

Binary Search Trees

Author Names: ?

Date: 2017-3-6

Binary Search Trees

Group ?

1	Introduction	2
1.1	Concepts Description	2
1.2	Project Description	3
2	Algorithm Specification	3
2.1	Implement of AVL Tree	3
2.2	Implement of Splay Tree	8
2.3	UnBalanced Search Tree	12
3	Testing Results	12
3.1	Test Cases	12
3.2	Test Results	13
4	Analysis and Comments	18
4.1	time complexities	18
4.2	space complexities:	20
4.3	comments:	21
5	Appendix	21
5.1	Source Code	21
5.2	Reference	47
5.3	Arthur List	48

5.4 Declaration	48
5.5 Signatures	48

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 $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ 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:

```

1 struct _avlnode;
2 typedef struct _avlnode AvlNode;
3 typedef AvlNode* AvlPointer;

```

The specific description is like below:

```

1 struct _avlnode
2 {
3     int value;
4     AvlPointer left;
5     AvlPointer right;
6     int height;
7 };
8 typedef struct _avltree
9 {
10     AvlPointer root;
11 } AvlTreeNode;
12 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.

```

1 AvlPointer find(int value, AvlTree tree)
2 {
3     AvlPointer p ← tree->root;
4     while(p != NULL and p->value != value){
5         if (p->value < value) p ← p->right;
6         else p ← p->left;
7     }
8     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.

```

1  if(p has left child)
2  {
3      AvlPointer lcmax ← AVLFindMaxNode(p->left);
4      p->value ← lcmax->value;
5      p->left ← AVLNodeDel(lcmax->value, p->left);
6  }
7  else
8  {
9      AvlPointer rchild ← p->right;
10     free(p);
11     p ← rchild;
12 }

```

Then, we adjust the height of nodes.

```

1  if(p exists)
2  {
3      if(p has left child)lh ← p->left->height + 1; else lh = 0;
4      if(p has right child)rh ← p->right->height + 1; else rh = 0;
5      p->height ← max(lh, rh);
6
7      if(p is not balanced and left is higher)
8      {
9          adjust llh and lrh
10         if(llh > lrh)p ← llrot(p);
11         else p ← lrrot(p);
12     }
13     else if(p is not balanced and right is higher)
14     {
15         adjust rlh and rrh
16         if(rlh > rrh)p ← rlrot(p);
17         else p ← rrrot(p);
18     }
19 }

```

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.

```

1 // assign the height of p
2 if(p has left child)lh ← p->left->height + 1;
3 if(p has right child)rh ← p->right->height + 1;
4 p->height ← max(lh, rh);
5 // the following code describes how to rotate in different cases
6 if(p is not balanced and right is higher){
7     if(value > p->right->value)
8         p ← rrrot(p);
9     else
10        p ← rlrot(p);
11 }else if(p is not balanced and left is higher){
12     if(value < p->left->value)
13         p ← llrot(p);
14     else
15         p ← lrrot(p);
16 }

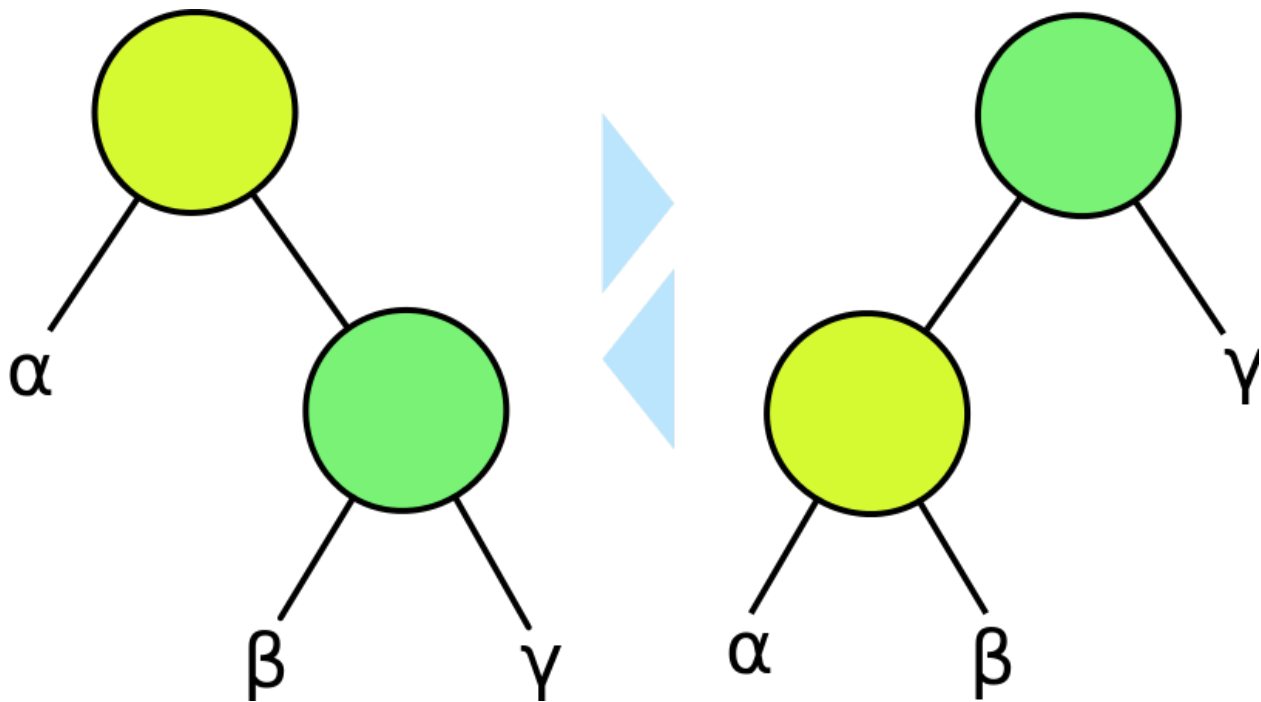
```

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.

```

1 // single left rotation
2 AvlPointer llrot(AvlPointer A){
3     AvlPointer B ← A->left;
4
5     A->left ← B->right;
6     B->right ← A;
7     A->height ← A->height - 2;
8
9     return B;
10 }
11
12 // double left right rotation
13 AvlPointer lrrot(AvlPointer A){
14     AvlPointer B ← A->left;
15     AvlPointer C ← B->right;
16

```



```

17   A->left ← C->right;
18   C->right ← A;
19   B->right ← C->left;
20   C->left ← B;
21   A->height ← A->height - 2;
22   B->height ← B->height - 1;
23   C->height ← C->height + 1;
24   return C;
25 }

```

- 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:

```

1  AvlPointer lrrot(AvlPointer A) {
2      A->left = rrrot(A->left);
3      return lrrot(A);
4  }

```


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

```
1 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.

```
1 typedef struct _splaynode SplayNode;
2 typedef SplayNode* SplayPointer;
3 struct _splaynode{
4     int value;
5     SplayPointer left;
6     SplayPointer right;
7     SplayPointer parent;
8 };
9
10 typedef struct _splaytree
11 {
12     SplayPointer root;
13 } SplayTreeNode;
14 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.

```

1  if(p has both children)
2  {
3      new_root ← splay_find_max(p->left);
4      tree->root ← Splay(new_root, p->left);
5      new_root->parent ← NULL;
6      new_root->right = p->right;
7      if(p->right)p->right->parent = new_root;
8  }
9  else if(p has no left child)
10 {
11     tree->root ← p->right;
12     if(p has right child)p->right->parent ← NULL;
13 }
14 else
15 {
16     tree->root = p->left;
17     if(p has left child)p->left->parent ← NULL;
18 }

```

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

```

1  if(!tree->root)
2  {

```

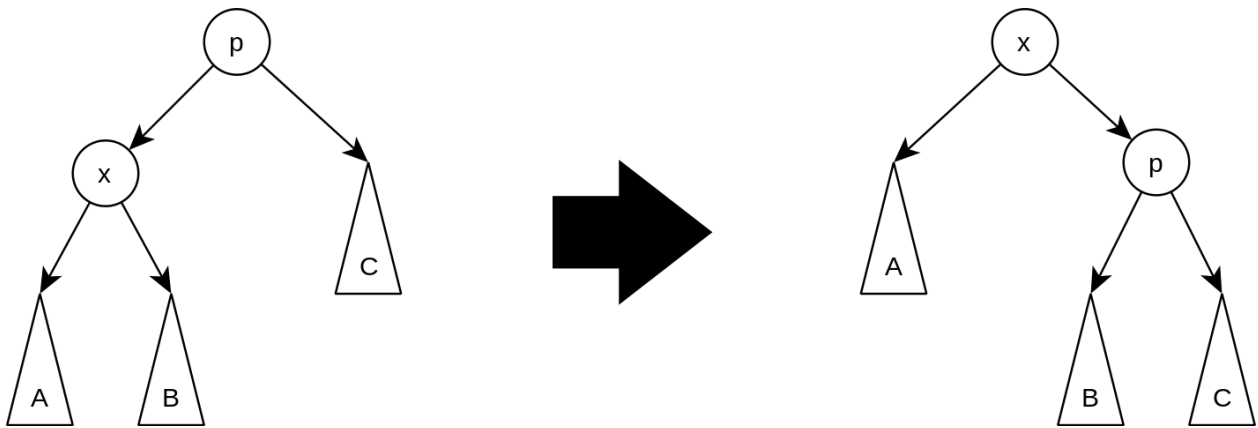
```

3   tree->root = (SplayPointer)malloc(sizeof(SplayNode));
4   tree->root->value = value;
5   tree->root->left = tree->root->right = tree->root->parent = NULL;
6 }
7 else
8 {
9     SplayPointer p = splayInsNode(value, tree->root);
10    tree->root = Splay(p, tree->root);
11 }

```

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.



```

1 SplayPointer Splay(SplayPointer p, SplayPointer root)
2 {
3     if(p is root) return p;        //If p is the root, we need not to splay
4     initialize finishSplay ← 0;    //to mark whether the splay is finished
5     while(!finishSplay)
6     {
7         if(p->parent is root)      //If p is a child of root,
8         {
9             finishSplay ← 1;       //the splay will be finished in the next
↪ step

```

```

10
11     if(p->value < root->value) //if p is the left child
12     {
13         root->left ← p->right; if(p->right)p->right->parent = root;
14         p->right ← root;
15     }
16     else //if p is the right child
17     {
18         root->right ← p->left; if(p->left)p->left->parent = root;
19         p->left ← root;
20     }
21     root->parent ← p;
22     p->parent ← NULL;
23 }
24 else
25 {
26     if(p->parent->parent is root) finishSplay ← 1;
27
28     if(p->value < p->parent->value and p->value <
↪ p->parent->parent->value) ZigZig(p);
29
30     else if(p->value > p->parent->value and p->value <
↪ p->parent->parent->value) ZigZag(p);
31
32     else if(p->value < p->parent->value and p->value >
↪ p->parent->parent->value) ZagZig(p);
33
34     else if(p->value > p->parent->value and p->value >
↪ p->parent->parent->value) ZagZag(p);
35 }
36 }
37 return p;
38 }
39
40 Procedure ZigZig(SplayPointer p)
41 {
42     SplayPointer A ← p, B ← p->parent, C ← B->parent;
43
44     <!--B->left ← A->right; if(A has right child)A->right->parent ← B;
45     C->left ← B->right; if(B has right child)B->right->parent ← C;
46     A->right ← B;

```

```
47     B->right ← C;
48     A->parent ← C->parent;
49     B->parent ← A;
50     C->parent ← B;
51
52     if(A->parent exists)
53     {
54         if(A->parent->value > A->value) A->parent->left ← A;
55         else A->parent->right ← A;
56     }
57 }
```

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.

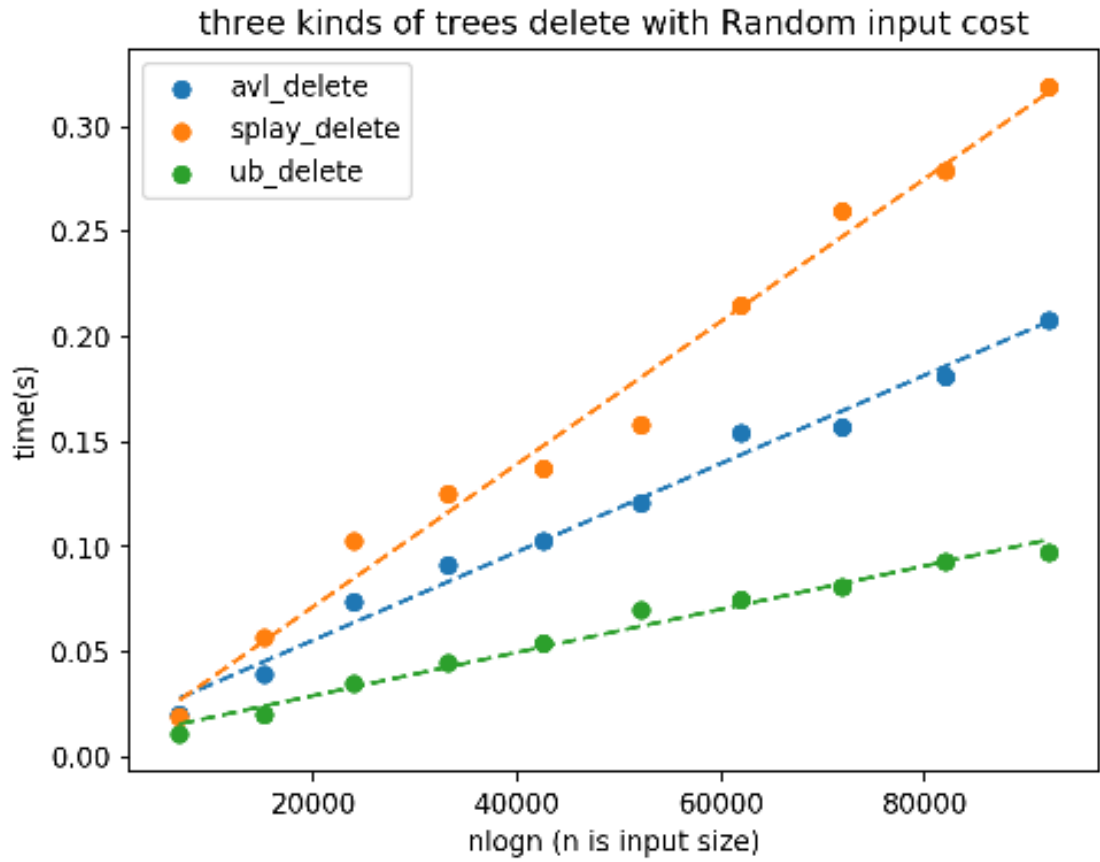
3.2 Test Results

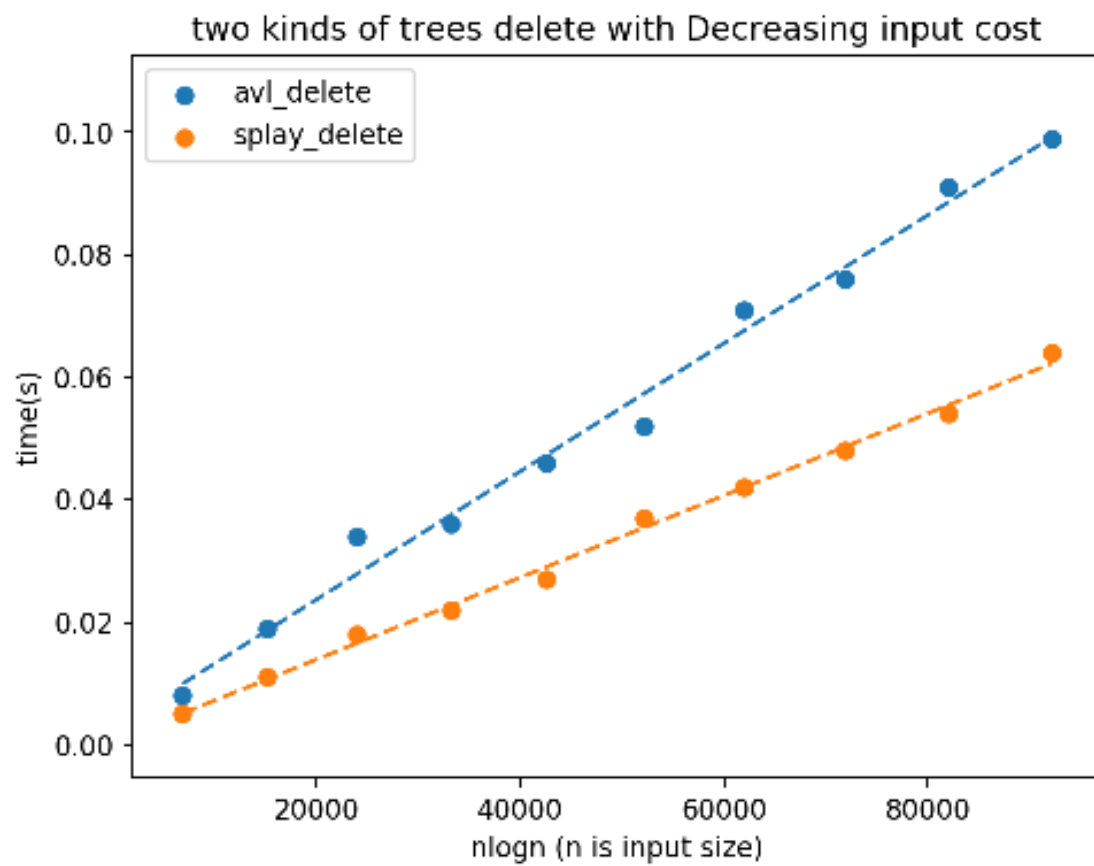
The result table is as followed.

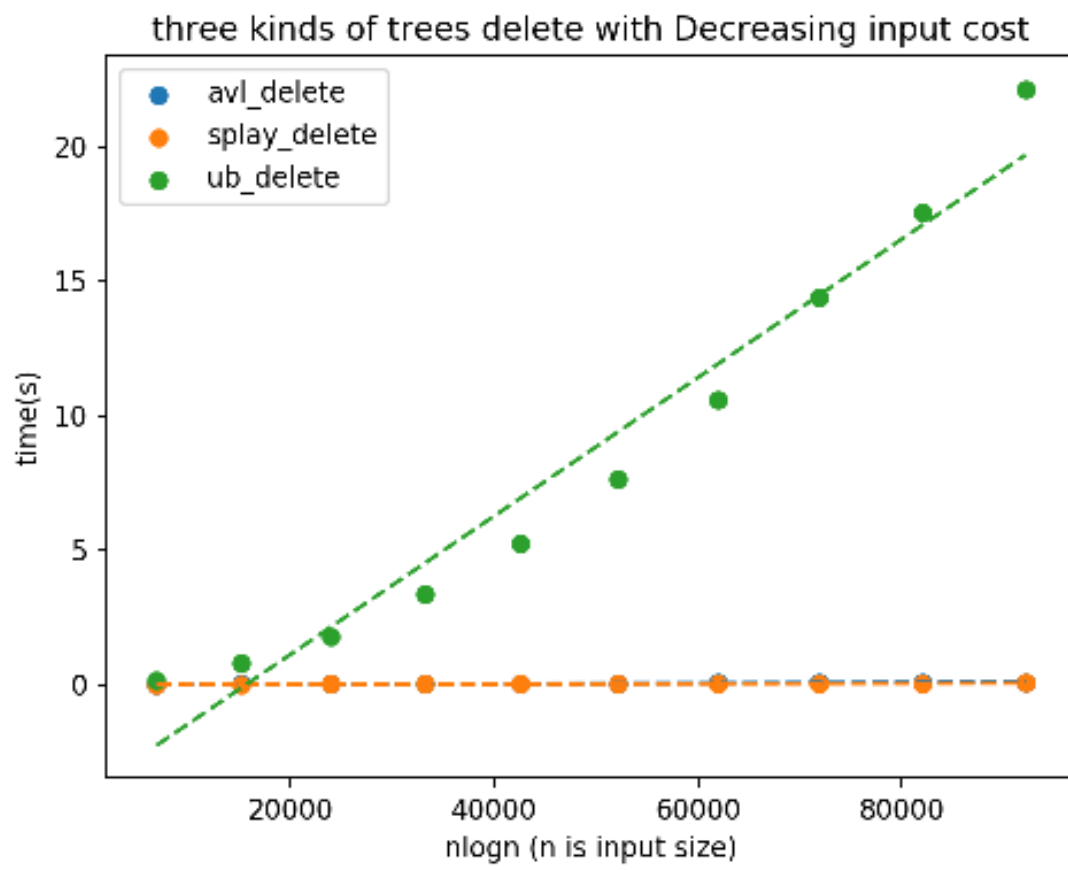
input_size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
avl_insert_increasing	0.007	0.016	0.017	0.032	0.058	0.058	0.066	0.086	0.084	0.095
avl_insert_decreasing	0.007	0.015	0.026	0.039	0.052	0.063	0.065	0.076	0.09	0.1
avl_insert_random	0.013	0.035	0.046	0.055	0.066	0.083	0.102	0.124	0.152	0.158
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

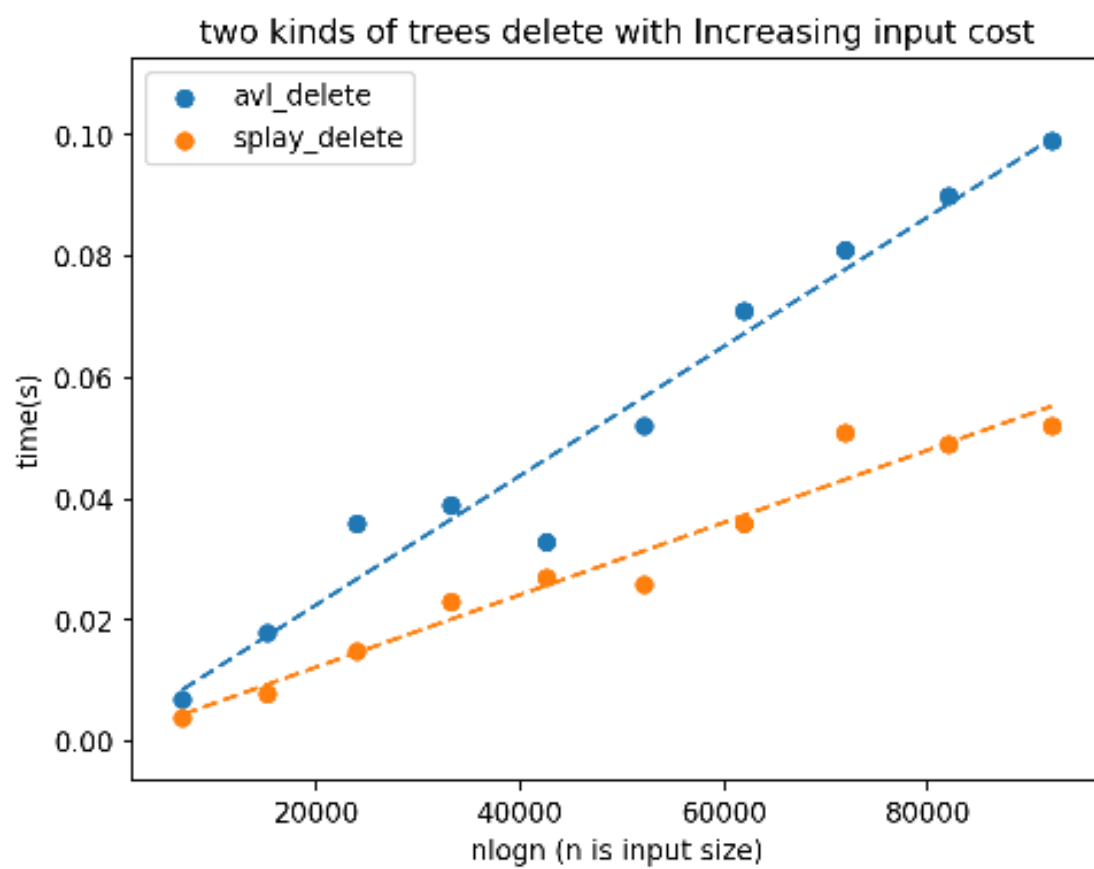
input_size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ub_delete_increasing	0.196	0.802	1.882	3.275	5.273	7.86	10.931	14.173	17.898	22.427
ub_delete_decreasing	0.186	0.813	1.807	3.389	5.269	7.668	10.583	14.401	17.571	22.157
ub_delete_random	0.013	0.014	0.021	0.04	0.049	0.079	0.074	0.09	0.091	0.103

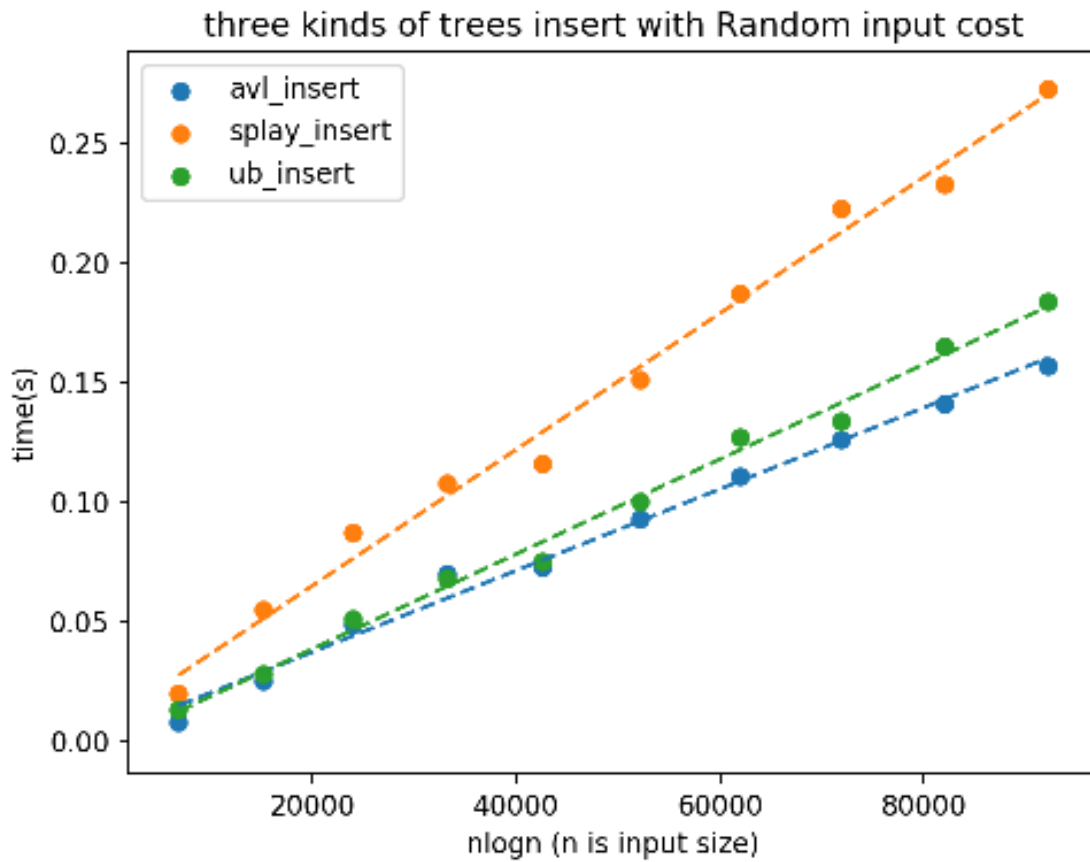
Then we used Python's matplotlib to draw a fitting curve. Because the pictures are too many, we selected some pictures that can present our idea.











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 $O(\log n)$, and the check of ancestors, which is the way back to the root, cost $O(\log n)$ at most. The rotation takes $O(1)$ each time. So the insertion can be completed in $O(\log n)$ time.

2. Deletion

the delete operation always try to find the max in the left and then substitute 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 subtree, then find the min of the right subtree, the rest same. The find process costs $O(\log n)$ and the check with rotation costs $O(\log n)$, for the height is $\log n$, so the delete operation costs $O(\log n)$ 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 subtree 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 node), p (the parent of x) and g (the parent of p) are the nodes affected by the rotation operation.

- Zig:

$$\Delta \Phi = rank'(p) - rank(p) + rank'(x) - rank(x) \text{ [since only } p \text{ and } x \text{ change ranks]} = rank'(p) - rank(x) \text{ [since } rank'(x) = rank(p)] \leq rank'(x) - rank(x) \text{ [since } rank'(p) < rank'(x)]$$

- Zig-Zig step:

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

- Zig-Zag step:

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

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 + $\Delta \Phi \leq 3(\text{rank}'(x) - \text{rank}(x))$

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

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

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 $O(\log n)$ for the height of avl tree is $O(\log n)$. 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

```

1  // AvlTree.c
2  #include "AvlTree.h"
3
4  AvlPointer llrot(AvlPointer A);
5  AvlPointer lrrot(AvlPointer A);
6  AvlPointer rlrot(AvlPointer A);
7  AvlPointer rrrot(AvlPointer A);
8
9  // Build an empty AVL tree
10 AvlTree AVLBuild(void)
11 {
12     AvlTree tree = (AvlTree)malloc(sizeof(AvlTreeNode));
13     tree->root = NULL;
14     return tree;
15 }
16
17 void AVL_Free_Node(AvlPointer node);    //Free the tree (or subtree) whose
    ↳ root is 'node'
18 //Free the whole tree
19 void AVLFree(AvlTree tree)
20 {
21     if (tree) AVL_Free_Node(tree->root);    //If tree is valid, simply use
    ↳ AVL_Free_Node function
22     free(tree);

```

```
23 }
24 void AVL_Free_Node(AvlPointer node)
25 {
26     if (node && node->left) AVL_Free_Node(node->left);
27     if (node && node->right) AVL_Free_Node(node->right);
28     free(node);
29 }
30
31 //Find the node with certain value in the tree
32 AvlPointer AVLFind(int value, AvlTree tree)
33 {
34     AvlPointer p = tree->root;
35     while(p && p->value != value){
36         if(p->value < value)p = p->right;
37         else p = p->left;
38     }
39     return p;
40 }
41
42 //Find the node with minimum value in the subtree whose root is p
43 AvlPointer AVLFindMinNode(AvlPointer p)
44 {
45     while(p && p->left)p = p->left;
46     return p;
47 }
48
49 //Find the node with minimum value in the tree
50 AvlPointer AVLFindMin(AvlTree tree)
51 {
52     return AVLFindMinNode(tree->root);
53 }
54
55 //Find the node with maximum value in the subtree whose root is p
56 AvlPointer AVLFindMaxNode(AvlPointer p)
57 {
58     while(p && p->right)p = p->right;
59     return p;
60 }
61
62 //Find the node with maximum value in the tree
63 AvlPointer AVLFindMax(AvlTree tree)
```

```

64 {
65     return AVLFindMaxNode(tree->root);
66 }
67
68 //Find the node with certain value in the tree whose root is p,
69 //it returns the root of the tree after deletion
70 AvlPointer AVLNodeDel(int value, AvlPointer p)
71 {
72     int lh = 0, rh = 0;
73     //lh means the height of the left subtree of p, while rh means the height
    ↪ of the right one
74
75     if(p->value == value)          //if p is to be deleted
76     {
77         if(!p->left && !p->right)    // if p is a leaf node, then simply free
    ↪ it and assign p NULL
78         {
79             free(p);
80             p = NULL;
81         }
82         else                        // otherwise
83         {
84             if(p->left) //if p has left child, find 'maximum node' in the
    ↪ left subtree and replace p with it
85             {
86                 AvlPointer lcmax = AVLFindMaxNode(p->left);
87                 // by replacing the value of p with its value and delete it
    ↪ in the left subtree of p
88                 p->value = lcmax->value;
89                 p->left = AVLNodeDel(lcmax->value, p->left);
90             }
91             else // or the case is much simpler. we only need to replace p
    ↪ with the right child of p
92             {
93                 AvlPointer rchild = p->right;
94                 free(p);
95                 p = rchild;
96             }
97         }
98     }
99     else

```



```

100     {
101         if(value < p->value)
102         {
103             p->left = AVLNodeDel(value, p->left);
104         }
105         else
106         {
107             p->right = AVLNodeDel(value, p->right);
108         }
109     }
110
111     // Now the deletion is completed in the recursion above, we need to check
112     // the height of two subtrees and do necessary rotations.
113     if(p)
114     {
115         // assign the height of p
116         if(p->left)lh = p->left->height + 1;    else lh = 0;
117         if(p->right)rh = p->right->height + 1;  else rh = 0;
118         p->height = lh > rh ? lh : rh;
119
120         // the following code describes how to rotate in different cases
121         if(lh - rh > 1)
122         {
123             int llh = p->left->left ? p->left->left->height + 1 : 0;
124             int lrh = p->left->right ? p->left->right->height + 1 : 0;
125             if(llh > lrh)p = llrot(p);
126             else p = lrrot(p);
127         }
128         else if(lh - rh < -1)
129         {
130             int rlh = p->right->left ? p->right->left->height + 1 : 0;
131             int rrh = p->right->right ? p->right->right->height + 1 : 0;
132             if(rlh > rrh)p = rlrot(p);
133             else p = rrrot(p);
134         }
135     }
136     return p;
137 }
138
139 // Delete the node with certain value in the tree, if deletion fails (the
↪ node is not found), then return 0

```

```

140 // otherwise return 1
141 int AVLDel(int value, AvlTree tree)
142 {
143     AvlPointer p;
144     int lh = 0, rh = 0;
145
146     p = AVLFind(value, tree);           //find the node to be deleted
147     if(!p) return 0;                   // if not found, return 0
148     else tree->root = AVLNodeDel(value, tree->root); //else use AVLNodeDel
149     ↪ function
150     return 1;
151 }
152
153 //Get the value stored in an AvlPointer
154 int AVLGetValue(AvlPointer p)
155 {
156     return p->value;
157 }
158
159 // Insert a node with value into the subtree whose root is p. It returns the
160 ↪ root of this subtree after insertion.
161 AvlPointer AVLInsEle(int value, AvlPointer p);
162 // Insert a node with value into the tree
163 void AVLIns(int value, AvlTree tree)
164 {
165     if(!tree) return; //if the tree is invalid, do nothing
166
167     if(!tree->root) //if the tree is empty, create a node with value to
168     ↪ be its root
169     {
170         AvlPointer p = (AvlPointer)malloc(sizeof(AvlNode));
171         p->value = value; p->left = p->right = NULL;
172         tree->root = p;
173         p->height = 0;
174     }
175     else //otherwise, use AVLInsEle function
176         tree->root = AVLInsEle(value, tree->root);
177 }
178
179 AvlPointer AVLInsEle(int value, AvlPointer p)

```

```

178 {
179     int lh = 0, rh = 0;
180     // lh means the height of the left subtree of p + 1, while rh means the
↪ height of the right one + 1
181     // the default value 0 means the height of NULL(-1) + 1
182
183     if( !p ){                               // if p is NULL, create a new node
184         p = (AvlPointer)malloc(sizeof(AvlNode));
185         p->value = value;
186         p->left = p->right = NULL;
187         p->height = 0;
188     }else if(value > p->value){              // in this case, insert value to the
↪ right subtree of p
189         p->right = AVLInsEle(value, p->right);
190     }else if(value < p->value){              // in this case, insert value to the left
↪ subtree of p
191         p->left = AVLInsEle(value, p->left);
192     }else{
193         return p;
194     }
195
196     // assign the height of p
197     if(p->left)lh = p->left->height + 1;
198     if(p->right)rh = p->right->height + 1;
199     p->height = lh > rh ? lh : rh;
200
201     // the following code describes how to rotate in different cases
202     if(lh - rh < -1){
203         if(value > p->right->value)
204             p = rrrot(p);
205         else
206             p = rlrot(p);
207     }else if(lh - rh > 1){
208         if(value < p->left->value)
209             p = llrot(p);
210         else
211             p = lrrot(p);
212     }
213
214     return p;
215 }

```

```
216
217 // single left rotation
218 AvlPointer llrot(AvlPointer A){
219     AvlPointer B = A->left;
220
221     A->left = B->right;
222     B->right = A;
223     A->height -= 2;
224
225     return B;
226 }
227
228 // single right rotation
229 AvlPointer rrrot(AvlPointer A){
230     AvlPointer B = A->right;
231
232     A->right = B->left;
233     B->left = A;
234     A->height -= 2;
235
236     return B;
237 }
238
239 // double left right rotation
240 AvlPointer lrrot(AvlPointer A){
241     AvlPointer B = A->left;
242     AvlPointer C = B->right;
243
244     A->left = C->right;
245     C->right = A;
246     B->right = C->left;
247     C->left = B;
248     A->height -= 2;
249     B->height --;
250     C->height ++;
251     return C;
252 }
253
254 // double right left rotation
255 AvlPointer rlrot(AvlPointer A){
256     AvlPointer B = A->right;
```

```

257     AvlPointer C = B->left;
258
259     A->right = C->left;
260     C->left = A;
261     B->left = C->right;
262     C->right = B;
263     A->height -= 2;
264     B->height --;
265     C->height ++;
266     return C;
267 }

```

```

1  /**
2   No Copyright. But if you copy this code, you may be verified as cheating
3   ↪ in ZJU.
4   BE CAREFUL!
5
6   This piece of code defines AVL tree and some algorithms to it.
7   */
8
9  #ifndef AVL_H
10 #define AVL_H
11
12 #include <stdio.h>
13 #include <stdlib.h>
14
15
16 struct _avlnode;
17 typedef struct _avlnode AvlNode;
18 typedef AvlNode* AvlPointer;
19 //=====
20 // AvlPointer is a pointer pointing at a node in the AVL tree.
21 // Users don't need to know the details in this block.
22 // Please use function 'AVLGetValue' to retrieve the value stored in the
23 ↪ node.
24 struct _avlnode
25 {
26     int value;
27     AvlPointer left;
28     AvlPointer right;

```

```

28     int height;
29 };
30 //=====
31 typedef struct _avltree
32 {
33     AvlPointer root;
34 } AvlTreeNode;
35 typedef AvlTreeNode* AvlTree;
36 // Declare 'AvlTree' variable to use the functions followed.
37
38 AvlTree AVLBuild(void);
39 // Build an empty AVL tree and return it
40 // An example to use this function:      AvlTree tree = AVLBuild();
41
42 void AVLIns(int value, AvlTree tree);
43 // Insert a node with its value to be 'value' into the tree
44 // An example to use this function:      AVLIns(0, tree);
45
46 AvlPointer AVLFind(int value, AvlTree tree);
47 AvlPointer AVLFindMin(AvlTree tree);
48 AvlPointer AVLFindMax(AvlTree tree);
49 // Find the node with (certain/minimal/maximal)value in the tree and return
   ↪ its pointer.
50 // Return NULL if not found
51 // Examples to use these functions:      AvlPointer p = AVLFind(0, tree);
   ↪ if(!p)printf("Not found!\n");
52 //                                     AvlPointer min =
   ↪ AVLFindMin(tree);
53 //                                     AvlPointer max =
   ↪ AVLFindMax(tree);
54
55 int AVLDel(int value, AvlTree tree);
56 // Delete the node with certain value in the tree.
57 // Return 1 if the deletion succeeded, and 0 if not (which means the node
   ↪ doesn't exist).
58 // An example to use this function:      int suc = AVLDel(0, tree);
   ↪ if(!suc)printf("Deletion failed!\n");
59
60 void AVLFree(AvlTree tree);
61 // Free the whole tree
62 // An example to use this function:      AVLFree(tree);

```

```

63
64 int AVLGetValue(AvlPointer p);
65 // When you get an AvlPointer, use this function to access the value stored
   ↪ in it.
66 // An example to use this function:           AvlPointer min =
   ↪ AVLFindMin(tree);
67 //                                           if(!min) printf("The tree is
   ↪ empty!\n");
68 //                                           else printf("The minimum value is
   ↪ %d\n", AVLGetValue(min));
69
70 #endif

```

5.1.2 Splay Tree

```

1 // SplayTree.c
2 #include "SplayTree.h"
3
4 // splay p in the subtree whose root is 'root'
5 SplayPointer Splay(SplayPointer p, SplayPointer root);
6
7 // Build an empty splay tree and return it
8 SplayTree splayBuild(void)
9 {
10     SplayTree t = (SplayTree)malloc(sizeof(SplayTreeNode));
11     if(t)t->root = NULL;
12     return t;
13 }
14
15 // Insert a node with value into the subtree whose root is p.
16 // It returns the root of this subtree after insertion.
17 SplayPointer splayInsNode(int value, SplayPointer p);
18 // Insert a node with its value to be 'value' into the tree
19 void splayIns(int value, SplayTree tree)
20 {
21     if(!tree)return;    //if the tree is invalid, do nothing
22
23     if(!tree->root)    //if the tree is empty, create a node with value to
   ↪ be its root
24     {
25         tree->root = (SplayPointer)malloc(sizeof(SplayNode));

```

```

26     tree->root->value = value;
27     tree->root->left = tree->root->right = tree->root->parent = NULL;
28 }
29 else //otherwise, use splayInsNode function, and then
↪ splay p in the whole tree
30 {
31     SplayPointer p = splayInsNode(value, tree->root);
32     tree->root = Splay(p, tree->root);
33 }
34 }
35
36 SplayPointer splayInsNode(int value, SplayPointer p)
37 {
38     while(p && p->value != value) //if p->value == value, there is no need
↪ to insert
39     {
40         if(p->value < value) //search the position in right subtree
41         {
42             if(!p->right) //insert it to the left position of p and
↪ break
43             {
44                 p->right = (SplayPointer)malloc(sizeof(SplayNode));
45                 p->right->value = value;
46                 p->right->parent = p;
47                 p->right->left = p->right->right = NULL;
48                 break;
49             }
50             p = p->right;
51         }
52         else //search the position in left subtree
53         {
54             if(!p->left) //insert it to the left position of p and
↪ break
55             {
56                 p->left = (SplayPointer)malloc(sizeof(SplayNode));
57                 p->left->value = value;
58                 p->left->parent = p;
59                 p->left->left = p->left->right = NULL;
60             }
61             p = p->left;
62         }

```



```

63     }
64     return p;
65 }
66
67 // Find the node with certain value in the tree and return its pointer.
68 SplayPointer splayFind(int value, SplayTree tree)
69 {
70     SplayPointer p = tree->root;
71     // find the node with value
72     while(p && p->value != value)
73     {
74         if(p->value < value)p = p->right;
75         else p = p->left;
76     }
77     // if found and it isn't the root, splay the node, otherwise simply
78     ↪ return the pointer
79     if(p && p != tree->root)tree->root = Splay(p, tree->root);
80     return p;
81 }
82
83 SplayPointer splay_find_min(SplayPointer root)
84 {
85     SplayPointer p = root;
86     while(p && p->left) p = p->left;
87     return p;
88 }
89 // Find the node with minimum value in the tree and return its pointer.
90 SplayPointer splayFindMin(SplayTree tree)
91 {
92     //find the minimum node and splay it
93     SplayPointer p = splay_find_min(tree->root);
94     if(p)tree->root = Splay(p, tree->root);
95     return p;
96 }
97
98 SplayPointer splay_find_max(SplayPointer root)
99 {
100     SplayPointer p = root;
101     while(p && p->right) p = p->right;
102     return p;
103 }

```

```

103 // Find the node with maximum value in the tree and return its pointer.
104 SplayPointer splayFindMax(SplayTree tree)
105 {
106     //find the maximum node and splay it
107     SplayPointer p = splay_find_max(tree->root);
108     if(p) tree->root = Splay(p, tree->root);
109     return p;
110 }
111
112 // Delete the node with certain value in the tree.
113 // Return 1 if the deletion succeeded, and 0 if not (which means the node
114 ↪ doesn't exist).
115 int splayDel(int value, SplayTree tree)
116 {
117     // Find the node. If found, it will be placed at the root of the tree.
118     SplayPointer p = splayFind(value, tree);
119     if(!p) return 0; //If not found, return 0.
120
121     // If p is deleted, the whole tree will be divided into two parts at most
122     SplayPointer new_root = NULL;
123     // If p has both children,
124     if(p->left && p->right)
125     {
126         // find the maximum node in the left subtree and splay it to the root
127         ↪ of the left subtree
128         new_root = splay_find_max(p->left);
129         tree->root = Splay(new_root, p->left);
130         new_root->parent = NULL;
131         // as the new_root is the maximum node in the subtree, it has no
132         ↪ right children
133         // we can simply make the right child of p be the right child of
134         ↪ new_root
135         new_root->right = p->right;
136         if(p->right) p->right->parent = new_root;
137     }
138     // If p has no left child, simply make the right subtree to be the new
139     ↪ tree
140     else if(!p->left)
141     {
142         tree->root = p->right;
143         if(p->right) p->right->parent = NULL;

```

```

139     }
140     // If p has no right child, simply make the left subtree to be the new
141     ↪ tree
142     else
143     {
144         tree->root = p->left;
145         if(p->left)p->left->parent = NULL;
146     }
147     free(p);
148     return 1;
149 }
150
151 //Free the tree (or subtree) whose root is 'node'
152 void splay_Free_Node(SplayPointer node)
153 {
154     if(node && node->left)splay_Free_Node(node->left);
155     if(node && node->right)splay_Free_Node(node->right);
156     free(node);
157 }
158 //Free the whole tree
159 void splayFree(SplayTree tree)
160 {
161     if(tree)splay_Free_Node(tree->root);
162     free(tree);
163 }
164
165 //Get the value stored in a splayPointer
166 int splayGetValue(SplayPointer p)
167 {
168     return p->value;
169 }
170
171 //=====
172 // The four actions using in splay function
173 void ZigZig(SplayPointer p);
174 void ZigZag(SplayPointer p);
175 void ZagZig(SplayPointer p);
176 void ZagZag(SplayPointer p);
177 //=====
178 SplayPointer Splay(SplayPointer p, SplayPointer root)

```

```

179 {
180     if(p == root)return p;           //If p is the root, we needn't to splay
181     int finishSplay = 0;           //to mark whether the splay is finished
182     while(!finishSplay)
183     {
184         if(p->parent == root)       //If p is a child of root,
185         {
186             finishSplay = 1;       //the splay will be finished in the next step
187
188             if(p->value < root->value) //if p is the left child
189             {
190                 root->left = p->right; if(p->right)p->right->parent = root;
191                 p->right = root;
192             }
193             else                               //if p is the right child
194             {
195                 root->right = p->left; if(p->left)p->left->parent = root;
196                 p->left = root;
197             }
198             root->parent = p;
199             p->parent = NULL;
200         }
201         else
202         {
203             if(p->parent->parent == root)finishSplay = 1;
204
205             if(p->value < p->parent->value && p->value <
↪ p->parent->parent->value) ZigZig(p);           //      /
206
207             ↪ //      /
208
209             ↪ //      p
210
211             else if(p->value > p->parent->value && p->value <
↪ p->parent->parent->value) ZigZag(p); //      /
212
213             ↪ //      \
214
215             ↪ //      p
216
217             else if(p->value < p->parent->value && p->value >
↪ p->parent->parent->value) ZagZig(p); //      \

```

```

214 ↪ // /
215
216 ↪ // p
217
218     else if(p->value > p->parent->value && p->value >
219 ↪ p->parent->parent->value) ZagZag(p); // \
220
221 ↪ // \
222
223 ↪ // p
224     }
225 }
226
227 return p;
228 }
229
230 void ZigZig(SplayPointer p)
231 {
232     SplayPointer A = p, B = p->parent, C = B->parent;
233
234     B->left = A->right;    if(A->right)A->right->parent = B;
235     C->left = B->right;    if(B->right)B->right->parent = C;
236     A->right = B;
237     B->right = C;
238     A->parent = C->parent;
239     B->parent = A;
240     C->parent = B;
241
242     if(A->parent)
243     {
244         if(A->parent->value > A->value) A->parent->left = A;
245         else A->parent->right = A;
246     }
247 }
248
249 void ZigZag(SplayPointer p)
250 {
251     SplayPointer A = p, B = p->parent, C = B->parent;
252
253     B->right = A->left;    if(A->left)A->left->parent = B;
254     C->left = A->right;    if(A->right)A->right->parent = C;

```

```

250     A->left = B;
251     A->right = C;
252     A->parent = C->parent;
253     B->parent = A;
254     C->parent = A;
255
256     if (A->parent)
257     {
258         if (A->parent->value > A->value) A->parent->left = A;
259         else A->parent->right = A;
260     }
261 }
262
263 void ZagZig (SplayPointer p)
264 {
265     SplayPointer A = p, B = p->parent, C = B->parent;
266
267     B->left = A->right;    if (A->right) A->right->parent = B;
268     C->right = A->left;    if (A->left) A->left->parent = C;
269     A->right = B;
270     A->left = C;
271     A->parent = C->parent;
272     B->parent = A;
273     C->parent = A;
274
275     if (A->parent)
276     {
277         if (A->parent->value > A->value) A->parent->left = A;
278         else A->parent->right = A;
279     }
280 }
281
282 void ZagZag (SplayPointer p)
283 {
284     SplayPointer A = p, B = p->parent, C = B->parent;
285
286     B->right = A->left;    if (A->left) A->left->parent = B;
287     C->right = B->left;    if (B->left) B->left->parent = C;
288     A->left = B;
289     B->left = C;
290     A->parent = C->parent;

```

```

291     B->parent = A;
292     C->parent = B;
293
294     if(A->parent)
295     {
296         if(A->parent->value > A->value) A->parent->left = A;
297         else A->parent->right = A;
298     }
299 }

```

```

1  /**
2   No Copyright. But if you copy this code, you may be verified as cheating
3   ↪ in ZJU.
4   BE CAREFUL!
5
6   This piece of code defines splay tree and some algorithms to it.
7  */
8  #ifndef SPLAY_H
9  #define SPLAY_H
10
11  #include <stdio.h>
12  #include <stdlib.h>
13
14  typedef struct _splaynode SplayNode;
15  typedef SplayNode* SplayPointer;
16  //=====
17  // SplayPointer is a pointer pointing at a node in the splay tree.
18  // Users don't need to know the details in this block.
19  // Please use function 'SplayGetValue' to retrieve the value stored in the
20  // ↪ node.
21  struct _splaynode{
22      int value;
23      SplayPointer left;
24      SplayPointer right;
25      SplayPointer parent;
26  };
27
28  typedef struct _splaytree
29  {
30      SplayPointer root;

```

```

30 } SplayTreeNode;
31 typedef SplayTreeNode* SplayTree;
32 // Declare 'SplayTree' variable to use the functions followed.
33
34 SplayTree splayBuild(void);
35 // Build an empty splay tree and return it
36 // An example to use this function:      SplayTree tree = splayBuild();
37
38 void splayIns(int value, SplayTree tree);
39 // Insert a node with its value to be 'value' into the tree
40 // An example to use this function:      splayIns(0, tree);
41
42 SplayPointer splayFind(int value, SplayTree tree);
43 SplayPointer splayFindMin(SplayTree tree);
44 SplayPointer splayFindMax(SplayTree tree);
45 // Find the node with (certain/minimal/maximal)value in the tree and return
46   ↪ its pointer.
47 // Return NULL if not found
48 // Examples to use these functions:      SplayPointer p = splayFind(0,
49   ↪ tree);    if(!p)printf("Not found!\n");
50 //
51   ↪ SplayPointer min =
52   ↪ splayFindMin(tree);
53 //
54   ↪ SplayPointer max =
55   ↪ splayFindMax(tree);
56
57 int splayDel(int value, SplayTree tree);
58 // Delete the node with certain value in the tree.
59 // Return 1 if the deletion succeeded, and 0 if not (which means the node
60   ↪ doesn't exist).
61 // An example to use this function:      int suc = splayDel(0, tree);
62   ↪ if(!suc)printf("Deletion failed!\n");
63
64 void splayFree(SplayTree tree);
65 // Free the whole tree
66 // An example to use this function:      splayFree(tree);
67
68 int splayGetValue(SplayPointer p);
69 // When you get an SplayPointer, use this function to access the value stored
70   ↪ in it.
71 // An example to use this function:      SplayPointer min =
72   ↪ splayFindMin(tree);

```



```

63 //                                     if(!min) printf("The tree is
    ↪ empty!\n");
64 //                                     else printf("The minimum value is
    ↪ %d\n", splayGetValue(min));
65
66 #endif

```

5.1.3 UBST

```

1  #include "UBTree.h"
2
3  // Build an empty unbalanced tree and return it
4  UBTree UBBuild(void)
5  {
6      UBTree t = (UBTree)malloc(sizeof(UBTreeNode));
7      t->root = NULL;
8      return t;
9  }
10
11 // Insert a node with value into the tree
12 void UBIns(int value, UBTree tree)
13 {
14     if(!tree) return;    //if the tree is invalid, do nothing
15
16     if(!tree->root)      //if the tree is empty, create a node with value to
    ↪ be its root
17     {
18         tree->root = (UBPointer)malloc(sizeof(UBNode));
19         tree->root->value = value;
20         tree->root->left = tree->root->right = NULL;
21     }
22     else
23     {
24         //search the proper position to insert
25         UBPointer p = tree->root;
26         while(p->value != value)    //if p->value == value, there is no need
    ↪ to insert
27         {
28             if(value < p->value)    //search the position in left subtree
29             {
30                 if(!p->left)        //insert it to the left position of p and
    ↪ break

```

```

30         {
31             p->left = (UBPointer)malloc(sizeof(UBNode));
32             p->left->value = value;
33             p->left->left = p->left->right = NULL;
34             break;
35         }
36         p = p->left;
37     }
38     else //search the position in left subtree
39     {
40         if(!p->right) //insert it to the left position of p and
↪ break
41         {
42             p->right = (UBPointer)malloc(sizeof(UBNode));
43             p->right->value = value;
44             p->right->left = p->right->right = NULL;
45             break;
46         }
47         p = p->right;
48     }
49 }
50 }
51 }
52
53 // Find the node with certain value in the tree and return its pointer.
54 UBPointer UBFind(int value, UBTree tree)
55 {
56     UBPointer p = tree->root;
57     while(p && p->value != value)
58     {
59         if(value < p->value)p = p->left;
60         else p = p->right;
61     }
62     return p;
63 }
64
65 // Find the node with minimum value in the tree and return its pointer.
66 UBPointer UBFindMin(UBTree tree)
67 {
68     UBPointer p = tree->root;
69     while(p && p->left)p = p->left;

```

```

70     return p;
71 }
72
73 // Find the node with maximum value in the tree and return its pointer.
74 UBPointer UBFindMax(UBTree tree)
75 {
76     UBPointer p = tree->root;
77     while(p && p->right)p = p->right;
78     return p;
79 }
80
81 //Find the node with certain value in the tree whose root is p,
82 //it returns the root of the tree after deletion
83 UBPointer UBNodeDel(int value, UBPointer p)
84 {
85     if(value < p->value) p->left = UBNodeDel(value, p->left);
86     ↪ //delete from left subtree
87     else if(value > p->value) p->right = UBNodeDel(value, p->right);
88     ↪ //delete from right subtree
89     else
90     {
91         ↪ //delete p
92         if(!p->left && !p->right) //if p is a leaf, simply free
93         ↪ it
94         {
95             free(p);
96             p = NULL;
97         }
98         else if(p->left && p->right) //if p has both children
99         { //swap p with its left child,
100             ↪ delete from the left subtree
101             p->value = p->left->value;
102             p->left->value = value;
103             p->left = UBNodeDel(value, p->left);
104         }
105         else if(!p->left) //if p doesn't has left
106         ↪ child, return the right child of p
107         { //and free p
108             UBPointer child = p->right;
109             free(p);
110             p = child;

```

```

105         }
106         else                                     //if p doesn't has right
↪ child, return the left child of p
107         {                                       //and free p
108             UBPointer child = p->left;
109             free(p);
110             p = child;
111         }
112     }
113     return p;
114 }
115
116 // Delete the node with certain value in the tree, if deletion fails (the
↪ node is not found), then return 0
117 // otherwise return 1
118 int UBDel(int value, UBTree tree)
119 {
120     UBPointer p = UBFind(value, tree);          //find the node to be deleted
121     if(!p) return 0;                            // if not found, return 0
122     else tree->root = UBNodeDel(value, tree->root); //else use UBNodeDel
↪ function
123     return 1;
124 }
125
126 //Free the tree (or subtree) whose root is p
127 void UBFreeNode(UBPointer p);
128 //Free the whole tree
129 void UBFree(UBTree tree)
130 {
131     if(tree) UBFreeNode(tree->root);
132     free(tree);
133 }
134 void UBFreeNode(UBPointer p)
135 {
136     if(p && p->left) UBFreeNode(p->left);
137     if(p && p->right) UBFreeNode(p->right);
138     free(p);
139 }
140
141 //Get the value stored in a UBPointer
142 int UBGetValue(UBPointer p)

```

```

143 {
144     return p->value;
145 }

```

```

1  /**
2      No Copyright. But if you copy this code, you may be verified as cheating
3      ↪ in ZJU.
4      BE CAREFUL!
5
6      This piece of code defines simple binary search tree (unbalanced tree)
7      ↪ and some algorithms to it.
8  */
9
10 #ifndef UBT_H
11 #define UBT_H
12
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 typedef struct _ubnode UBNode;
17 typedef UBNode* UBPointer;
18 //=====
19 // UBPointer is a pointer pointing at a node in the unbalanced tree.
20 // Users don't need to know the details in this block.
21 // Please use function 'UBGetValue' to retrieve the value stored in the node.
22 struct _ubnode
23 {
24     int value;
25     UBPointer left, right;
26 };
27 //=====
28
29 typedef struct _ubtreenode
30 {
31     UBPointer root;
32 } UBTreeNode;
33 typedef UBTreeNode* UBTree;
34 // Declare 'UBTree' variable to use the functions followed.
35
36 UBTree UBBuild(void);
37 // Build an empty unbalanced tree and return it

```

```

36 // An example to use this function:          UBTree tree = UBBuild();
37
38 void UBIns(int value, UBTree tree);
39 // Insert a node with its value to be 'value' into the tree
40 // An example to use this function:          UBIns(0, tree);
41
42 UBPointer UBFind(int value, UBTree tree);
43 UBPointer UBFindMin(UBTree tree);
44 UBPointer UBFindMax(UBTree tree);
45 // Find the node with (certain/minimal/maximal)value in the tree and return
   ↪ its pointer.
46 // Return NULL if not found
47 // Examples to use these functions:          UBPointer p = UBFind(0, tree);
   ↪ if(!p)printf("Not found!\n");
48 //
   ↪ UBPointer min = UBFindMin(tree);
49 //
   ↪ UBPointer max = UBFindMax(tree);
50
51 int UBDel(int value, UBTree tree);
52 // Delete the node with certain value in the tree.
53 // Return 1 if the deletion succeeded, and 0 if not (which means the node
   ↪ doesn't exist).
54 // An example to use this function:          int suc = UBDel(0, tree);
   ↪ if(!suc)printf("Deletion failed!\n");
55
56 void UBFree(UBTree tree);
57 // Free the whole tree
58 // An example to use this function:          UBFree(tree);
59
60 int UBGetValue(UBPointer p);
61 // When you get an UBPointer, use this function to access the value stored in
   ↪ it.
62 // An example to use this function:          UBPointer min = UBFindMin(tree);
63 //
   ↪ if(!min) printf("The tree is
64 //
   ↪ else printf("The minimum value is
   ↪ %d\n", UBGetValue(min));
65
66 #endif

```

5.1.4 Main

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include "AvlTree.h"
5  #include "SplayTree.h"
6  #include "UBTree.h"
7
8  clock_t start, stop;
9  double duration;
10
11 int *ins, *del;
12
13 int main()
14 {
15     int n;
16     AvlTree a = AVLBuild();
17     SplayTree s = splayBuild();
18     UBTree u = UBBuild();
19
20     scanf("%d", &n);
21     ins = (int*)malloc(sizeof(int) * n);
22     del = (int*)malloc(sizeof(int) * n);
23     if(!ins || !del)
24     {
25         printf("Error when allocating space in memory!\n");
26         exit(1);
27     }
28
29     for(int ri = 0; ri < 2 * n; ri++)
30     {
31         if(ri < n)scanf("%d", ins + ri);
32         else scanf("%d", del + ri - n);
33     }
34
35     start = clock();
36     for(int ri = 0; ri < n; ri++)AVLIns(ins[ri], a);
37     stop = clock();
38     duration = ((double)(stop - start)) / CLK_TCK;
39     printf("%f ", duration);
40     start = clock();

```

```

41     for(int ri = 0; ri < n; ri++)AVLDel(del[ri], a);
42     stop = clock();
43     duration = ((double)(stop - start)) / CLK_TCK;
44     printf("%f\n", duration);
45
46     start = clock();
47     for(int ri = 0; ri < n; ri++)splayIns(ins[ri], s);
48     stop = clock();
49     duration = ((double)(stop - start)) / CLK_TCK;
50     printf("%f ", duration);
51     start = clock();
52     for(int ri = 0; ri < n; ri++)splayDel(del[ri], s);
53     stop = clock();
54     duration = ((double)(stop - start)) / CLK_TCK;
55     printf("%f\n", duration);
56
57     start = clock();
58     for(int ri = 0; ri < n; ri++)UBIns(ins[ri], u);
59     stop = clock();
60     duration = ((double)(stop - start)) / CLK_TCK;
61     printf("%f ", duration);
62     start = clock();
63     for(int ri = 0; ri < n; ri++)UBDel(del[ri], u);
64     stop = clock();
65     duration = ((double)(stop - start)) / CLK_TCK;
66     printf("%f\n", duration);
67
68     AVLFree(a);
69     splayFree(s);
70     UBFree(u);
71
72     return 0;
73 }

```

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