         T->Father = NULL;
86         p1 = T;
87     }
88     else if (X < T->Element) {
89         T->Left = InsertNode(X, T->Left);//Recursive
              Insertion to the Left.
90         T->Left->Father = T;
91     }
92     else if (X > T->Element) {
93         T->Right = InsertNode(X, T->Right);//Recursive
              Insertion to the Right.
94         T->Right->Father = T;
95     }
96     return T;
97 }
98
99 Pos AccessNode(int X, SplayTree T) {
100     if (X == T->Element) return T;
101     if (X > T->Element) return AccessNode(X, T->Right);
102     if (X < T->Element) return AccessNode(X, T->Left);
103 }
104
105 Pos Access(int X, SplayTree T) {
106     Pos p;
107     p = AccessNode(X, T);
108     return Splay(p);
109 }
110
111 //Functions of rotation
```

```
112 //clockwise
113 void Zig(Pos x) {
114     Pos tmp;
115     if (x->Father->Father) {
116         if(x->Father->Father->Right == x->Father)
117             x->Father->Father->Right = x;
118         else x->Father->Father->Left = x;
119     }
120     tmp = x->Father->Father;
121     x->Father->Father = x;
122     if (x->Right) {
123         x->Right->Father = x->Father;
124     }
125     x->Father->Left = x->Right;
126     x->Right = x->Father;
127     x->Father = tmp;
128 }
129
130 //unclockwise
131 void Zag(Pos x) {
132     Pos tmp;
133     if (x->Father->Father) {
134         if(x->Father->Father->Left == x->Father)
135             x->Father->Father->Left = x;
136         else x->Father->Father->Right = x;
137     }
138     tmp = x->Father->Father;
139     x->Father->Father = x;
140     if (x->Left) {
141         x->Left->Father = x->Father;
142     }
143     x->Father->Right = x->Left;
144     x->Left = x->Father;
145     x->Father = tmp;
146 }
147
148 SplayTree Splay(Pos x) {
149     if (x->Father == NULL) return x;
150     //Nonrecursive solution
151     //To splay the current node to the root.
152     while (x->Father != NULL) {
153         Pos p = x->Father;
154         Pos gp = p->Father;
155         //The grandparent is still existed.
156         if (gp != NULL) {
157             //LL
158             if (gp->Left == p && p->Left == x) {
159                 Zig(p); Zig(x);
```

```
160             }
161             //RR
162             else if (gp->Right == p && p->Right == x) {
163                 Zag(p); Zag(x);
164             }
165             //LR
166             else if (gp->Left == p && p->Right == x) {
167                 Zag(x); Zig(x);
168             }
169             //RL
170             else if (gp->Right == p && p->Left == x) {
171                 Zig(x); Zag(x);
172             }
173         }
174         //Parent is the root.
175         else {
176             if (p->Left == x) {
177                 Zig(x);
178             }
179             else {
180                 Zag(x);
181             }
182         }
183         if (x->Father == NULL) return x;
184     }
185 
186 }
187 
188 Pos p2 = NULL;
189 
190 SplayTree DeleteNode(int X, SplayTree T) {
191     Pos tmp;
192     if (X < T->Element) T->Left = DeleteNode(X, T->Left);
193     else if (X > T->Element) T->Right = DeleteNode(X, T->
        Right);
194     /*
195     The element to be deleted has been found.
196     case: Two children.
197     */
198     else if (T->Left&&T->Right) {
199         /*Replace with smallest in right subtree*/
200         tmp = FindMin(T->Right);
201         T->Element = tmp->Element;
202         T->Right = DeleteNode(T->Element, T->Right);
203     }
204     /*case: One or zero children*/
205     else {
206         tmp = T;
```

```
207         if (T->Left == NULL) {
208             if (T->Right) T->Right->Father = T->Father;
209             T = T->Right;
210 
211         }
212         else if (T->Right == NULL) {
213             if (T->Left) T->Left->Father = T->Father;
214             T = T->Left;
215         }
216         p2 = T;
217         delete tmp;
218     }
219 
220     return T;
221 }
222 
223 SplayTree Delete(int X, SplayTree T) {
224     T = Access(X, T);
225     T = DeleteNode(X, T);
226     return T;
227 }
```

Appendix B

Author List, Declaration and Signatures

Author List

Splay Tree: Wang Zhongwei
Unbalanced Tree: Lin Jiafeng
AVL Tree: Wang Zhongwei , Lin Jiafeng

Declaration

We hereby declare that all the work done in this project titled ”Binary Search Trees” is of our independent effort as a group.

Signatures