Binary Search Trees

Author Names: ?

Date: 2017-3-6

Binary Search Trees

Group ?

1 Introduction

1.1 Concepts Description

1.1.1 Binary Search Tree

Binary search tree is a data structure, which meets the following requirements:

- it is a binary tree;
- each node contains a value;
- a total order is defined on these values (every two values can be compared with each other);
- left subtree of a node contains only values lesser, than the node's value;
- right subtree of a node contains only values greater, than the node's value.

Notice, that definition above doesn't allow duplicates. Storing data in binary search tree allows to look up for the record by key faster, than if it was stored in unordered list. Also, BST can be utilized to construct set data structure, which allows to store an unordered collection of unique values and make operations with such collections. Performance of a binary search tree depends of its height. In order to keep tree balanced and minimize its height, the idea of binary search trees was advanced in balanced search trees (AVL trees, Red-Black trees, Splay trees). Here we will discuss the basic ideas, laying in the foundation of binary search trees.

Actually, we should be very familiar with this data structure, so no more further discussion of BST will be implemented.

1.1.2 AVL Tree

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(Logn)$ after every insertion and deletion, then we can guarantee an upper bound of $O(Logn)$ for all these operations. The height of an AVL tree is always $O(Logn)$ where n is the number of nodes in the tree. As for specific Implementation, it will be discussed in Chapter 2.

1.1.3 Splay Tree

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. All normal operations on a binary search tree are combined with one basic operation, called splaying. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Advantages:

1. Comparable performance: Average-case performance is as efficient as other trees.
2. Small memory footprint: Splay trees do not need to store any bookkeeping data.

Disadvantages: The most significant disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all n elements in non-decreasing order.

1.2 Project Description

This project is not so difficult. We need to simply implement operations on binary search trees, AVL trees and splay tree. In order to study and compare these data structures, we should compare the performances by inserting and deleting a sequence of numbers in different ways as the project itself gave us.

2 Algorithm Specification

2.1 Implement of AVL Tree

2.1.1 Data Structure

The data structure AVL tree uses is like below:

```
struct _avlnode;
typedef struct _avlnode AvlNode;
typedef AvlNode* AvlPointer;
```

The specific description is like below:

```
struct _avlnode
{
    int value;
    AvlPointer left;
    AvlPointer right;
    int height;
};
typedef struct _avltree
{
    AvlPointer root;
} AvlTreeNode;
typedef AvlTreeNode* AvlTree;
```

2.1.2 Algorithm and Pseudo Code

1. Firstly, we need to implement the functions of building an empty tree and freeing an AVL tree, which is rather easy and it clearly needs no more discussion. You can find it in the appendix.

2. Then we implement the function of finding a certain value. All we have to do is use a while loop to locate the exact node of certain value.

```
AvlPointer find(int value, AvlTree tree)
{
    AvlPointer p ← tree->root;
    while(p != NULL and p->value != value){
        if (p->value < value) p ← p->right;
        else p ← p->left;
    }
    return p;
```

3. In order to delete the nodes in a increasing or decreasing order, we need to implement the functions of finding the min and max node of value. Both requires a single while loop to search the left(min) sub-tree or right(max) sub-tree.

4. The deletion of certain nodes. This function finds it and deletes it and returns the root of the tree after deletion. If the deletion is successful, return 1 to the caller, otherwise return 0. We only need to focus on what happens when the node has children, so the rest of the code was left out. First, we should delete the node recursively.

```
if(p has left child)
{
    AvlPointer lcmax ← AVLFindMaxNode(p->left);
    p->value ← lcmax->value;
    p->left ← AVLNodeDel(lcmax->value, p->left);
}
else
{
    AvlPointer rchild ← p->right;
    free(p);
    p ← rchild;
}
```

Then, we adjust the height of nodes.

```
if(p exists)
{
    if(p has left child)lh ← p->left->height + 1;   else lh = 0;
    if(p has right child)rh ← p->right->height + 1; else rh = 0;
    p->height ← max(lh, rh);

    if(p is not balanced and left is higher)
    {
        adjust llh and lrh
        if(llh > lrh)p ← llrot(p);
        else p ← lrrot(p);
    }
    else if(p is not balanced and right is higher)
    {
        adjust rlh and rrh
        if(rlh > rrh)p ← rlrot(p);
        else p ← rrrot(p);
    }
}
```

5. The insertion of certain nodes of value. It also returns the root of this sub-tree after insertion. We only focus on how we adjust the height and do the rotation function.

```
// assign the height of p
if(p has left chile)lh ← p->left->height + 1;
if(p has right child)rh ← p->right->height + 1;
p->height ← max(lh, rh);
// the following code describes how to rotate in different cases
if(p is not balanced and right is higher){
    if(value > p->right->value)
        p ← rrrot(p);
    else
        p ← rlrot(p);
    }else if(p is not balanced and left is higher){
    if(value < p->left->value)
        p ← llrot(p);
    else
        p ← lrrot(p);
}
```

6. Then it comes to the 4 rotation functions of AVL tree, which is very similar to each other. So we just implement single left rotation and double left right rotation in pseudo code. The rest functions are the same. For example, RR rotation is as showed in the following figure.

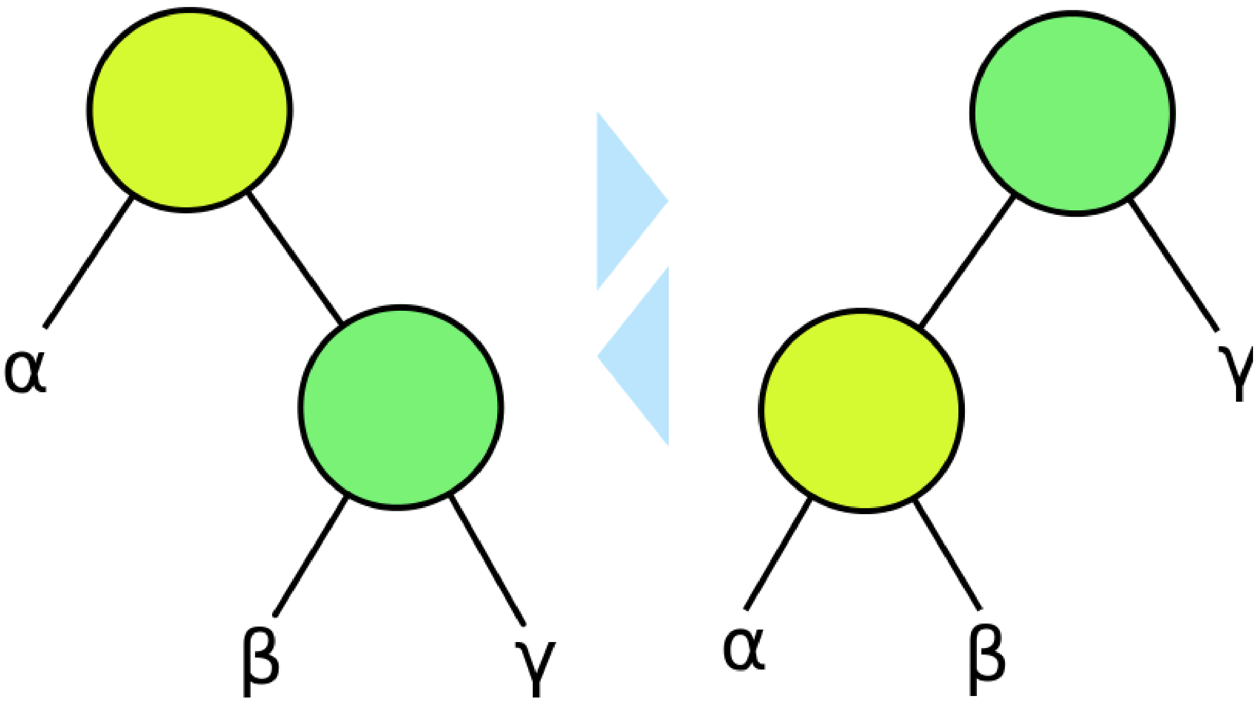
```
// single left rotation
AvlPointer llrot(AvlPointer A){
    AvlPointer B ← A->left;

    A->left ← B->right;
    B->right ← A;
    A->height ← A->height - 2;

    return B;
}

// double left right rotation
AvlPointer lrrot(AvlPointer A){
    AvlPointer B ← A->left;
    AvlPointer C ← B->right;

```




```
    A->left ← C->right;
    C->right ← A;
    B->right ← C->left;
    C->left ← B;
    A->height ← A->height - 2;
    B->height ← B->height - 1;
    C->height ← C->height + 1;
    return C;
}
```

- Second solution: To simplify the length of code, we can actually implement the double rotation by using the single rotation function twice. At the same time, we should calculate the height of the current node by calculating its left and right sub-trees and plus 1 rather than directly plus and minus. Like:

```
AvlPointer lrrot(AvlPointer A) {
    A->left = rrrot(A->left);
    return lrrot(A);
}
```

Of course we need to adjust the code of single rotation, we should calculate the height of the current node by

```
A->height = max(A->left->height, A->right->height) + 1;
```

2.1.3 C

You can find C code in the appendix

2.1.4 Proof of correctness

After thorough testing, our results are acceptable. Actually our rotation functions make sure that every adjustment of height can change the BF into less than 1. So we can tell that they are correct.

2.2 Implement of Splay Tree

2.2.1 Data Structure

Instead of the member variable height in AVL tree, we used the pointer parent as a member variable to implement splay tree. The rest part of data structure is very similar to the AVL tree.

```
typedef struct _splaynode SplayNode;
typedef SplayNode* SplayPointer;
struct _splaynode{
    int value;
    SplayPointer left;
    SplayPointer right;
    SplayPointer parent;
};

typedef struct _splaytree
{
    SplayPointer root;
} SplayTreeNode;
typedef SplayTreeNode* SplayTree;
```

2.2.2 Pseudo Code

1. Firstly, we need to implement the functions of building an empty tree and freeing a splay tree, which is rather similar to AVL tree and it clearly needs no more discussion.

2. Then we implement the function of finding a certain value. If found and it isn't the root, splay the node, otherwise simply return the pointer. It's like normal BST tree's function of finding. So no more extra description.

3. In order to delete the nodes in a increasing or decreasing order, we need to implement the functions of finding the min and max node of value. Both requires a single while loop to search the left(min) sub-tree or right(max) sub-tree. They are very much like AVL tree. To simplify the report, we left out the specific implement and you can find them in the appendix.

4. The deletion of certain nodes. This function finds it and deletes it and returns the root of the tree after deletion. If the deletion is successful, return 1 to the caller, otherwise return 0. Like AVL tree, we only focus on the key part of adjusting.

```
if(p has both children)
{
    new_root ← splay_find_max(p->left);
    tree->root ← Splay(new_root, p->left);
    new_root->parent ← NULL;
    new_root->right = p->right;
    if(p->right)p->right->parent = new_root;
}
else if(p has no left child)
{
    tree->root ← p->right;
    if(p has right child)p->right->parent ← NULL;
}
else
{
    tree->root = p->left;
    if(p has left child)p->left->parent ← NULL;
}
```

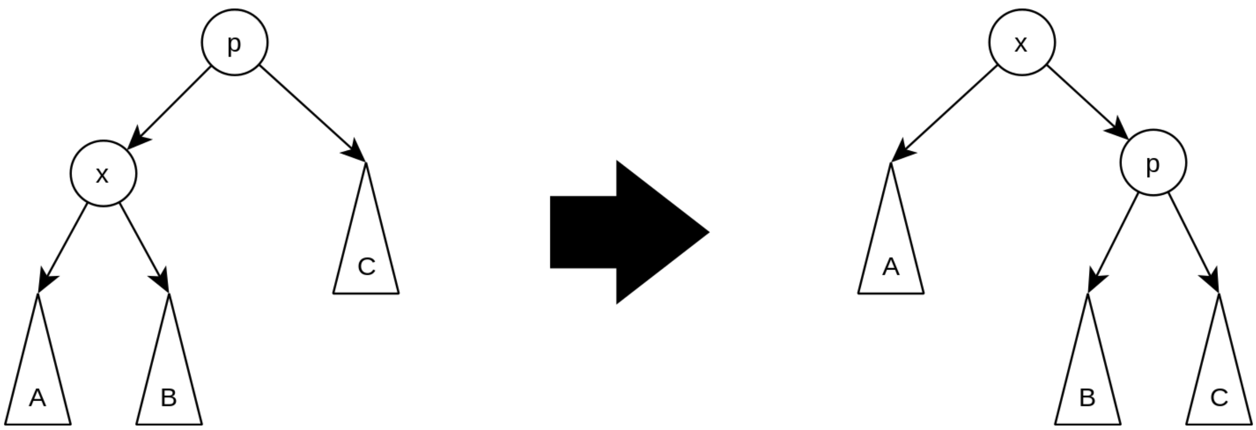
5. The insertion of certain nodes of value. It also returns the root of this sub-tree after insertion.

```
if(!tree->root)
{
```

```
    tree->root = (SplayPointer)malloc(sizeof(SplayNode));
    tree->root->value = value;
    tree->root->left = tree->root->right = tree->root->parent = NULL;
}
else
{
    SplayPointer p = splayInsNode(value, tree->root);
    tree->root = Splay(p, tree->root);
}
```

6. The 4 actions of splay tree: ZigZig, ZigZag, ZagZig and ZagZag. Here are our implementations. The 4 actions are used in the function splay. We also implement one function as an example, the rest can be referred.

For example, the Zig step is showed in the following picture.




```
SplayPointer Splay(SplayPointer p, SplayPointer root)
{
    if(p is root)return p;      //If p is the root, we need not to splay
    initialize finishSplay ← 0;     //to mark whether the splay is finished
    while(!finishSplay)
    {
        if(p->parent is root)   //If p is a child of root,
        {
            finishSplay ← 1;    //the splay will be finished in the next
 ↪  step
```

```

            if(p->value < root->value)  //if p is the left child
            {
                root->left ← p->right;  if(p->right)p->right->parent = root;
                p->right ← root;
            }
            else                        //if p is the right child
            {
                root->right ← p->left;  if(p->left)p->left->parent = root;
                p->left ← root;
            }
            root->parent ← p;
            p->parent ← NULL;
        }
        else
        {
            if(p->parent->parent is root) finishSplay ← 1;

            if(p->value < p->parent->value and p->value < p->parent->parent->value) ZigZig(p);

            else if(p->value > p->parent->value and p->value < p->parent->parent->value) ZigZag(p);

            else if(p->value < p->parent->value and p->value > p->parent->parent->value) ZagZig(p);

            else if(p->value > p->parent->value and p->value > p->parent->parent->value) ZagZag(p);
        }
    }
    return p;
}

Procedure ZigZig(SplayPointer p)
{
    SplayPointer A ← p, B ← p->parent, C ← B->parent;

    <!--B->left ← A->right;     if(A has right child)A->right->parent ← B;
    C->left ← B->right;     if(B has right child)B->right->parent ← C;
    A->right ← B;
```

```
    B->right ← C;
    A->parent ← C->parent;
    B->parent ← A;
    C->parent ← B;

    if(A->parent exists)
    {
        if(A->parent->value > A->value) A->parent->left ← A;
        else A->parent->right ← A;
    }
}
```

2.2.3 C

You can find C code in the appendix

2.2.4 Proof of correctness

After thorough testing, our results are as we expected. So we can tell that they are correct.

2.3 UnBalanced Search Tree

Its functions are mostly alike with AVL and Splay trees. Like build, free, find, insert and delete. So no more discussion will be stated here. You can find the specific description in the source code appendix.

3 Testing Results

3.1 Test Cases

- PS. Because the data is very large to present, so we just describe it and give out the results of cases. All the test cases can be found in the folder we handed in. The results are all repeated 50 times to be more obvious.
- We used a main program to test the results. We need to Input an integer 'n' first. Then input n integers, which will be inserted into the binary trees talked above in sequence. Again input n integers, which will be deleted from the trees in sequence. The output will consist of 3 lines

representing 3 trees respectively, and 2 floating numbers each line representing the time spent on insertion and deletion respectively.

The main program can be found in the appendix.

1. Increasing Order

   We designed different test cases of data range from 1000 to 10000. Each case is in increasing order. For example, case 10 is 10000 numbers from 1 to 10000.

2. Decreasing Order

   We designed different test cases of data range from 1000 to 10000. Each case is in decreasing order. For example, case 10 is 10000 numbers from 10000 down to 1.

3. Random Order

   We designed different test cases of data range from 1000 to 10000. Each case is in random order.

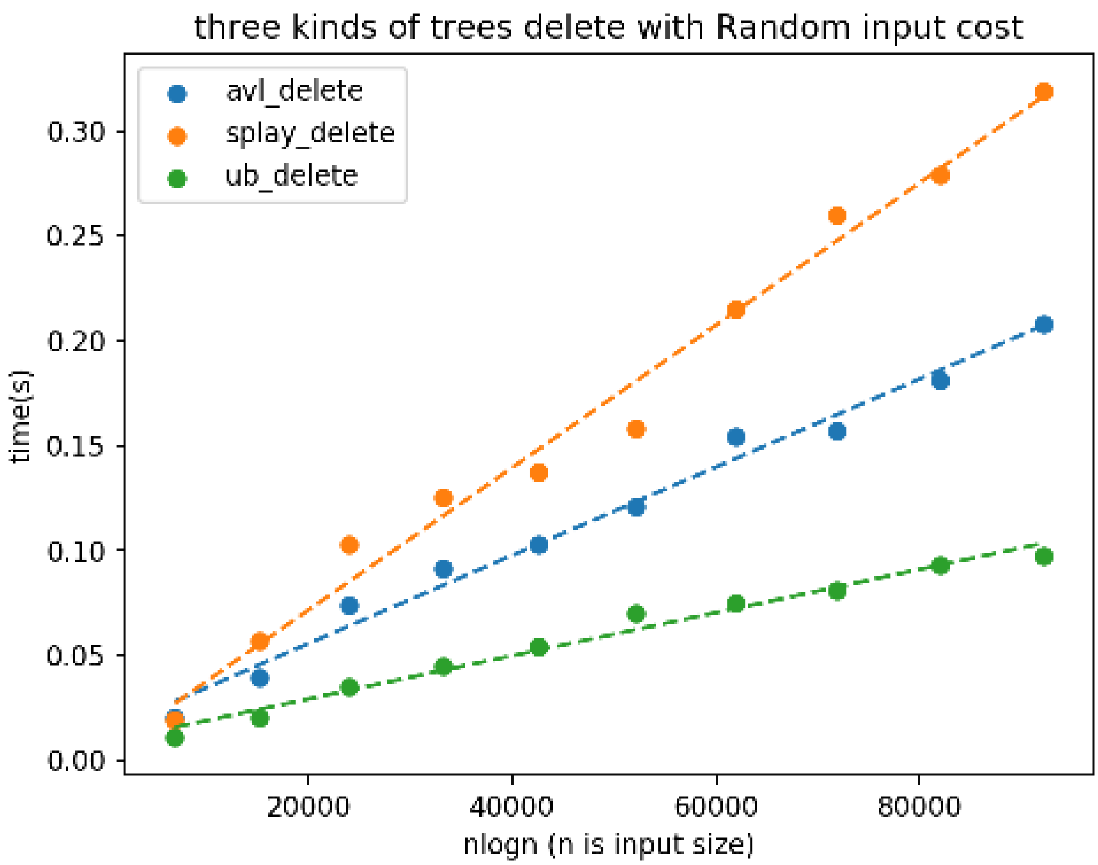
3.2 Test Results

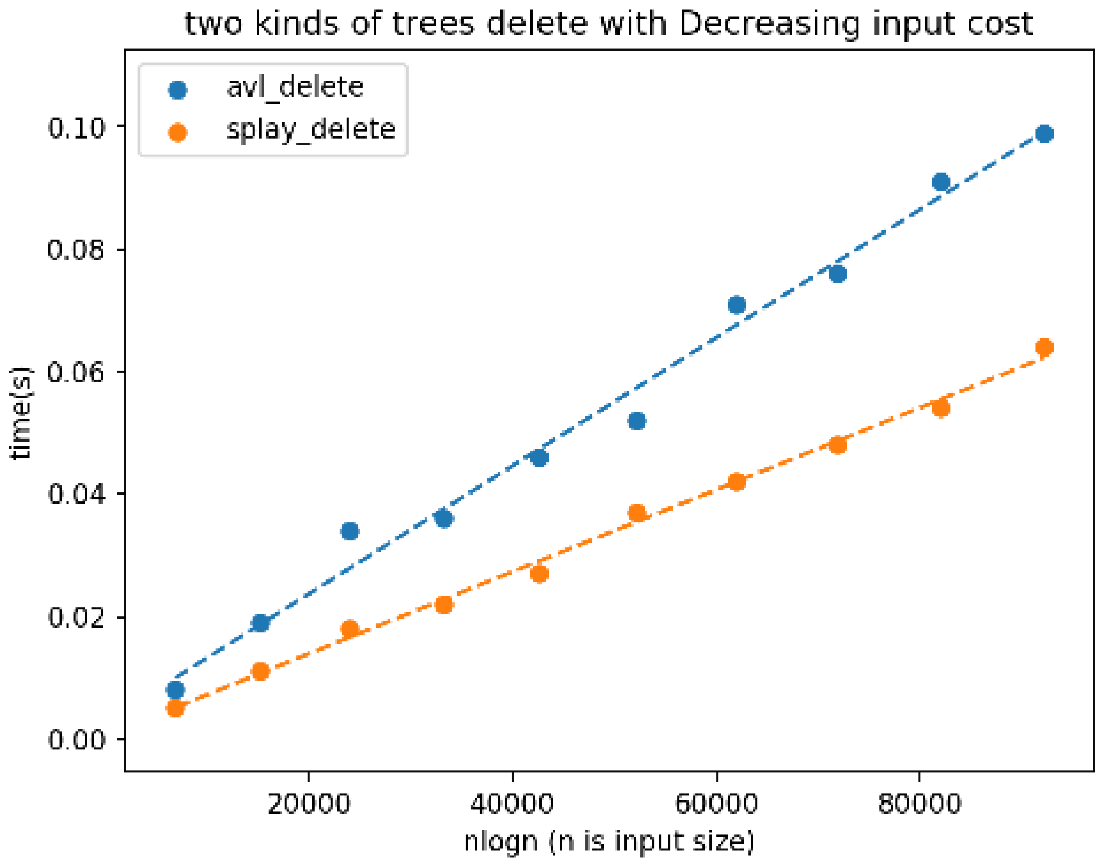
The result table is as followed.

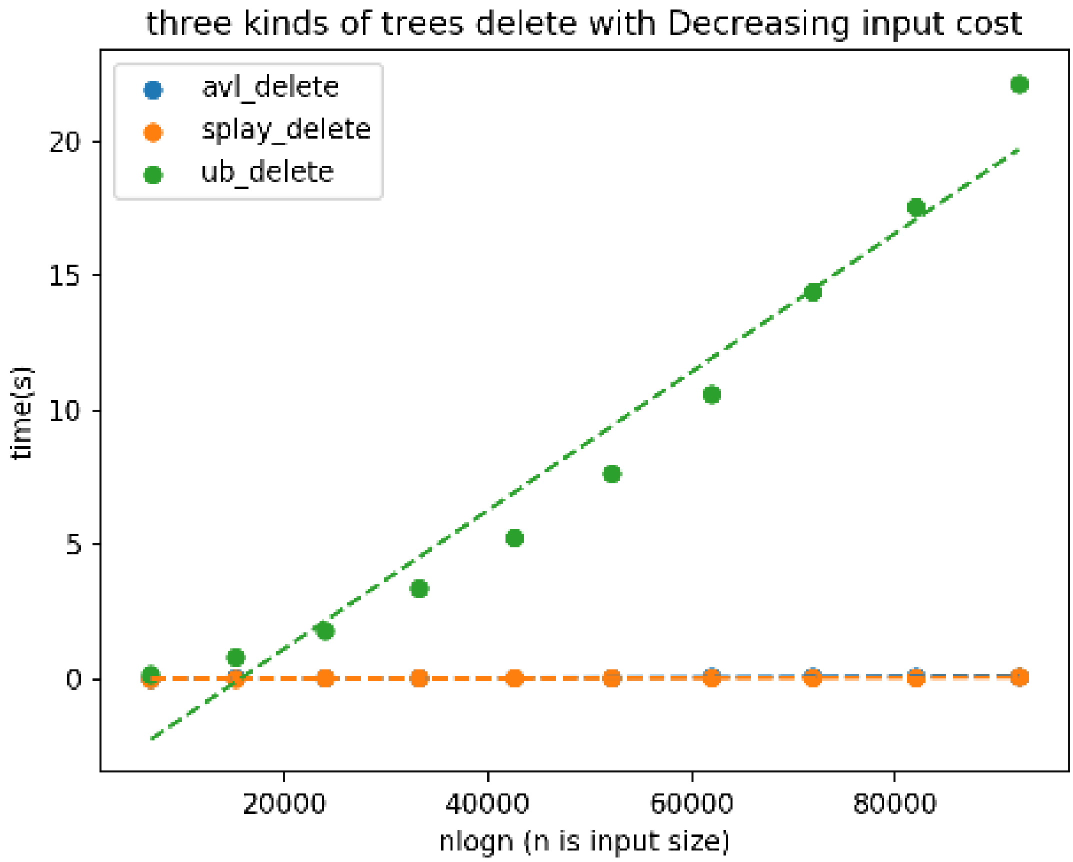
input_size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
avl_insert_increasing	0.007	0.016	0.017	0.032	0.058	0.058	0.066	0.086	0.084	0.095
avl_insert_decreasing	0.007	0.015	0.026	0.039	0.052	0.063	0.065	0.076	0.09	0.1
avl_insert_random	0.013	0.035	0.046	0.055	0.066	0.083	0.102	0.124	0.152	0.158
avl_delete_increasing	0.007	0.018	0.036	0.039	0.033	0.052	0.071	0.081	0.09	0.099
avl_delete_decreasing	0.008	0.019	0.034	0.036	0.046	0.052	0.071	0.076	0.091	0.099
avl_delete_random	0.013	0.024	0.06	0.074	0.096	0.118	0.14	0.168	0.183	0.211
splay_insert_increasing	0.004	0.01	0.011	0.012	0.016	0.023	0.021	0.024	0.03	0.037
splay_insert_decreasing	0.003	0.006	0.012	0.016	0.015	0.018	0.021	0.025	0.029	0.039
splay_insert_random	0.021	0.039	0.068	0.096	0.119	0.158	0.171	0.217	0.234	0.27
splay_delete_increasing	0.004	0.008	0.015	0.023	0.027	0.026	0.036	0.051	0.049	0.052
splay_delete_decreasing	0.005	0.011	0.018	0.022	0.027	0.037	0.042	0.048	0.054	0.064
splay_delete_random	0.017	0.048	0.08	0.105	0.132	0.172	0.204	0.252	0.271	0.321
ub_insert_increasing	0.118	0.499	1.184	2.058	3.23	4.717	6.412	8.52	10.78	13.522
ub_insert_decreasing	0.114	0.505	1.152	2.086	3.256	4.684	6.429	8.613	10.659	13.585
ub_insert_random	0.01	0.027	0.044	0.057	0.073	0.105	0.122	0.151	0.151	0.178

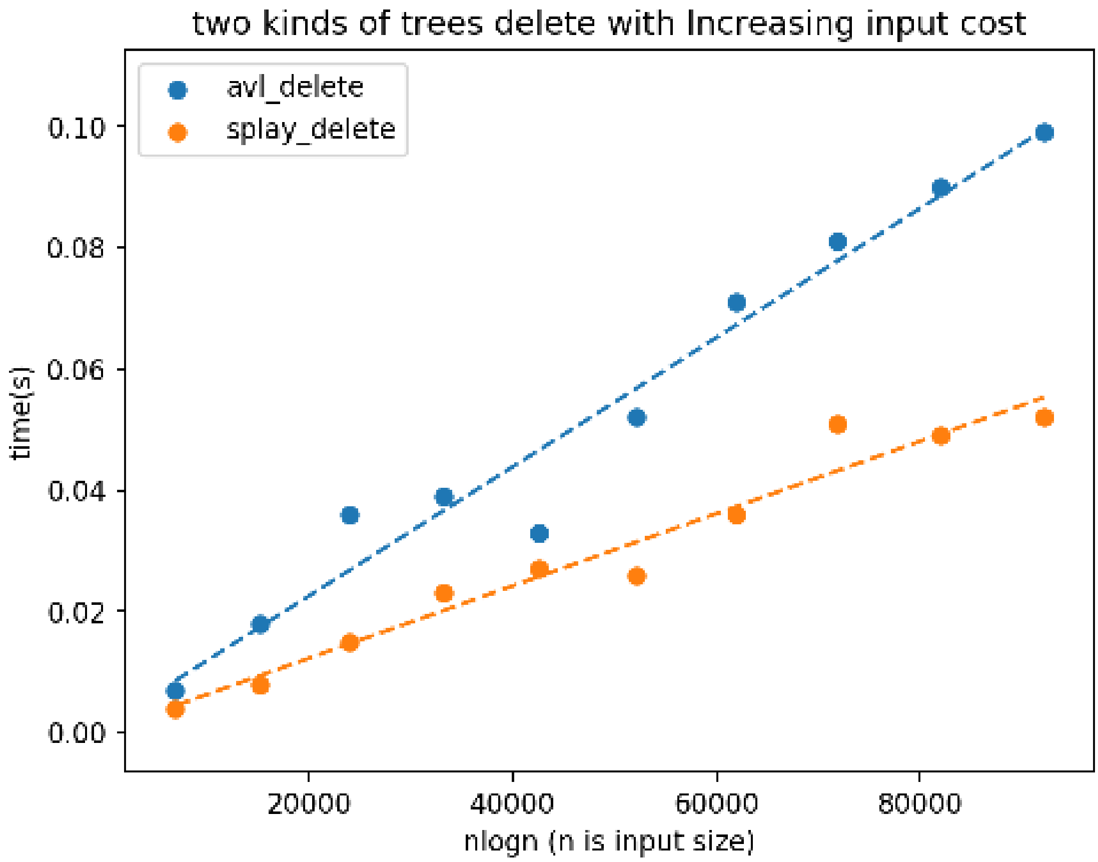
input_size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ub_delete_increasing	0.196	0.802	1.882	3.275	5.273	7.86	10.931	14.173	17.898	22.427
ub_delete_decreasing	0.186	0.813	1.807	3.389	5.269	7.668	10.583	14.401	17.571	22.157
ub_delete_random	0.013	0.014	0.021	0.04	0.049	0.079	0.074	0.09	0.091	0.103

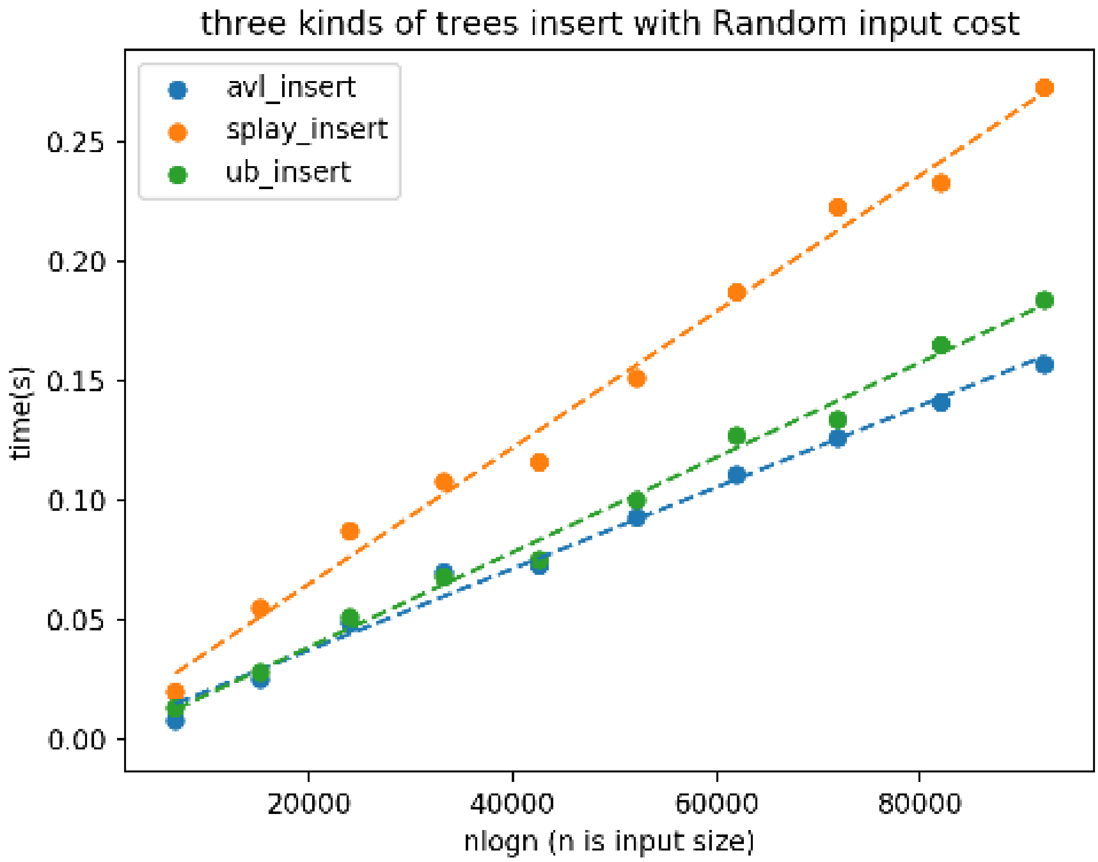
Then we used Python's matplotlib to draw a fitting curve. Because the pictures are too many, we selected some pictures that can present our idea.




two kinds of trees delete with Decreasing input cost
avl_delete
splay_delete
0.10
0.08
0.06
0.04
0.02
0.00
time(s)
20000
40000
60000
80000
nlogn (n is input size)


three kinds of trees delete with Decreasing input cost
avl_delete
splay_delete
ub_delete
time(s)
0
5
10
15
20
20000
40000
60000
80000
nlogn (n is input size)


two kinds of trees delete with Increasing input cost
avl_delete
splay_delete
time(s)
0.00
0.02
0.04
0.06
0.08
0.10
20000
40000
60000
80000
nlogn (n is input size)



We can see that unbalanced tree has some disadvantage in performing those functions.

4 Analysis and Comments

4.1 time complexities

4.1.1 AVL tree

1. Insertion

the insertion routine for avl tree follow the same process as inserting into a Binary Search Tree. When the node is inserted, avl tree should check whether the node's ancestors are still avl trees. Once the Balance Factor(BF) is changed to 2 or -2, then a single rotation or double rotation will be execute to keep the tree is balanced. The time for lookup is $O(logn)$, and the check of ancestors, which is the way back to the root, cost $O(logn)$ at most. The rotation takes $O(1)$ each time. So the insertion can be completed in $O(logn)$ time.

2. Deletion

the delete operation always try to find the max in the left and then subsitute the deleted node with the it. Then check all the ancestors of the max of the left subtree for possible rotation. If there is no left substree, then find the min of the right subtree, the rest same. The find process costs $O(logn)$ and the check with roration costs $O(logn)$, for the height is $logn$, so the delete operation costs $O(logn)$ time.

4.1.2 splay tree

the first step for splay tree insertion is the same as the insertion a normal binary search tree. Then a splay is performed. The new root will be the inserted one, as a result. Since the height of splay tree is uncertain, varying from $log(n)$ to n, (n is the number of nodes), the cost of insertion for splay tree is uncertain. The delete first find the node that to be deleted and do a splay to make it a root. Then find the max of the left subtree of new root(if there exist a left subtree) and splay the left substree to make the max of left the root of the left subtree. Next delete the new root, and make the right subtree be the left subtree's root's right subtree. For the same reason with the insertion, the cost of deletion can not be determined.

However, a amortized analysis of splay tree can be carried out using the potential method. Define:

- $S(r)$ = the number of nodes in the sub-tree rooted at node r (including r).
- $rank(r) = log(size(r))$.
- $\phi(T)$ = $\sum rank(allnodes)$ = the sum of the ranks of all the nodes in the tree.

So we first calculate the $\Delta\ \Phi$: the change in the potential caused by a splay operation. There is three case, zig, zig-zag, zig-zig. We discuss them separately. Denote by rank$'$ the rank function after the operation. x(the splayed node0, p(the parent of x) and g(the parent of p) are the nodes affected by the rotation operation.

- Zig:
  $\Delta\ \Phi$ = rank$'$ (p) - rank(p) + rank$'$ (x) - rank(x) [since only p and x change ranks] = rank$'$ (p) - rank(x) [since rank$'$ (x)=rank(p)] $\leq$ rank$'$ (x) - rank(x) [since rank$'$ (p)<rank$'$ (x)]
- Zig-Zig step:

$\Delta\ \Phi$ = rank$'$ (g) - rank(g) + rank$'$ (p) - rank(p) + rank$'$ (x) - rank(x) = rank$'$ (g) + rank$'$ (p) - rank(p) - rank(x) [since rank$'$ (x)=rank(g)] $\leq$ rank$'$ (g) + rank$'$ (x) - 2 rank(x) [since rank(x)rank$'$ (p)] $\leq$ 3(rank$'$ (x)-rank(x)) - 2 [due to the concavity of the log function]

- Zig-Zag step:

  $\Delta\ \Phi$ = rank$'$ (g) - rank(g) + rank$'$ (p) - rank(p) + rank$'$ (x) - rank(x) $\leq$ rank$'$ (g) + rank$'$ (p) - 2 rank(x) [since rank$'$ (x)=rank(g) and rank(x)<rank(p)] $\leq$ 2(rank$'$ (x)-rank(x)) - 2 [due to the concavity of the log function]

The amortized cost of any operation is $\Delta\ \Phi$ plus the actual cost. The actual cost of any zig-zig or zig-zag operation is no bigger than 2 since there are almost two rotations to make. amortized - cost = cost + $\Delta\ \Phi$ $\leq$ 3(rank'(x)-rank'(x))

When add all the x together, we got that amortized-cost=cost+$\Delta\ \Phi$=3(rank(root)-rank(x)), which is log(n).

so the amortized-cost for a splay that bring a node to the root is $O(logn)$. thus the amortized-cost for insertion and deletion is $O(logn)$

4.1.3 unbalanced tree:

the insertion for unbalanced tree is to find the proper position. Every time go left if smaller than the node or go right otherwise. Depending on the input, the operation costs $O(log(n))$ to $O(n)$. The worst case is that the input is increasing or decreasing. The best case is that the input always keep the tree to be a balanced tree.

the delete is similar to the insertion, cost from $O(log(n))$ to $O(n)$.

4.2 space complexities:

1. avl tree: each node cost $O(1)$ space, so n nodes cost $O(n)$ space. The recursion in insertion or deletion cost at most $O(logn)$ for the height of avl tree is $O(logn)$. So the space complexity is $O(n)$.

2. splay tree: each node cost $O(1)$ space, so n nodes cost $O(n)$ space. The recursion cost at insertion or deletion most $O(n)$ for the height of splay tree is $O(n)$. So the space complexity is $O(n)$.

3. unbalanced tree: each node cost $O(1)$ space, so n nodes cost $O(n)$ space. The recursion cost at deletion most $O(n)$ for the height of unbalanced tree is $O(n)$. So the space complexity is $O(n)$.

4.3 comments:

from the tests results we can see that in random order the unbalanced tree performs best, for it does not have any balanced operations and can still be relatively balanced for the input is random. However, when the input is increasing and decreasing the splay tree and avl tree works fine, having little difference with the performance when the input is random, while the performance of unbalanced is terrible. From the test result we can draw a conclusion that generally the ubalanced tree works fine for random input while avl tree and splay tree have excellent performance guarantees for different kinds of inputs.

5 Appendix

5.1 Source Code

5.1.1 AVL tree

```
// AvlTree.c
#include "AvlTree.h"

AvlPointer llrot(AvlPointer A);
AvlPointer lrrot(AvlPointer A);
AvlPointer rlrot(AvlPointer A);
AvlPointer rrrot(AvlPointer A);

// Build an empty AVL tree
AvlTree AVLBuild(void)
{
    AvlTree tree = (AvlTree)malloc(sizeof(AvlTreeNode));
    tree->root = NULL;
    return tree;
}

void AVL_Free_Node(AvlPointer node);    //Free the tree (or subtree) whose root is 'node'
//Free the whole tree
void AVLFree(AvlTree tree)
{
    if(tree)AVL_Free_Node(tree->root);  //If tree is valid, simply use AVL_Free_Node function
    free(tree);
```

```c
}
void AVL_Free_Node(AvlPointer node)
{
    if(node && node->left)AVL_Free_Node(node->left);
    if(node && node->right)AVL_Free_Node(node->right);
    free(node);
}

//Find the node with certain value in the tree
AvlPointer AVLFind(int value, AvlTree tree)
{
    AvlPointer p = tree->root;
    while(p && p->value != value){
        if(p->value < value)p = p->right;
        else p = p->left;
    }
    return p;
}

//Find the node with minimum value in the subtree whose root is p
AvlPointer AVLFindMinNode(AvlPointer p)
{
    while(p && p->left)p = p->left;
    return p;
}

//Find the node with minimum value in the tree
AvlPointer AVLFindMin(AvlTree tree)
{
    return AVLFindMinNode(tree->root);
}

//Find the node with maximum value in the subtree whose root is p
AvlPointer AVLFindMaxNode(AvlPointer p)
{
    while(p && p->right)p = p->right;
    return p;
}

//Find the node with maximum value in the tree
AvlPointer AVLFindMax(AvlTree tree)
```

```
{
    return AVLFindMaxNode(tree->root);
}

//Find the node with certain value in the tree whose root is p,
//it returns the root of the tree after deletion
AvlPointer AVLNodeDel(int value, AvlPointer p)
{
    int lh = 0, rh = 0;
    //lh means the height of the left subtree of p, while rh means the height of the right one

    if(p->value == value)       //if p is to be deleted
    {
        if(!p->left && !p->right)   // if p is a leaf node, then simply free it and assign p NULL
        {
            free(p);
            p = NULL;
        }
        else        // otherwise
        {
            if(p->left) //if p has left child, find 'maximum node' in the left subtree and replace p with it
            {
                AvlPointer lcmax = AVLFindMaxNode(p->left);
                // by replacing the value of p with its value and delete it in the left subtree of p
                p->value = lcmax->value;
                p->left = AVLNodeDel(lcmax->value, p->left);
            }
            else    // or the case is much simpler. we only need to replace p with the right child of p
            {
                AvlPointer rchild = p->right;
                free(p);
                p = rchild;
            }
        }
    }
    else
```

```
    {
        if(value < p->value)
        {
            p->left = AVLNodeDel(value, p->left);
        }
        else
        {
            p->right = AVLNodeDel(value, p->right);
        }
    }

    // Now the deletion is completed in the recursion above, we need to check
    // the height of two subtrees and do necessary rotations.
    if(p)
    {
        // assign the height of p
        if(p->left)lh = p->left->height + 1;    else lh = 0;
        if(p->right)rh = p->right->height + 1;  else rh = 0;
        p->height = lh > rh ? lh : rh;

        // the following code describes how to rotate in different cases
        if(lh - rh > 1)
        {
            int llh = p->left->left ? p->left->left->height + 1 : 0;
            int lrh = p->left->right ? p->left->right->height + 1 : 0;
            if(llh > lrh)p = llrot(p);
            else p = lrrot(p);
        }
        else if(lh - rh < -1)
        {
            int rlh = p->right->left ? p->right->left->height + 1 : 0;
            int rrh = p->right->right ? p->right->right->height + 1 : 0;
            if(rlh > rrh)p = rlrot(p);
            else p = rrrot(p);
        }
    }
    return p;
}

// Delete the node with certain value in the tree, if deletion fails (the
↪  node is not found), then return 0
```

```
// otherwise return 1
int AVLDel(int value, AvlTree tree)
{
    AvlPointer p;
    int lh = 0, rh = 0;

    p = AVLFind(value, tree);           //find the node to be deleted
    if(!p)return 0;                     // if not found, return 0
    else tree->root = AVLNodeDel(value, tree->root);     //else use AVLNodeDel
 ↪  function
    return 1;
}

//Get the value stored in an AvlPointer
int AVLGetValue(AvlPointer p)
{
    return p->value;
}


// Insert a node with value into the subtree whose root is p. It returns the
 ↪  root of this subtree after insertion.
AvlPointer AVLInsEle(int value, AvlPointer p);
// Insert a node with value into the tree
void AVLIns(int value, AvlTree tree)
{
    if(!tree)return;    //if the tree is invalid, do nothing

    if(!tree->root)     //if the tree is empty, create a node with value to
 ↪  be its root
    {
        AvlPointer p = (AvlPointer)malloc(sizeof(AvlNode));
        p->value = value; p->left = p->right = NULL;
        tree->root = p;
        p->height = 0;
    }
    else                //otherwise, use AVLInsEle function
        tree->root = AVLInsEle(value, tree->root);
}

AvlPointer AVLInsEle(int value, AvlPointer p)
```

```
{
    int lh = 0, rh = 0;
    // lh means the height of the left subtree of p + 1, while rh means the height of the right one + 1
    // the default value 0 means the height of NULL(-1) + 1

    if( !p ){                       // if p is NULL, create a new node
        p = (AvlPointer)malloc(sizeof(AvlNode));
        p->value = value;
        p->left = p->right = NULL;
        p->height = 0;
    }else if(value > p->value){      // in this case, insert value to the right subtree of p
        p->right = AVLInsEle(value, p->right);
    }else if(value < p->value){      // in this case, insert value to the left subtree of p
        p->left = AVLInsEle(value, p->left);
    }else{
        return p;
    }

    // assign the height of p
    if(p->left)lh = p->left->height + 1;
    if(p->right)rh = p->right->height + 1;
    p->height = lh > rh ? lh : rh;

    // the following code describes how to rotate in different cases
    if(lh - rh < -1){
        if(value > p->right->value)
            p = rrrot(p);
        else
            p = rlrot(p);
    }else if(lh - rh > 1){
        if(value < p->left->value)
            p = llrot(p);
        else
            p = lrrot(p);
    }

    return p;
}
```

```

// single left rotation
AvlPointer llrot(AvlPointer A){
    AvlPointer B = A->left;

    A->left = B->right;
    B->right = A;
    A->height -= 2;

    return B;
}

// single right rotation
AvlPointer rrrot(AvlPointer A){
    AvlPointer B = A->right;

    A->right = B->left;
    B->left = A;
    A->height -= 2;

    return B;
}

// double left right rotation
AvlPointer lrrot(AvlPointer A){
    AvlPointer B = A->left;
    AvlPointer C = B->right;

    A->left = C->right;
    C->right = A;
    B->right = C->left;
    C->left = B;
    A->height -= 2;
    B->height --;
    C->height ++;
    return C;
}

// double right left rotation
AvlPointer rlrot(AvlPointer A){
    AvlPointer B = A->right;
```

```
    AvlPointer C = B->left;

    A->right = C->left;
    C->left = A;
    B->left = C->right;
    C->right = B;
    A->height -= 2;
    B->height --;
    C->height ++;
    return C;
}
```

```
/**
    No Copyright. But if you copy this code, you may be verified as cheating
    in ZJU.
    BE CAREFUL!

    This piece of code defines AVL tree and some algorithms to it.
*/


#ifndef AVL_H
#define AVL_H

#include <stdio.h>
#include <stdlib.h>


struct _avlnode;
typedef struct _avlnode AvlNode;
typedef AvlNode* AvlPointer;
//=========================================================
// AvlPointer is a pointer pointing at a node in the AVL tree.
// Users don't need to know the details in this block.
// Please use function 'AVLGetValue' to retrieve the value stored in the
    node.
struct _avlnode
{
    int value;
    AvlPointer left;
    AvlPointer right;
```

```
    int height;
};
//=========================================================
typedef struct _avltree
{
    AvlPointer root;
} AvlTreeNode;
typedef AvlTreeNode* AvlTree;
// Declare 'AvlTree' varible to use the functions followed.

AvlTree AVLBuild(void);
// Build an empty AVL tree and return it
// An example to use this function:         AvlTree tree = AVLBuild();

void AVLIns(int value, AvlTree tree);
// Insert a node with its value to be 'value' into the tree
// An example to use this function:         AVLIns(0, tree);

AvlPointer AVLFind(int value, AvlTree tree);
AvlPointer AVLFindMin(AvlTree tree);
AvlPointer AVLFindMax(AvlTree tree);
// Find the node with (certain/minimal/maximal)value in the tree and return
  ↪  its pointer.
// Return NULL if not found
// Examples to use these functions:         AvlPointer p = AVLFind(0, tree);
  ↪  if(!p)printf("Not found!\n");
//                                          AvlPointer min =
  ↪  AVLFindMin(tree);
//                                          AvlPointer max =
  ↪  AVLFindMax(tree);

int AVLDel(int value, AvlTree tree);
// Delete the node with certain value in the tree.
// Return 1 if the deletion succeeded, and 0 if not (which means the node
  ↪  doesn't exist).
// An example to use this function:         int suc = AVLDel(0, tree);
  ↪  if(!suc)printf("Deletion failed!\n");

void AVLFree(AvlTree tree);
// Free the whole tree
// An example to use this function:         AVLFree(tree);
```

```

int AVLGetValue(AvlPointer p);
// When you get an AvlPointer, use this function to access the value stored
 ↪ in it.
// An example to use this function:         AvlPointer min =
 ↪ AVLFindMin(tree);
//                                          if(!min) printf("The tree is
 ↪ empty!\n");
//                                          else printf("The minimum value is
 ↪ %d\n", AVLGetValue(min));

#endif
```

5.1.2 Splay Tree

```
// SplayTree.c
#include "SplayTree.h"

// splay p in the subtree whose root is 'root'
SplayPointer Splay(SplayPointer p, SplayPointer root);

// Build an empty splay tree and return it
SplayTree splayBuild(void)
{
    SplayTree t = (SplayTree)malloc(sizeof(SplayTreeNode));
    if(t)t->root = NULL;
    return t;
}

// Insert a node with value into the subtree whose root is p.
// It returns the root of this subtree after insertion.
SplayPointer splayInsNode(int value, SplayPointer p);
// Insert a node with its value to be 'value' into the tree
void splayIns(int value, SplayTree tree)
{
    if(!tree)return;    //if the tree is invalid, do nothing

    if(!tree->root)     //if the tree is empty, create a node with value to
 ↪ be its root
    {
        tree->root = (SplayPointer)malloc(sizeof(SplayNode));
```

```
        tree->root->value = value;
        tree->root->left = tree->root->right = tree->root->parent = NULL;
    }
    else                //otherwise, use splayInsNode function, and then splay p in the whole tree
    {
        SplayPointer p = splayInsNode(value, tree->root);
        tree->root = Splay(p, tree->root);
    }
}

SplayPointer splayInsNode(int value, SplayPointer p)
{
    while(p && p->value != value)   //if p->value == value, there is no need to insert
    {
        if(p->value < value)        //search the position in right subtree
        {
            if(!p->right)           //insert it to the left position of p and break
            {
                p->right = (SplayPointer)malloc(sizeof(SplayNode));
                p->right->value = value;
                p->right->parent = p;
                p->right->left = p->right->right = NULL;
                break;
            }
            p = p->right;
        }
        else                        //search the position in left subtree
        {
            if(!p->left)            //insert it to the left position of p and break
            {
                p->left = (SplayPointer)malloc(sizeof(SplayNode));
                p->left->value = value;
                p->left->parent = p;
                p->left->left = p->left->right = NULL;
            }
            p = p->left;
        }
```

```
    }
    return p;
}

// Find the node with certain value in the tree and return its pointer.
SplayPointer splayFind(int value, SplayTree tree)
{
    SplayPointer p = tree->root;
    // find the node with value
    while(p && p->value != value)
    {
        if(p->value < value)p = p->right;
        else p = p->left;
    }
    // if found and it isn't the root, splay the node, otherwise simply
    ↪ return the pointer
    if(p && p != tree->root)tree->root = Splay(p, tree->root);
    return p;
}

SplayPointer splay_find_min(SplayPointer root)
{
    SplayPointer p = root;
    while(p && p->left) p = p->left;
    return p;
}
// Find the node with minimum value in the tree and return its pointer.
SplayPointer splayFindMin(SplayTree tree)
{
    //find the minimum node and splay it
    SplayPointer p = splay_find_min(tree->root);
    if(p)tree->root = Splay(p, tree->root);
    return p;
}

SplayPointer splay_find_max(SplayPointer root)
{
    SplayPointer p = root;
    while(p && p->right) p = p->right;
    return p;
}
```

```c
// Find the node with maximum value in the tree and return its pointer.
SplayPointer splayFindMax(SplayTree tree)
{
    //find the maximum node and splay it
    SplayPointer p = splay_find_max(tree->root);
    if(p)tree->root = Splay(p, tree->root);
    return p;
}

// Delete the node with certain value in the tree.
// Return 1 if the deletion succeeded, and 0 if not (which means the node doesn't exist).
int splayDel(int value, SplayTree tree)
{
    // Find the node. If found, it will be placed at the root of the tree.
    SplayPointer p = splayFind(value, tree);
    if(!p)return 0; //If not found, return 0.

    // If p is deleted, the whole tree will be divided into two parts at most
    SplayPointer new_root = NULL;
    // If p has both children,
    if(p->left && p->right)
    {
        // find the maximum node in the left subtree and splay it to the root of the left subtree
        new_root = splay_find_max(p->left);
        tree->root = Splay(new_root, p->left);
        new_root->parent = NULL;
        // as the new_root is the maximum node in the subtree, it has no right children
        // we can simply make the right child of p be the right child of new_root
        new_root->right = p->right;
        if(p->right)p->right->parent = new_root;
    }
    // If p has no left child, simply make the right subtree to be the new tree
    else if(!p->left)
    {
        tree->root = p->right;
        if(p->right)p->right->parent = NULL;
```

```
    }
    // If p has no right child, simply make the left subtree to be the new
    ↪ tree
    else
    {
        tree->root = p->left;
        if(p->left)p->left->parent = NULL;
    }

    free(p);
    return 1;
}

//Free the tree (or subtree) whose root is 'node'
void splay_Free_Node(SplayPointer node)
{
    if(node && node->left)splay_Free_Node(node->left);
    if(node && node->right)splay_Free_Node(node->right);
    free(node);
}
//Free the whole tree
void splayFree(SplayTree tree)
{
    if(tree)splay_Free_Node(tree->root);
    free(tree);
}

//Get the value stored in a splayPointer
int splayGetValue(SplayPointer p)
{
    return p->value;
}

//=============================
// The four actions using in splay function
void ZigZig(SplayPointer p);
void ZigZag(SplayPointer p);
void ZagZig(SplayPointer p);
void ZagZag(SplayPointer p);
//=============================
SplayPointer Splay(SplayPointer p, SplayPointer root)
```

```
{
    if(p == root)return p;       //If p is the root, we needn't to splay
    int finishSplay = 0;         //to mark whether the splay is finished
    while(!finishSplay)
    {
        if(p->parent == root)    //If p is a child of root,
        {
            finishSplay = 1;     //the splay will be finished in the next step

            if(p->value < root->value)  //if p is the left child
            {
                root->left = p->right;  if(p->right)p->right->parent = root;
                p->right = root;
            }
            else                        //if p is the right child
            {
                root->right = p->left;  if(p->left)p->left->parent = root;
                p->left = root;
            }
            root->parent = p;
            p->parent = NULL;
        }
        else
        {
            if(p->parent->parent == root)finishSplay = 1;

            if(p->value < p->parent->value && p->value < p->parent->parent->value) ZigZig(p);       //     /
            //    /
            //   p

            else if(p->value > p->parent->value && p->value < p->parent->parent->value) ZigZag(p);  //   /
            //   \
            //    p

            else if(p->value < p->parent->value && p->value > p->parent->parent->value) ZagZig(p);  //   \
```

```
                                                                                  //   /
                                                                                  //  p

            else if(p->value > p->parent->value && p->value > p->parent->parent->value) ZagZag(p);  //   \
                                                                                  //     \
                                                                                  //      p
        }
    }
    return p;
}

void ZigZig(SplayPointer p)
{
    SplayPointer A = p, B = p->parent, C = B->parent;

    B->left = A->right;     if(A->right)A->right->parent = B;
    C->left = B->right;     if(B->right)B->right->parent = C;
    A->right = B;
    B->right = C;
    A->parent = C->parent;
    B->parent = A;
    C->parent = B;

    if(A->parent)
    {
        if(A->parent->value > A->value) A->parent->left = A;
        else A->parent->right = A;
    }
}

void ZigZag(SplayPointer p)
{
    SplayPointer A = p, B = p->parent, C = B->parent;

    B->right = A->left;     if(A->left)A->left->parent = B;
    C->left = A->right;     if(A->right)A->right->parent = C;
```

```
    A->left = B;
    A->right = C;
    A->parent = C->parent;
    B->parent = A;
    C->parent = A;

    if(A->parent)
    {
        if(A->parent->value > A->value) A->parent->left = A;
        else A->parent->right = A;
    }
}

void ZagZig(SplayPointer p)
{
    SplayPointer A = p, B = p->parent, C = B->parent;

    B->left = A->right;     if(A->right)A->right->parent = B;
    C->right = A->left;     if(A->left)A->left->parent = C;
    A->right = B;
    A->left = C;
    A->parent = C->parent;
    B->parent = A;
    C->parent = A;

    if(A->parent)
    {
        if(A->parent->value > A->value) A->parent->left = A;
        else A->parent->right = A;
    }
}

void ZagZag(SplayPointer p)
{
    SplayPointer A = p, B = p->parent, C = B->parent;

    B->right = A->left;     if(A->left)A->left->parent = B;
    C->right = B->left;     if(B->left)B->left->parent = C;
    A->left = B;
    B->left = C;
    A->parent = C->parent;
```

```
    B->parent = A;
    C->parent = B;

    if(A->parent)
    {
        if(A->parent->value > A->value) A->parent->left = A;
        else A->parent->right = A;
    }
}
```

```
/**
    No Copyright. But if you copy this code, you may be verified as cheating
 ↪  in ZJU.
    BE CAREFUL!

    This piece of code defines splay tree and some algorithms to it.
*/

#ifndef SPLAY_H
#define SPLAY_H

#include <stdio.h>
#include <stdlib.h>

typedef struct _splaynode SplayNode;
typedef SplayNode* SplayPointer;
//=========================================================
// SplayPointer is a pointer pointing at a node in the splay tree.
// Users don't need to know the details in this block.
// Please use function 'SplayGetValue' to retrieve the value stored in the
 ↪  node.
struct _splaynode{
    int value;
    SplayPointer left;
    SplayPointer right;
    SplayPointer parent;
};

typedef struct _splaytree
{
    SplayPointer root;
```

```
} SplayTreeNode;
typedef SplayTreeNode* SplayTree;
// Declare 'SplayTree' varible to use the functions followed.

SplayTree splayBuild(void);
// Build an empty splay tree and return it
// An example to use this function:         SplayTree tree = splayBuild();

void splayIns(int value, SplayTree tree);
// Insert a node with its value to be 'value' into the tree
// An example to use this function:         splayIns(0, tree);

SplayPointer splayFind(int value, SplayTree tree);
SplayPointer splayFindMin(SplayTree tree);
SplayPointer splayFindMax(SplayTree tree);
// Find the node with (certain/minimal/maximal)value in the tree and return
 ↪  its pointer.
// Return NULL if not found
// Examples to use these functions:         SplayPointer p = splayFind(0,
 ↪  tree);    if(!p)printf("Not found!\n");
//                                          SplayPointer min =
 ↪  splayFindMin(tree);
//                                          SplayPointer max =
 ↪  splayFindMax(tree);

int splayDel(int value, SplayTree tree);
// Delete the node with certain value in the tree.
// Return 1 if the deletion succeeded, and 0 if not (which means the node
 ↪  doesn't exist).
// An example to use this function:         int suc = splayDel(0, tree);
 ↪  if(!suc)printf("Deletion failed!\n");

void splayFree(SplayTree tree);
// Free the whole tree
// An example to use this function:         splayFree(tree);

int splayGetValue(SplayPointer p);
// When you get an SplayPointer, use this function to access the value stored
 ↪  in it.
// An example to use this function:         SplayPointer min =
 ↪  splayFindMin(tree);
```

```
//                                      if(!min) printf("The tree is
    empty!\n");
//                                      else printf("The minimum value is
    %d\n", splayGetValue(min));

#endif
```

5.1.3 UBST

```
#include "UBTree.h"

// Build an empty unbalanced tree and return it
UBTree UBBuild(void)
{
    UBTree t = (UBTree)malloc(sizeof(UBTreeNode));
    t->root = NULL;
    return t;
}

// Insert a node with value into the tree
void UBIns(int value, UBTree tree)
{
    if(!tree)return;    //if the tree is invalid, do nothing

    if(!tree->root)     //if the tree is empty, create a node with value to
    be its root
    {
        tree->root = (UBPointer)malloc(sizeof(UBNode));
        tree->root->value = value;
        tree->root->left = tree->root->right = NULL;
    }
    else
    {                   //search the proper position to insert
        UBPointer p = tree->root;
        while(p->value != value)    //if p->value == value, there is no need
    to insert
        {
            if(value < p->value)    //search the position in left subtree
            {
                if(!p->left)        //insert it to the left position of p and
    break
```

```
                {
                    p->left = (UBPointer)malloc(sizeof(UBNode));
                    p->left->value = value;
                    p->left->left = p->left->right = NULL;
                    break;
                }
                p = p->left;
            }
            else                    //search the position in left subtree
            {
                if(!p->right)       //insert it to the left position of p and break
                {
                    p->right = (UBPointer)malloc(sizeof(UBNode));
                    p->right->value = value;
                    p->right->left = p->right->right = NULL;
                    break;
                }
                p = p->right;
            }
        }
    }
}

// Find the node with certain value in the tree and return its pointer.
UBPointer UBFind(int value, UBTree tree)
{
    UBPointer p = tree->root;
    while(p && p->value != value)
    {
        if(value < p->value)p = p->left;
        else p = p->right;
    }
    return p;
}

// Find the node with minimum value in the tree and return its pointer.
UBPointer UBFindMin(UBTree tree)
{
    UBPointer p = tree->root;
    while(p && p->left)p = p->left;
```

```
    return p;
}

// Find the node with maximum value in the tree and return its pointer.
UBPointer UBFindMax(UBTree tree)
{
    UBPointer p = tree->root;
    while(p && p->right)p = p->right;
    return p;
}

//Find the node with certain value in the tree whose root is p,
//it returns the root of the tree after deletion
UBPointer UBNodeDel(int value, UBPointer p)
{
    if(value < p->value) p->left = UBNodeDel(value, p->left); //delete from left subtree
    else if(value > p->value) p->right = UBNodeDel(value, p->right); //delete from right subtree
    else
    {                                                                         //delete p
        if(!p->left && !p->right)                   //if p is a leaf, simply free it
        {
            free(p);
            p = NULL;
        }
        else if(p->left && p->right)                //if p has both children
        {                                           //swap p with its left child, delete from the left subtree
            p->value = p->left->value;
            p->left->value = value;
            p->left = UBNodeDel(value, p->left);
        }
        else if(!p->left)                           //if p doesn't has left child, return the right child of p
        {                                           //and free p
            UBPointer child = p->right;
            free(p);
            p = child;
```

```
        }
        else                                    //if p doesn't has right child, return the left child of p
        {                                       //and free p
            UBPointer child = p->left;
            free(p);
            p = child;
        }
    }
    return p;
}

// Delete the node with certain value in the tree, if deletion fails (the node is not found), then return 0
// otherwise return 1
int UBDel(int value, UBTree tree)
{
    UBPointer p = UBFind(value, tree);      //find the node to be deleted
    if(!p) return 0;                        // if not found, return 0
    else tree->root = UBNodeDel(value, tree->root); //else use UBNodeDel function
    return 1;
}

//Free the tree (or subtree) whose root is p
void UBFreeNode(UBPointer p);
//Free the whole tree
void UBFree(UBTree tree)
{
    if(tree)UBFreeNode(tree->root);
    free(tree);
}
void UBFreeNode(UBPointer p)
{
    if(p && p->left)UBFreeNode(p->left);
    if(p && p->right)UBFreeNode(p->right);
    free(p);
}

//Get the value stored in a UBPointer
int UBGetValue(UBPointer p)
```

```
{
    return p->value;
}
```

```
/**
    No Copyright. But if you copy this code, you may be verified as cheating
  ↪ in ZJU.
    BE CAREFUL!

    This piece of code defines simple binary search tree (unbalanced tree)
  ↪ and some algorithms to it.
*/

#ifndef UBT_H
#define UBT_H

#include <stdio.h>
#include <stdlib.h>

typedef struct _ubnode UBNode;
typedef UBNode* UBPointer;
//=========================================================
// UBPointer is a pointer pointing at a node in the unbalanced tree.
// Users don't need to know the details in this block.
// Please use function 'UBGetValue' to retrieve the value stored in the node.
struct _ubnode
{
    int value;
    UBPointer left, right;
};
//=========================================================

typedef struct _ubtreenode
{
    UBPointer root;
} UBTreeNode;
typedef UBTreeNode* UBTree;
// Declare 'UBTree' varible to use the functions followed.

UBTree UBBuild(void);
// Build an empty unbalanced tree and return it
```

```
// An example to use this function:         UBTree tree = UBBuild();

void UBIns(int value, UBTree tree);
// Insert a node with its value to be 'value' into the tree
// An example to use this function:         UBIns(0, tree);

UBPointer UBFind(int value, UBTree tree);
UBPointer UBFindMin(UBTree tree);
UBPointer UBFindMax(UBTree tree);
// Find the node with (certain/minimal/maximal)value in the tree and return its pointer.
// Return NULL if not found
// Examples to use these functions:         UBPointer p = UBFind(0, tree); if(!p)printf("Not found!\n");
//                                          UBPointer min = UBFindMin(tree);
//                                          UBPointer max = UBFindMax(tree);

int UBDel(int value, UBTree tree);
// Delete the node with certain value in the tree.
// Return 1 if the deletion succeeded, and 0 if not (which means the node doesn't exist).
// An example to use this function:         int suc = UBDel(0, tree); if(!suc)printf("Deletion failed!\n");

void UBFree(UBTree tree);
// Free the whole tree
// An example to use this function:         UBFree(tree);

int UBGetValue(UBPointer p);
// When you get an UBPointer, use this function to access the value stored in it.
// An example to use this function:         UBPointer min = UBFindMin(tree);
//                                          if(!min) printf("The tree is empty!\n");
//                                          else printf("The minimum value is %d\n", UBGetValue(min));

#endif
```

5.1.4 Main

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "AvlTree.h"
#include "SplayTree.h"
#include "UBTree.h"

clock_t start, stop;
double duration;

int *ins, *del;

int main()
{
    int n;
    AvlTree a = AVLBuild();
    SplayTree s = splayBuild();
    UBTree u = UBBuild();

    scanf("%d", &n);
    ins = (int*)malloc(sizeof(int) * n);
    del = (int*)malloc(sizeof(int) * n);
    if(!ins || !del)
    {
        printf("Error when allocating space in memory!\n");
        exit(1);
    }

    for(int ri = 0; ri < 2 * n; ri++)
    {
        if(ri < n)scanf("%d", ins + ri);
        else scanf("%d", del + ri - n);
    }

    start = clock();
    for(int ri = 0; ri < n; ri++)AVLIns(ins[ri], a);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f ", duration);
    start = clock();
```

```
    for(int ri = 0; ri < n; ri++)AVLDel(del[ri], a);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f\n", duration);

    start = clock();
    for(int ri = 0; ri < n; ri++)splayIns(ins[ri], s);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f ", duration);
    start = clock();
    for(int ri = 0; ri < n; ri++)splayDel(del[ri], s);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f\n", duration);

    start = clock();
    for(int ri = 0; ri < n; ri++)UBIns(ins[ri], u);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f ", duration);
    start = clock();
    for(int ri = 0; ri < n; ri++)UBDel(del[ri], u);
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("%f\n", duration);

    AVLFree(a);
    splayFree(s);
    UBFree(u);

    return 0;
}
```

5.2 Reference

[1] Wikipedia: AVL tree (https://en.wikipedia.org/wiki/Splay_tree)

[2] Wikipedia: Splay tree (https://en.wikipedia.org/wiki/Avl_tree)

[3] Weiss M A. Data structures and algorithm analysis[M]. The Benjamin/Cummings Pub. Co. Inc, 1992.

5.3 Arthur List

Programmer: ?

Tester: ?

Report Writer: ?

5.4 Declaration

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

5.5 Signatures