Binary Search Trees

Advance Data Structure

Project 1

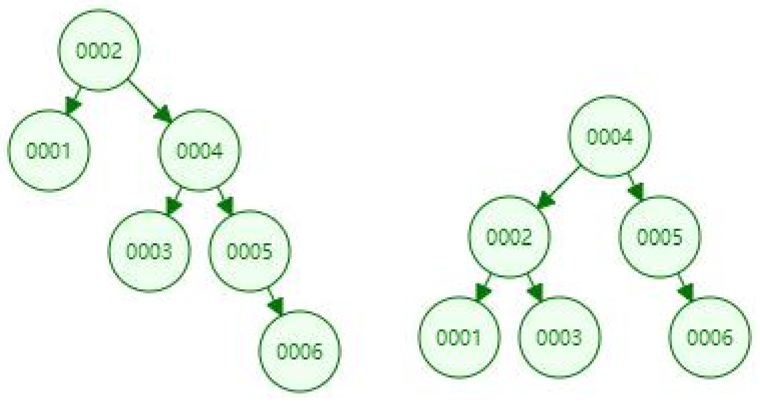
Yizhou Chen Zewei Cai Jinze Wu

2019-3-1

Chapter 1 Introduction

1.1 Background Information

As we all know, the main roles for the binary search tree(BST) are search and dynamic sorting, and the time complexity for a BST to insert/find/delete is O (log (n). However, it's usually not so fast when you actually use it, beacuse the insertion order which you use is not always so accurate, especially when the order of insertion data is ordered or basically ordered, the binary tree will be seriously unbalanced. Fortunately, AVL Tree and Splay Tree can be used to solve this problem.



- AVL Tree is a binary search tree which the heights of the two child subtrees of any node differ by at most one.
- Splay Tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.

1.2 Problem Description

General Description

The problem is that we need to measure the performance of three kinds of trees ——unbalanced binary search trees, AVL trees, and splay trees. We will test it by measure the run times in three different situation——ascending insertion and deletion, ascending insertion and descending deletion, random insertion and random deletion.

From the program, we can get the data of time comsumption of each tree, for which we will compare to analyze and compare the performance of three different trees.

Chapter 2

2.1 Structure Description

2.1.1 Specification:

The structure is used in the program, which is used to build tree different trees and to operate the insertion and deletion.

2.1.2 Specification:

```
Struct TreeNode
1:  int Element, Height
2:  Pointer Left, Right, Parent
```

2.2 Main Sketch of the Program

2.2.1 Specification:

The whole program including a main function and many extra functions. In the main function, firstly we input N, the total number of the nodes, then we create four different arrays which are used for tree’ s input. Finally, we test the time consumption of three different trees in three different situations.

2.2.2 Pseudo code:

```
Main Function
1: function main ( )
2:   Input : N ← total number of the nodes
3:       Call CreateArray( ) ← four different arrays in three different situations
4:       Call TestTime( ) ← test running time in 12 kinds of conditions
5: End function
```

2.3 Module of BST

2.3.1 Specification:

This module can be divided two parts including insertion and deletion of BST. In each part, we also test the time consumption.

2.2.2 Pseudo code:

Function BSTInsert

```
1: function BSTInsert(Tree T, int X)
2:     If T==NULL then Create New Node
3:     else if X < T->Element then
4:         T->Left ← AVLInsert(T->Left, X)
5:     else if X > T->Element then
6:         T->Right = AVLInsert(T->Right, X)
7:     return T
8: end function
```

Function BSTDelete

```
1: function BSTDelete(Tree T, int X)
2:     if T==NULL then return NULL
3:     else
4:         if X < T->Element then
5:             T->Left ← BSTDelete(T->Left, X)
6:         else if X > T->Element then
7:             T->Right ← BSTDelete(T->Right, X)
8:         else if T has both left and right subtrees then
9:             T->Element ← FindMin(T->Right)->Element
10:            T->Right ← BSTDelete(T->Right, T->Element)
11:        else delete T
12:    return T
13: end function
```

2.4 Module AVL Tree

2.4.1 Specification:

The module can be divided into two parts including insertion and deletion of AVL Tree. In each part, we also test the time consumption. For one thing, we insert an element according to the principle of binary search tree, and then update the height of the affected nodes. If the balanced factor of the node is greater than 1 or less than - 1, the node will be rotated. For another, when we delete a node, we immediately adjust the height of the affected node and adjust.

2.4.2 Pseudo code:

Function AVLInsert

```
1:  function AVLInsert(Tree T, int X)
2:      If T==NULL then Create New Node
3:      else if X < T->Element then
4:          T->Left ← AVLInsert(T->Left, X)
5:          if BF of node T is 2 then
6:              if X < T->Left->Element then
7:                  LL Rotation
8:              else
9:                  LR Rotation
10:     else if X > T->Element then
11:         T->Right = AVLInsert(T->Right, X)
12:         if BF of node T is -2
13:             if X > T->Right->Element then
14:                 RR Rotation
15:             else
16:                 RL Rotation
17:     Update height of node T
```

```
18:     return T
19: end function
```

Function AVLDelete

```
1:  function AVLDelete(Tree T, int X)
2:      if T==NULL then return NULL
3:      else
4:          if X < T->Element then
5:              T->Left ← AVLDelete(T->Left, X)
6:          else if X > T->Element then
7:              T->Right ← AVLDelete(T->Right, X)
8:          else
9:              if T has both left and right subtrees then
10:                 Replace the current element with the minimum of right subtree
11: and delete the minimum in the subtree.
12:             else
13:                 Connect its subtree directly
14:         If T==NULL then return NULL
15:         Update height of node T
16:         if BF of node T is 2 then
17:             if X < T->Left->Element then LL Rotation
18:             else LR Rotation
19:         if BF of node T is -2
20:             if X > T->Right->Element then RR Rotation
21:             else RL Rotation
22:     return T
23: end function
```

2.5 Module Splay Tree

2.5.1 Specification:

The module can be divided into two parts including insertion and deletion of the splay Tree. In each part, we also test the time consumption. For one thing, we insert an element according to the principle of binary search tree, and then adjust the new element to be the root of the new tree through splay operation. For another, when we delete a node, we first find the position of the node to be deleted and then splay it to be the root, and remove it. If the deleted root had a left sub-tree, we find the largest element in the left sub-tree and splay it to be the new root, then link it with the right sub-tree. If there’ s no left sub tree, we directly make the root of the right sub-tree the new root of the tree. Until the tree become empty, we finish our operation.

The splay operation includes three kinds of rotations: the pattern from the node’ s grandpa to is zig-zag, and zig-zig.

For 1 when the node has no grandpa

For 2 Grandpa-Father-Son=zig-zag

For 3 Grandpa-Father-Son=zig-zag

2.5.2 Pseudo code:

Function SplayInsert

```
1:  function SplayInsert(Tree &R, int x)
2:      Create New Node T for x
3:      If the tree is empty then R ← T
4:      else
5:          P ← R
6:          while P ≠ NULL
7:              G ← P
8:              if T->Element < P->Element then
9:                  P ← P->Left
10:             else P ← P->Right
11:         make G the P’ s father
12:         Splay the T to be the root
13: end function
```

Function SplayDelete

```
1:  function SplayDelete(Tree &R, int x)
2:      if the tree is empty then return
```

```
3:      TREE P ← R
4:      while P ≠ NULL
5:          if x < P->Element then
6:              P ← P->Left
7:          if x < P->Element then
8:              P ← P->Right
9:          else break
10:         Splay P to be the root
11:         R ← P
12:         TREE TL ← R->Left
13:         TREE TR ← R->Right
14:         delete R
15:         If TL is not empty then
16:             TL->Parent ← NULL
17:             P ← TL
18:             while P->Right ≠ NULL
19:                 P ← P->Right
20:             Splay P to be the root
21:             R ← P
22:             R -> Right ← TR
23:         else If TL is not empty then
24:             TR->Parent ← NULL
25:             R ← TR
26:         else
27:             R ← NULL
27: end function
```

Function Splay

```
1: function Splay (Tree T)
2:      TREE P, G, PG
3:      while T has parent
4:          P ← T->Parent;
5:          if P has no parent then
6:              if (T = P->Left) then T ← LeftRotation(P);
```

```
7:              else T ← RightRotation(P);
8:          else
9:              G ← P->Parent;
10:             if (P = G->Left) then
11:                 if (T = P->Left) then T ← LeftZigZigRotation(G)
12:                 else T ← LeftRightZigZagRotation(G)
13:             else
14:                 if (T = P->Left) T ← RightLeftZigZagRotation(G);
15:                 else T ← RightZigZigRotation(G);
16:             PG ← T->Parent;
17:             if (PG ≠ NULL) then link PG to be T' s parent
18:     return T;
19: end function
```

2.6 Function Test Time

2.6.1 Specification:

The purpose of this function is to test the time consumption for three trees' insertion and deletion. To make it more precise, we do 100 cycles in each case and average the results.

2.6.2

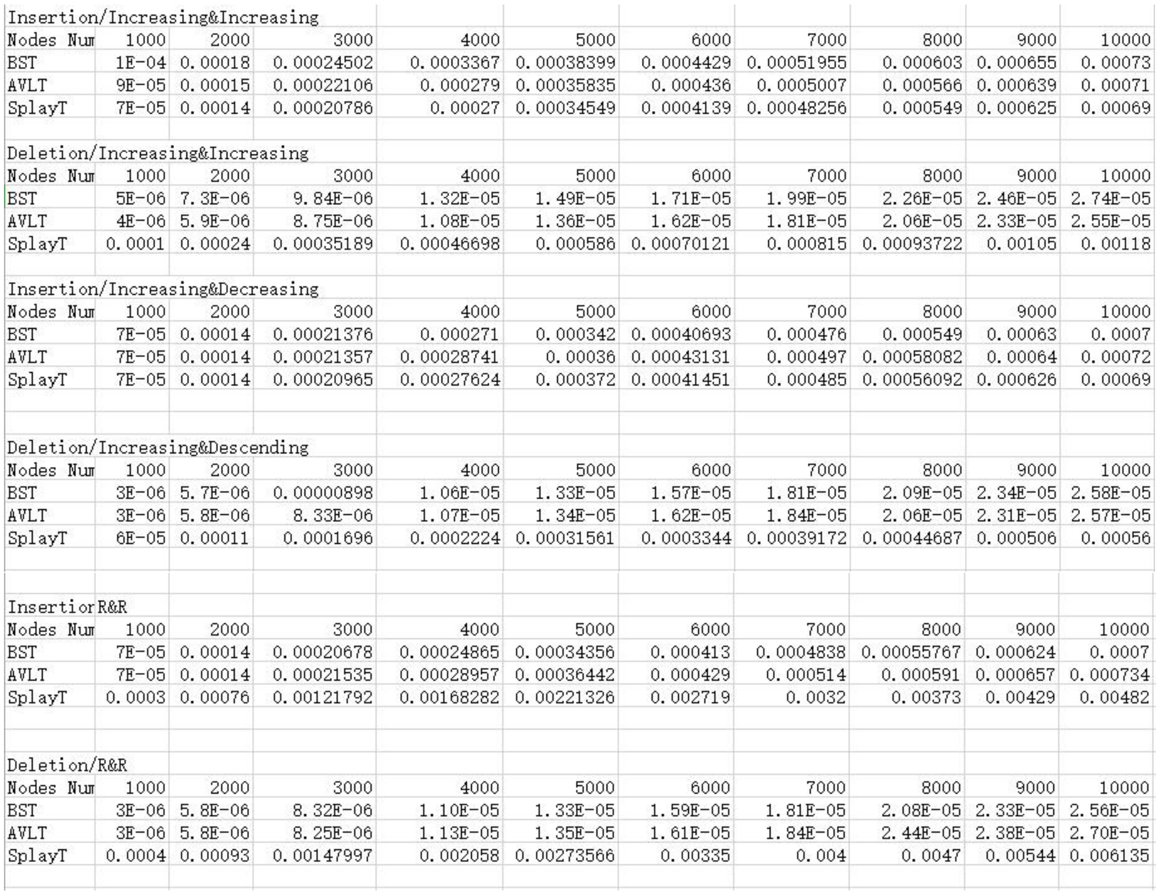
Function TestTime

```
1: function TestTime
2: TimeStart( )
3: Call TreeInsert( )
4: TimeStop( )
5: TimeStart( )
6: Call TreeDelete( )
7: TimeStop( )
8: end function
```

Chapter 3 Testing Result

3.1 Data Source

By comparing the insertion and deletion time of the one tree in three c ases , and the insertion and deletion time of three trees in the same case, t he performance of a series of insertions and deletions on the structure of t hese search trees can be analyzed and compared.



Insertion/Increasing&Increasing

Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	1E-04	0.00018	0.00024502	0.0003367	0.00038399	0.0004429	0.00051955	0.000603	0.000655	0.00073
AVLT	9E-05	0.00015	0.00022106	0.000279	0.00035835	0.000436	0.0005007	0.000566	0.000639	0.00071
SplayT	7E-05	0.00014	0.00020786	0.00027	0.00034549	0.0004139	0.00048256	0.000549	0.000625	0.00069

Deletion/Increasing&Increasing

Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	5E-06	7.3E-06	9.84E-06	1.32E-05	1.49E-05	1.71E-05	1.99E-05	2.26E-05	2.46E-05	2.74E-05
AVLT	4E-06	5.9E-06	8.75E-06	1.08E-05	1.36E-05	1.62E-05	1.81E-05	2.06E-05	2.33E-05	2.55E-05
SplayT	0.0001	0.00024	0.00035189	0.00046698	0.000586	0.00070121	0.000815	0.00093722	0.00105	0.00118

Insertion/Increasing&Decreasing

Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	7E-05	0.00014	0.00021376	0.000271	0.000342	0.00040693	0.000476	0.000549	0.00063	0.0007
AVLT	7E-05	0.00014	0.00021357	0.00028741	0.00036	0.00043131	0.000497	0.00058082	0.00064	0.00072
SplayT	7E-05	0.00014	0.00020965	0.00027624	0.000372	0.00041451	0.000485	0.00056092	0.000626	0.00069

Deletion/Increasing&Descending

Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	3E-06	5.7E-06	0.00000898	1.06E-05	1.33E-05	1.57E-05	1.81E-05	2.09E-05	2.34E-05	2.58E-05
AVLT	3E-06	5.8E-06	8.33E-06	1.07E-05	1.34E-05	1.62E-05	1.84E-05	2.06E-05	2.31E-05	2.57E-05
SplayT	6E-05	0.00011	0.0001696	0.0002224	0.00031561	0.0003344	0.00039172	0.00044687	0.000506	0.00056

Insertior R&R

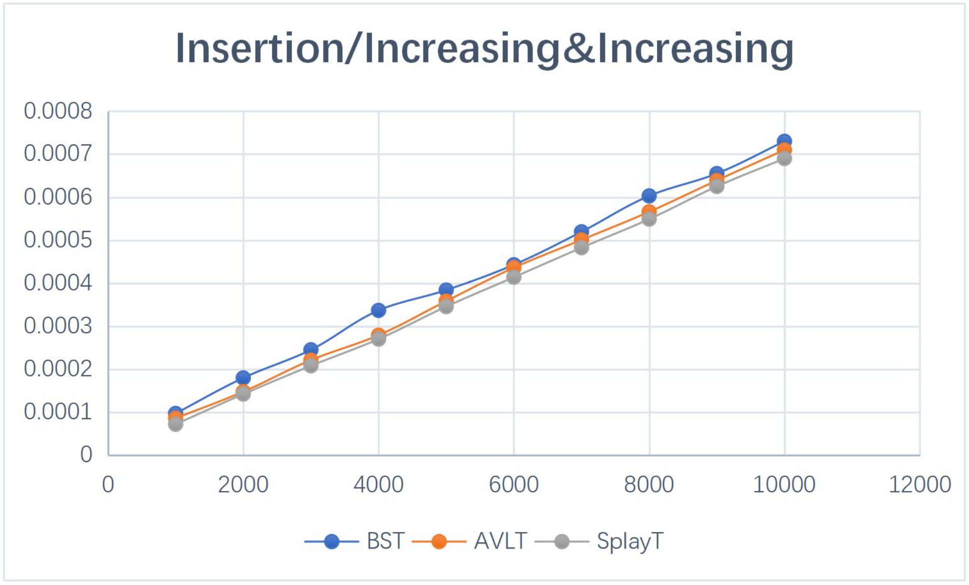
Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	7E-05	0.00014	0.00020678	0.00024865	0.00034356	0.000413	0.0004838	0.00055767	0.000624	0.0007
AVLT	7E-05	0.00014	0.00021535	0.00028957	0.00036442	0.000429	0.000514	0.000591	0.000657	0.000734
SplayT	0.0003	0.00076	0.00121792	0.00168282	0.00221326	0.002719	0.0032	0.00373	0.00429	0.00482

Deletion/R&R

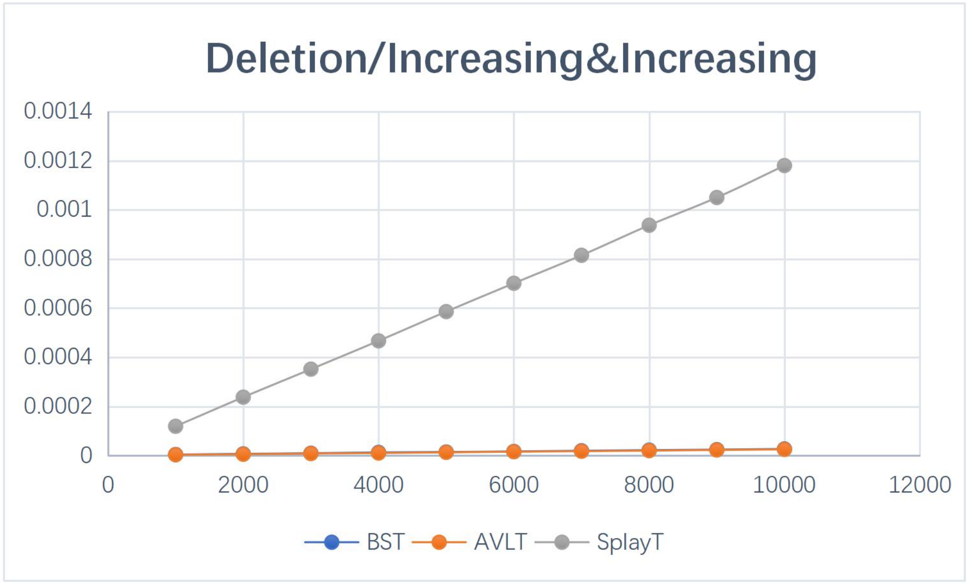
Nodes Nu	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BST	3E-06	5.8E-06	8.32E-06	1.10E-05	1.33E-05	1.59E-05	1.81E-05	2.08E-05	2.33E-05	2.56E-05
AVLT	3E-06	5.8E-06	8.25E-06	1.13E-05	1.35E-05	1.61E-05	1.84E-05	2.44E-05	2.38E-05	2.70E-05
SplayT	0.0004	0.00093	0.00147997	0.002058	0.00273566	0.00335	0.004	0.0047	0.00544	0.006135

3.2 Time Consumption for three trees in the same condition

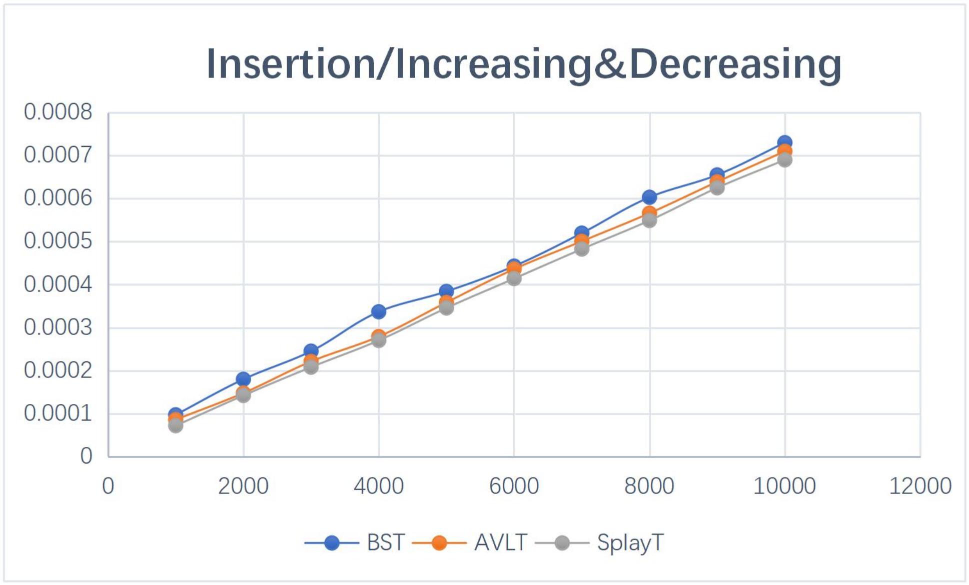
3.2.1Insertion/Increasing&Increasing



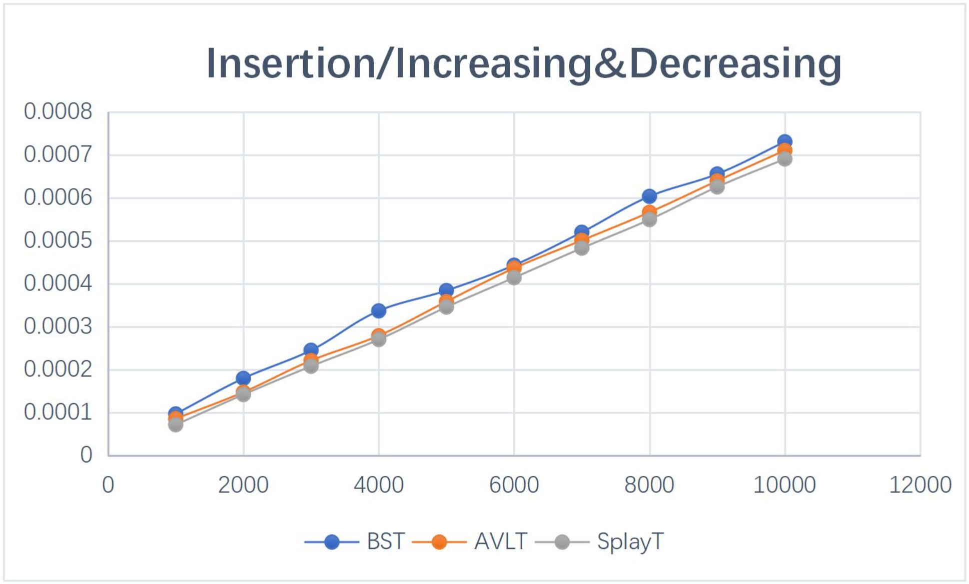
3.2.2 Deletion/Increasing&Increasing



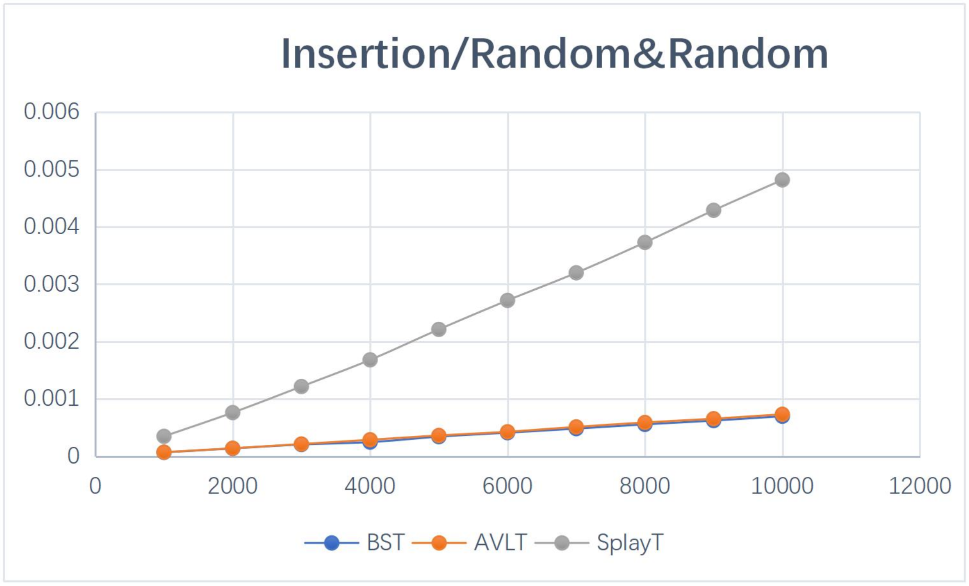
3.2.3 Insertion/Increasing&Increasing



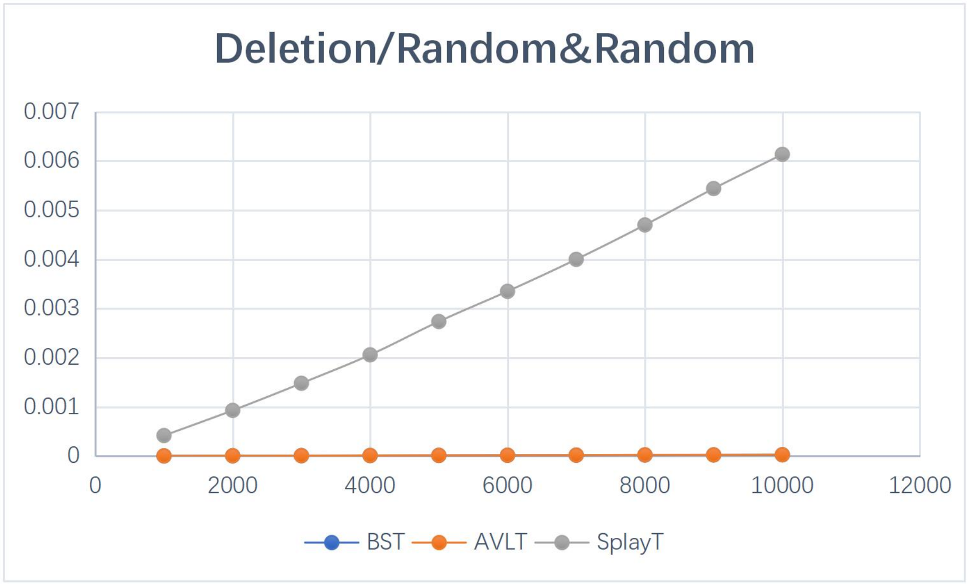
3.2.4 Deletion/Increasing&Decreasing



3.2.5 Insertion/Random&Random

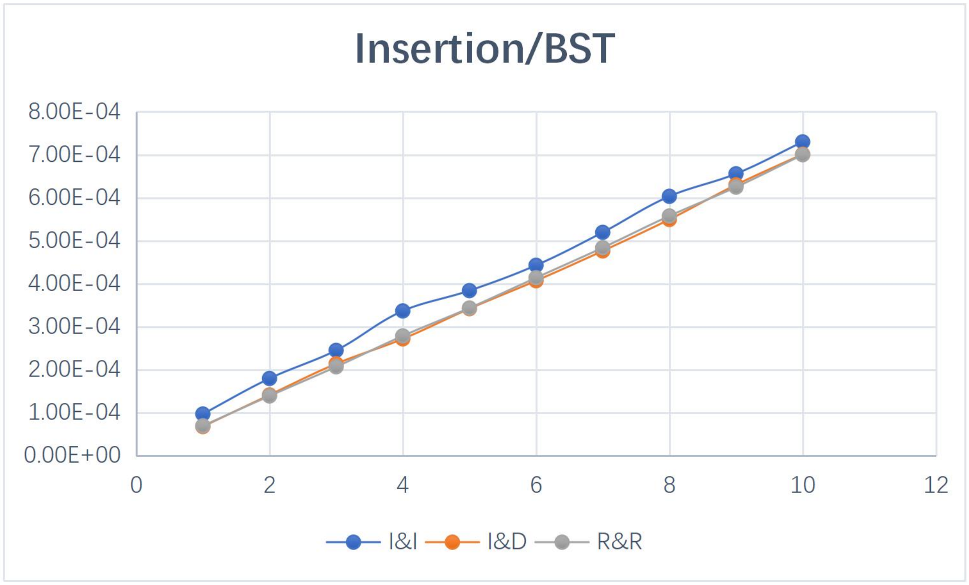


3.2.6 Deletion/Random&Random

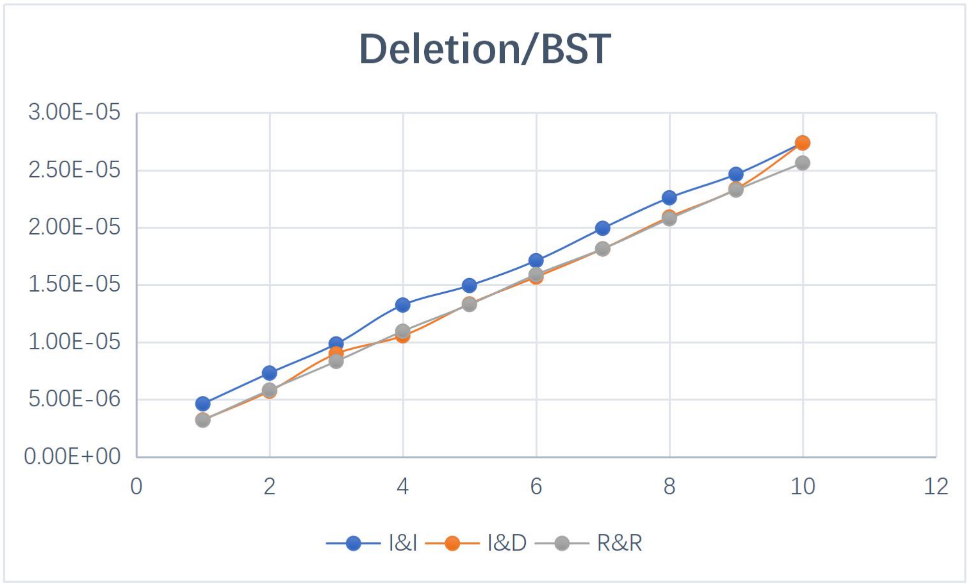


3.3 Time Consumption for one trees in the three conditions

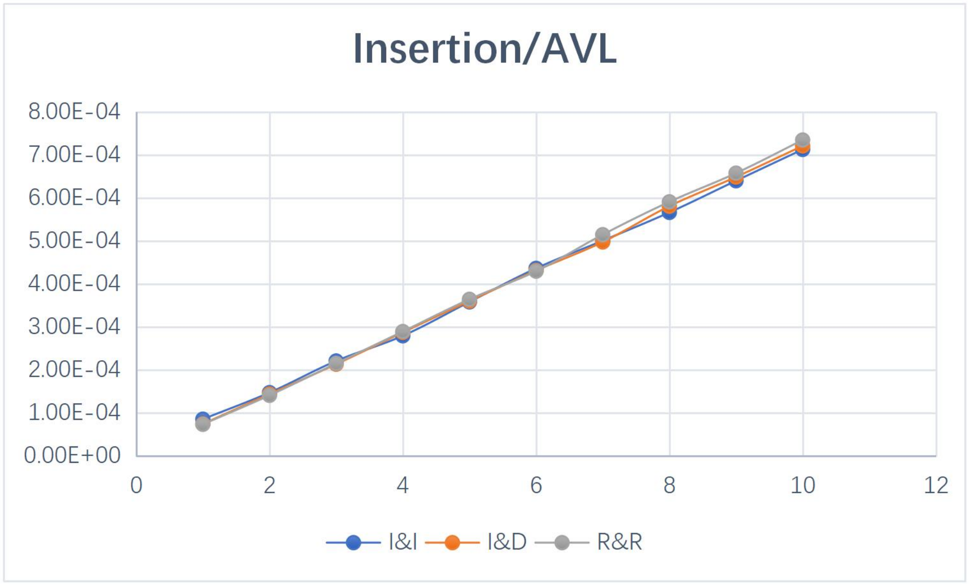
3.3.1 Insertion of BST



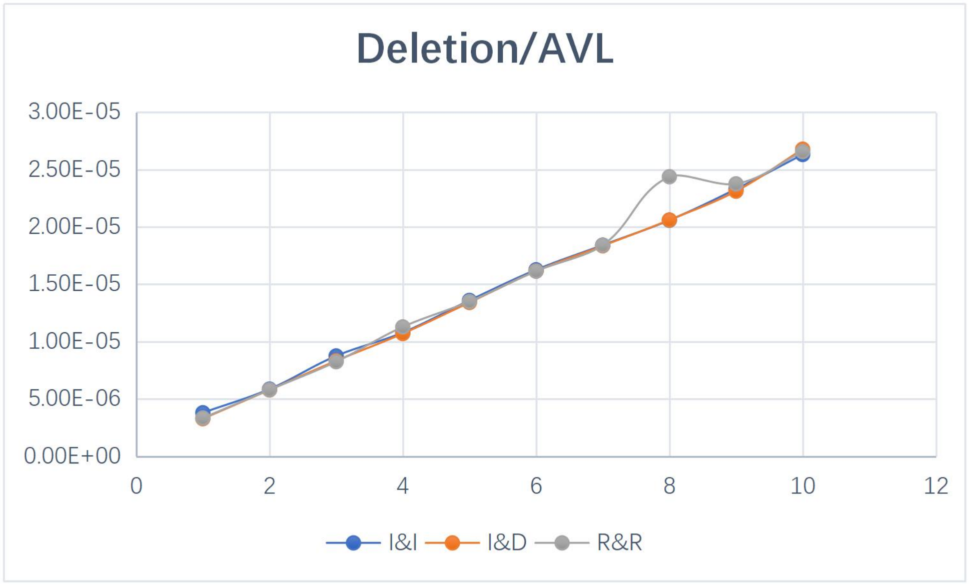
3.3.2 Deletion of BST



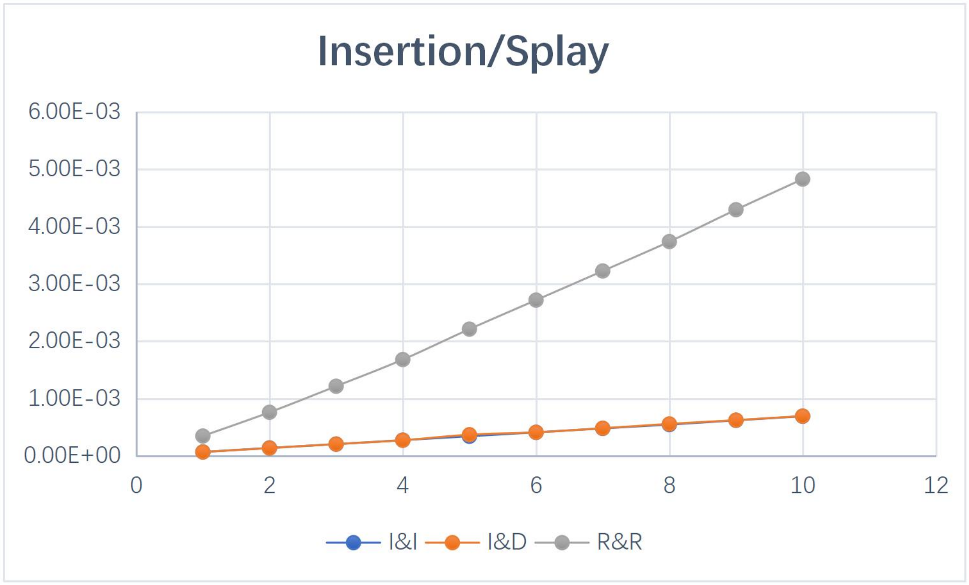
3.3.3 Insertion/AVL



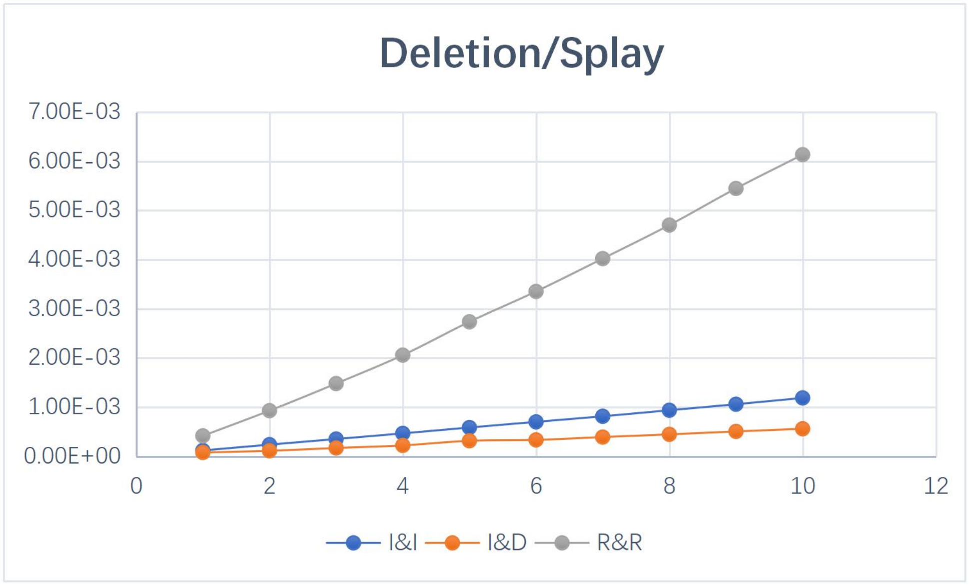
3.3.4 Deletion/AVL



3.3.5 Insertion/Splay



3.3.6 Deletion/Splay



Chapter 4 Analysis and Comments

4.2 Analysis and Comments of the Testing Data

- For insertion and deletion in splay tree:

We can see that in the random case it takes much more time than I/I or I/D. The reason may be that in random cases the tree may be quite unbalanced and the random position may be quite different and every insertion takes much time for splay operation. Plus, in random case, the advantage of splay operation in access for same or similar elements frequently didn't work, so it may be quiet time consuming.

We can see that for time consumption in deletion, I/D<I/I<R/R. It's reasonable because for random access the splay operation has not got much advantage but take more time. I/D is an ideal case for splay as each element to delete is already at the root. And I/I is a bit more time consuming, but each splay operation may reduce the height of the element near the deleted element, so it is still much better than random.

- For insertion and deletion in AVL:

It's quite similar and stable in three cases. As AVL is always keep balanced, it's reasonable.

- For insertion and deletion in random case, splay takes much more time than AVL and BST. It's reasonable because BST in random case won't be too unbalance and AVL is a balance tree, but splay may do too much unnecessary splay operation.

- For deletion in I/I case, splay takes much more times it may do too many rotations to bring back the far elements to the root and again find the largest and splay the largest. And it's still not balanced.

- For insertion and deletion in I/D case, BST takes a bit more time than the other two.

4.2 Pre-analysis about Time Complexity

- Simple Binary Search Tree

In the worst case, BST will degenerate into a skew binary tree, which meas all nodes have only left subtree or right subtree. In this case, the time complexity of insertion or deletion a new node is O (N). In the best case, the height of a BST is O (logN), so the time complexity of insertion and deletion operations is O (logN).

- AVL Tree

Suppose the height of root is h, the height of its left subtree is h-1, and the height of its right subtree is h-2. Assume that T (h) is the minimum nu mber of nodes that AVLTree can put at the height of H. According to T (h) = T (h-1) + T (h-2) + 1, we can get the height of the tree h < 2 * logT (h). S o the height of the tree is O (logN). Thus, the insertion and deletion operat ions of AVL tree are O(logN).

- Splay Tree

For a Splay tree, by using potential function, we can get the upper bound of amortization time complexity O (log2 (| T |)) for splay operations also shows that the upper bound of amortization time complexity for all splay operations is O (log2 (| T |)

4.3 Pre-analysis about Space Complexity

Establish a AVLTree with N nodes and the space complexity is O (N). For insertion and deletion operations, a recursive method is used. The recursive depth is O (logN). Temporary O (1) space complexity is used for each recursive normalization, so the space complexity of insertion and deletion operations is O (logN). Apart from the worst situation for BST' s recursive depth is O (N), BST is same as AVLTree. Splay' s space complexity is also same as AVLTree. In a word, all of these three trees space complexity are O(N).

Appendix

Appendix A Source code

```cpp
#include<iostream>
#include<iomanip>
#include<algorithm>
#include<cstdio>
#include<cstdlib>
#include<ctime>

using namespace std;

/* the basic structure of the three tree*/
typedef struct TreeNode *Tree;
struct TreeNode {
    int Element,Height;
    Tree Left,Right,Parent;
};

int n; // the size of the data
int *a,*b,*RandomArray1,*RandomArray2; // the arrays to store sequence of input data

/* Splay */

/* T's father P is the root and T is P's left child */
Tree LeftRotation(Tree T) {
    Tree P = T->Left;
    P->Parent = T->Parent;
    T->Left = P->Right;
    if (P->Right!=NULL)
        P->Right->Parent = T;
    P->Right = T;
    T->Parent =P;
    T = P;
    return T;
}

/* T's father P is the root and T is P's right child */
Tree RightRotation(Tree T) {
```

```c
    Tree P = T->Right;
    P->Parent = T->Parent;
    T->Right = P->Left;
    if (P->Left!=NULL)
        P->Left->Parent = T;
    P->Left = T;
    T->Parent = P;
    T = P;
    return T;
}

/* T has a grandpa and the LR zig-zag case */
Tree LeftRightZigZagRotation(Tree T) {
    T->Left = RightRotation(T->Left);
    T = LeftRotation(T);
    return T;
}

/* T has a grandpa and the RL zig-zag case */
Tree RightLeftZigZagRotation(Tree T) {
    T->Right = LeftRotation(T->Right);
    T = RightRotation(T);
    return T;
}

/* T has a grandpa and the LL zig-zig case */
Tree LeftZigZigRotation(Tree T) {
    T = LeftRotation(T);
    T = LeftRotation(T);
    return T;
}

/* T has a grandpa and the RR zig-zig case */
Tree RightZigZigRotation(Tree T) {
    T = RightRotation(T);
    T = RightRotation(T);
    return T;
}

/*
 * Description: This function is used to implement splay operation.
 * Parameter[in] T  the element to splay up to the root
 * return T after splay as the new root of the tree
 */
Tree Splay(Tree T) {
    /*
    * P father of T
    * G grandfather of T if exist
    * PG great-grandfather of T is exist
    */
   Tree P,G,PG;
   while (T->Parent!=NULL) { // until T is the root
       P = T->Parent;
       if (P->Parent == NULL) { // if T has no grandfather
           if (T == P->Left) T = LeftRotation(P);
           else T = RightRotation(P);
       }
       else {
           G = P->Parent;
           if (P == G->Left) {
               if (T == P->Left) T = LeftZigZigRotation(G); // LL zig-zig case
               else T = LeftRightZigZagRotation(G); // LR zig-zag case
           }
            else {
                 if (T == P->Left) T = RightLeftZigZagRotation(G); // RL zig-zag case
                else T = RightZigZigRotation(G); // RR zig-zig case
            }
            /* link the new T to the initial tree */
            PG = T->Parent;
            if (PG!=NULL) {
                if (PG->Left == G) PG->Left = T;
                else PG->Right = T;
            }
        }
    }
    return T;
}

/*
 * Description: This function is used to implement insertion.
 * Parameter[in] T  the root of the the tree
 * Parameter[in] x  the element to be inserted
 */
void SplayInsert(Tree &R,int x) {
    Tree T,P,G;

    /* build a new node*/
    T = new TreeNode;
    T->Element = x;
    T->Parent = T->Left = T->Right = NULL;

    /* if the tree is empty */
    if (R==NULL) R = T;
    else {
        /* find the place for the new node*/
        P = R;
        while (P != NULL) {
            G = P;
            if (T->Element < P->Element)
                P = P->Left;
            else P = P->Right;
        }
        /* insert the new node */
        if (T->Element < G->Element)
            G->Left = T;
        else if (T->Element > G->Element)
            G->Right = T;
        T->Parent = G;
        /* splay the new node to root */
        R = Splay(T);
    }
}

/*
 * Description: This function is used to implement deletion.
 * Parameter[in] T  the root of the tree
 * Parameter[in] X  the element to be deleted
 */
void SplayDelete(Tree &R,int x) {
    Tree P,TL,TR;
    /* if there is no node left in the tree*/
    if (R == NULL) return;

    /* otherwise find the node to delete */
    P = R;
    while (P != NULL) {
        if (x < P->Element)
            P = P->Left;
        else if (x > P->Element)
            P = P->Right;
        else break;
    }
    /* splay the node to the root*/
    if (P != R)
        R = Splay(P);

    /* record the right and left tree of the root */
    TL = R->Left;
    TR = R->Right;
    /* romove the root */
    delete R;

    /* if there is a left tree then adjust the Left Tree */
    if (TL != NULL) {
        /* break the link between the left child and the deleted root */
        TL->Parent = NULL;
        /* find the largest element in the left tree */
        P = TL;
        while (P->Right != NULL)
            P=P->Right;
        /* splay the maximum to the root */
        if (P!=TL)
            R = Splay(P);
        else R = P;
        /* link the new root and the right tree */
        R->Right = TR;
        if (TR != NULL)
            TR->Parent = R;
    }
    /* if the root has no left tree the make its right son the new root */
    else if (TR != NULL) {
        TR->Parent = NULL;
        R = TR;
    }
    /* if the root has no child then the tree will be empty */
    else R = NULL;
}

/* AVL */

/*
 * Description: This function is used to find the larger one between two numbe
rs
 */
int Max(int a, int b) {
    if (a > b)return a;
    else return b;
}

/*
 * Description: This function is used to get the Height of one node.
 */
int GetHeight(Tree T) {
    if (!T)
```

```c
        return -1;/* Define the height of empty nodes as -1 */
    else
        return T->Height;
}

/*
 * Description: This function is used to implement the single left rotation.
 */
Tree AVL_LL_Rotation(Tree A) {
    Tree B = A->Left;
    A->Left = B->Right;
    B->Right = A;
     A->Height = Max(GetHeight(A->Left), GetHeight(A->Right)) + 1;//Update the height of the affected nodes
    B->Height = Max(GetHeight(B->Left), A->Height) + 1;
    return B;
}

/*
 * Description: This function is used to implement the single right rotation.
 */
Tree AVL_RR_Rotation(Tree A) {
    Tree B = A->Right;
    A->Right = B->Left;
    B->Left = A;
     A->Height = Max(GetHeight(A->Left), GetHeight(A->Right)) + 1;//Update the height of the affected nodes
    B->Height = Max(GetHeight(B->Right), A->Height) + 1;
    return B;
}

/*
 * Description: This function is used to implement the double single right rotation.
 */
Tree AVL_LR_Rotation(Tree A) {
    Tree B = A->Left;
    Tree C = B->Right;
    B->Right = C->Left;
    C->Left = B;
    A->Left = C->Right;
    C->Right = A;
     A->Height = Max(GetHeight(A->Left), GetHeight(A->Right)) + 1;//Update the height of the affected nodes
    B->Height = Max(GetHeight(B->Left), GetHeight(B->Right)) + 1;
    C->Height = Max(GetHeight(C->Left), GetHeight(C->Right)) + 1;
    return C;
```

```c
}

/*
 * Description: This function is used to implement the double right single rotation.
 */
Tree AVL_RL_Rotation(Tree A) {
    Tree B = A->Right;
    Tree C = B->Left;
    B->Left = C->Right;
    C->Right = B;
    A->Right = C->Left;
    C->Left = A;
     A->Height = Max(GetHeight(A->Left), GetHeight(A->Right)) + 1;//Update the height of the affected nodes
    B->Height = Max(GetHeight(B->Left), GetHeight(B->Right)) + 1;
    C->Height = Max(GetHeight(C->Left), GetHeight(C->Right)) + 1;
    return C;
}

/*
 * Description: This function is used to implement insertion.
 * Parameter[in] T  the root of the sub-tree to insert the new element
 * Parameter[in] X  the element to be inserted
 * return T  the new sub-tree after insertion
 */
Tree AVLInsert(Tree T, int X) {
    if (!T) {// If X have reach the right position, Create a new node.
        T = (Tree)malloc(sizeof(struct TreeNode));
        T->Element = X;
        T->Height = 0;//Set the height as 0, for it's a leaf node.
        T->Left = T->Right = NULL;
    }
    else if (X < T->Element) {
        T->Left = AVLInsert(T->Left, X);
         if (GetHeight(T->Left) - GetHeight(T->Right) == 2) {// If X have been inserted, check the balance factor of current node
            if (X < T->Left->Element)// Judge LL or LR
                T = AVL_LL_Rotation(T);
            else
                T = AVL_LR_Rotation(T);
        }
    }
    else if (X > T->Element) {
        T->Right = AVLInsert(T->Right, X);
         if (GetHeight(T->Left) - GetHeight(T->Right) == -2) {// If X have been inserted, check the balance factor of current node
```

```
            if (X > T->Right->Element)// Judge RR or RL
                T =AVL_RR_Rotation(T);
            else
                T = AVL_RL_Rotation(T);
        }
    }
    T->Height = Max(GetHeight(T->Left), GetHeight(T->Right)) + 1;//Update the height of current node
    return T;
}

/*
 * Description: This function is used to find the minimum number of current node's right subtree.
 */
Tree FindMin(Tree T) {
    if (!T)return NULL;
    else if (!T->Left)return T;
    else return FindMin(T->Left);
}

/*
 * Description: This function is used to implement deletion.
 * Parameter[in] T  the root of the sub-tree to delete the new element
 * Parameter[in] X  the element to be deleted
 * return T  the new sub-tree after deletion
 */
Tree AVLDelete(Tree T, int X) {
    Tree Temp;
    if (!T)return NULL;
    else {
        if (X < T->Element)// If X is less than current element, turn to the left subtree.
            T->Left = AVLDelete(T->Left, X);
        else if (X > T->Element)// If X is greater than current element, turn to the right subtree.
            T->Right = AVLDelete(T->Right, X);
        else {
            if (T->Left&&T->Right) {// If the element deleted have both left and right subtrees, adjust the structure.
                 Temp = FindMin(T->Right);// Find the minimum node of the right subtree
                T->Element = Temp->Element;// Replace the current node.
                T->Right = AVLDelete(T->Right, T->Element);
            }
            else {//If the element deleted have one left or right subtrees, directly connect left or right subtree.
                Temp = T;
                if (!T->Left)T = T->Right;
                else T = T->Left;
                free(Temp);
            }
        }
        if (!T)return NULL;// If one leaf node is deleted, return NULL.

        T->Height = Max(GetHeight(T->Left), GetHeight(T->Right)) + 1; // Update the height of the affected nodes

        if (GetHeight(T->Left) - GetHeight(T->Right) == 2) { // If X have been inserted, check the balance factor of current node
            if (X < T->Left->Element)// Judge LL or LR
                T = AVL_LL_Rotation(T);
            else
                T = AVL_LR_Rotation(T);
        }
        if (GetHeight(T->Left) - GetHeight(T->Right) == -2) { // If X have been inserted, check the balance factor of current node
            if (X > T->Right->Element)// Judge RR or RL
                T = AVL_RR_Rotation(T);
            else
                T = AVL_RL_Rotation(T);
        }
    }
    return T;
}

/* BST */

/*
 * Description: This function is used to implement insertion.
 * Parameter[in] T  the root of the sub-tree to insert the new element
 * Parameter[in] X  the element to be inserted
 * return T  the new sub-tree after insertion
 */
Tree BSTInsert(Tree T, int X) {
    if (!T) { // If X have reach the right position, Create a new node.
        T = new TreeNode;
        T->Element = X;
        T->Height = 0; //Set the height as 0, for it's a leaf node.
        T->Left = T->Right = NULL;
    }
    else {
        if (X < T->Element) // If X is less than current element, turn to the left subtree.
            T->Left = BSTInsert(T->Left, X);
```

```cpp
        else if (X > T->Element) // If X is greater than current element, turn to the right subtree.
            T->Right = BSTInsert(T->Right, X);
    }
    return T;
}

/*
 * Description: This function is used to implement deletion.
 */
Tree BSTDelete(Tree T, int X) {
    Tree Temp;
    if (!T)return NULL;
    else {
        if (X < T->Element) // If X is less than current element, turn to the left subtree.
            T->Left = BSTDelete(T->Left, X);
        else if (X > T->Element) // If X is greater than current element, turn to the right subtree.
            T->Right = BSTDelete(T->Right, X);
        else {
            if (T->Left&&T->Right) { // If the element deleted have both left and right subtrees, adjust the structure.
                Temp = FindMin(T->Right); // Find the minimum node of the right subtree
                T->Element = Temp->Element; //Replace the current element
                T->Right = BSTDelete(T->Right, T->Element);
            }
            else { //If the element deleted have one left or right subtrees, directly connect left or right subtree.
                Temp = T;
                if (!T->Left)T = T->Right;
                else T = T->Left;
                delete Temp;
            }
        }
    }
    return T;
}


/* create input */

void CreateArray(){
        int i,k,temp;
        srand((unsigned)time(NULL));
        /* randomly create an increasing squence*/
        RandomArray1 = new int[n+1];
        RandomArray2 = new int[n+1];
        a = new int[n+1];  //increasing
        b = new int[n+1];  //reverse

        a[0]=2;
        for (i=1;i<n;i++)   //generate the random increasing array
            a[i]=a[i-1]+rand()%8+1;

        for (i=0;i<n;i++){
            b[i]=a[n-i-1];
        }

        /* shuffle the random array to generate a constraint */
        for (i=0;i<n;i++)
            RandomArray1[i]=a[i];
        for (i=n-1;i>=1;i--) {   //Disrupt the order for insert
            k=rand()%i;
            temp=RandomArray1[i];
            RandomArray1[i]=RandomArray1[k];
            RandomArray1[k]=temp;
        }
        for (i=0;i<n;i++)
            RandomArray2[i]=a[i];
        for (i=n-1;i>=1;i--) {   //Disrupt the order for insert
            k=rand()%i;
            temp=RandomArray2[i];
            RandomArray2[i]=RandomArray2[k];
            RandomArray2[k]=temp;
        }
}

/* time-testing part*/

/* test part for splay */
void SplayTree(int *Array1,int *Array2){   // Array1 and Array2 is the insertion array and deletion array
    cout<<"SplayTree:"<<endl;
    int i;
    Tree SplayT = NULL;
    clock_t startins,endins,startdel,enddel;   // timing part
    double durationins=0,durationdel=0;

    for (int ite=0;ite<100;ite++) {
        startins=clock();  //start timing for inserting
        for (i=0;i<n;i++) {
            SplayInsert(SplayT,Array1[i]);  //insert the tree node
        }
        endins=clock();  //end timing for inserting
        durationins+=double(endins-startins)/CLOCKS_PER_SEC;

        startdel=clock();  //start timing for deleting
        for (i=0;i<n;i++) {
            SplayDelete(SplayT,Array2[i]);  //delete the tree node
        }
        enddel=clock();  //end timing for deleting
         durationdel+=double(enddel-startdel)/CLOCKS_PER_SEC;  //get the time interval in seconds
    }
    cout<<"InsertTime: "<<setprecision(10)<<durationins/100<<"s"<<endl;
    cout<<"DeleteTime: "<<setprecision(10)<<durationdel/100<<"s"<<endl;
}


/* test part for splay */
void AVLTree(int *Array1,int *Array2){  // Array1 and Array2 is the insertion array and deletion array
    cout<<"AVLTree:"<<endl;
    Tree AVLT=NULL;
    clock_t startins,endins,startdel,enddel; // timing part
    double durationins=0,durationdel=0;

    for (int ite=0;ite<100;ite++) {
        startins=clock();  //start timing for inserting
        for(int i=0;i<n;i++){
            AVLInsert(AVLT,Array1[i]);  //insert the tree node
        }
        endins=clock();  //end timing for inserting
        durationins+=double(endins-startins)/CLOCKS_PER_SEC;

        startdel=clock();  //start timing for deleting
        for(int i=0;i<n;i++){
            AVLDelete(AVLT,Array2[i]);
        }
        enddel=clock();  //end timing for deleting
        durationdel+=double(enddel-startdel)/CLOCKS_PER_SEC;
    }
    cout<<"InsertTime: "<<setprecision(10)<<durationins/100<<"s"<<endl;
    cout<<"DeleteTime: "<<setprecision(10)<<durationdel/100<<"s"<<endl;
}


/* test part for BST */
void BSTree(int *Array1,int *Array2){  // Array1 and Array2 is the insertion array and deletion array
```

```
    cout<<"BSTree:"<<endl;
    Tree BST=NULL;
    clock_t startins,endins,startdel,enddel; // timing part
    double durationins=0,durationdel=0;

    for (int ite=0;ite<100;ite++) {
        startins=clock();  //start timing for inserting
        for(int i=0;i<n;i++){
            BSTInsert(BST,Array1[i]);
        }
        endins=clock();  //end timing for inserting
        durationins+=double(endins-startins)/CLOCKS_PER_SEC;

        startdel=clock();  //start timing for deleting
        for(int i=0;i<n;i++){
            BSTDelete(BST,Array2[i]);
        }
        enddel=clock();  //end timing for deleting
        durationdel+=double(enddel-startdel)/CLOCKS_PER_SEC;
    }
    cout<<"InsertTime: "<<setprecision(10)<<durationins/100<<"s"<<endl;
    cout<<"DeleteTime: "<<setprecision(10)<<durationdel/100<<"s"<<endl;
}

/* time test */
void TestTime(){
        cout<<"increasing insertion & increasing deletion"<<endl;  // get the time consumption in the situation with increasing order insertion and increasing order deletion
        BSTree(a,a);
        AVLTree(a,a);
        SplayTree(a,a);
        cout<<endl;

        cout<<"increasing insertion & decreasing deletion"<<endl;  // get the time consumption in the situation with increasing order insertion and reversing order deletion
        BSTree(a,b);
        AVLTree(a,b);
        SplayTree(a,b);
        cout<<endl;

        cout<<"random insertion & random deletion"<<endl;  // get the time consumption in the situation with random order insertion and random order deletion

        BSTree(RandomArray1,RandomArray2);
        AVLTree(RandomArray1,RandomArray2);
        SplayTree(RandomArray1,RandomArray2);
```

```cpp
        cout<<endl;
}

int main(){
    int flag;
    cout<<"Please input the size of the tree:"<<endl;
    cin>>n;  //input the nodes of the tree
    CreateArray();  //create three kinds of arrays
    TestTime();  //timing function
    return 0;
}
```

Appendix B Declaration

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

Appendix B Signature

[illegible] 蔡泽伟 吴金泽

Duty Assignments

Splay Tree: Yizhou Chen

AVL Tree: Zewei Cai

Simple BST: Jinze Wu