Project 1

Binary Search Trees

Group 17

Date:2022-03-02

Chapter 1: Introduction

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. And a splay tree is a self-adjusting binary search tree that moves a node to the root through a series of rotations along a path from a node to the tree root.
The unbalanced binary search trees, AVL trees, and splay trees may perform differently when insertions and deletions happened. In the project, we are to implerment operations on these three trees and compare and analyze their performances under different order of insertions and deletions.

Chapter 2: Algorithm Specification

Main Program

```
For j = 1000 To 10000 Step 200
        Output Opr_BST(1,j,seq);
        Output Opr_SPL(1,j,seq);
        Output Opr_AVL(1,j,seq);
Next j
```

Binary Search Tree

```
Procedure insert(x:integer)
    int nxt := (now->val < x);
    while now->son[nxt] != nullptr  do
    {
            now := now->son[nxt];
            nxt := (now->val < x);
    }
    now->son[nxt] := new node(x, now);

Procedure erase(x:integer,now:node*)
    while now->val != x do
    {
        now := now->son[now->val < x];
```

```
    }
    if now->son[0] == nullptr then
        connect(now->son[1], now->father, sonn(now));
    else if now->son[1] == nullptr then
        connect(now->son[0], now->father, sonn(now));
    else
    {
        node* alt := _extreme(now->son[1], 0);
        connect(now->son[0], alt, 0);
        if (alt->father != now) then
        {
            connect(alt->son[1], alt->father, 0);
            connect(now->son[1], alt, 1);
        }
        connect(alt, now->father, sonn(now));
    }
    delete now;
```

AVL-Tree

```
Procedure insert(X:integer,AVLtree & A)
    if A==NULL then
    {
        A:=(AVLtree)malloc(sizeof(struct AVLNode));
        if A == NULL then
        {
            output ("ERROR! Out of space!!!");
        }
        else
        {
            A->data := X;
            A->Left := NULL;
            A->Right := NULL;
            A->height := 0;
        }
    }
    else if (X < A->data) then
    {
        A->Left := AVL_Insert(X, A->Left);
        if AVL_GetHeight(A->Left) - AVL_GetHeight(A->Right) == 2 then
        {
```

```
            if X < A->Left->data then
                A := AVL_SingleRotateWithLeft(A);
            else A := AVL_DoubleRotateWithLeft(A);
        }
    }
    else if X > A->data then
    {
        A->Right := AVL_Insert(X, A->Right);
        if AVL_GetHeight(A->Right) - AVL_GetHeight(A->Left) == 2 then
        {
            if X > A->Right->data then
                A := AVL_SingleRotateWithRight(A);
            else A := AVL_DoubleRotateWithRight(A);
        }
    }
    A->height := Max(AVL_GetHeight(A->Left), AVL_GetHeight(A->Right)) + 1;
Procedure deletion(X:integer)
        if X < A->data then
        {
            A->Left := AVL_Delete(A->Left, X);
            if abs(AVL_GetHeight(A->Right) - AVL_GetHeight(A->Left)) == 2 the
            {
                AVLtree P := A->Right;
                if AVL_GetHeight(P->Left) > AVL_GetHeight(P->Right) then
                {
                    A := AVL_DoubleRotateWithRight(A);
                }
                else
                {
                    A := AVL_SingleRotateWithRight(A);
                }
            }
        }
        else if X > A->data then
        {
            A->Right := AVL_Delete(A->Right, X);
            if abs(AVL_GetHeight(A->Left) - AVL_GetHeight(A->Right)) == 2 the
            {
                AVLtree P := A->Left;
                if AVL_GetHeight(P->Right) > AVL_GetHeight(P->Left) then
                {
                    A := AVL_DoubleRotateWithLeft(A);
                }
```

```
                else
                {
                    A ;= AVL_SingleRotateWithLeft(A);
                }
            }
        }
        else if X == A->data then
        {
            if A->Left && A->Right then
            {
                if AVL_GetHeight(A->Left) > AVL_GetHeight(A->Right) then
                {
                    AVLtree max := AVL_getMaxNum(A->Left);
                    A->data := max->data;
                    A->Left := AVL_Delete(A->Left, max->data);
                }
                else
                {
                    AVLtree min := AVL_getMinNum(A->Right);
                    A->data := min->data;
                    A->Right := AVL_Delete(A->Right, min->data);
                }
            }
            else
            {
                A := A->Left ? A->Left : A->Right;
            }
        }
        A->height := Max(AVL_GetHeight(A->Left), AVL_GetHeight(A->Right)) + 1
```

Splay Tree

```
Procedure add(splaytree,root,element:integer)
    if root=0 then
    {
        root := (SplayNode*)malloc(sizeof(SplayNode));
        if root=0 then
        {
            output ("init splay node error.");
        }
```

```
        root->left := root->right := NULL;
        root->element := element;
        splaytree->size++;
    }
    else if root->element > element then
    {
        root->left := add_node(splaytree, root->left, element);
        if splaytree->root == root && root->left->element == element then
            root := rotate_left_single(root);
        else ifroot->left->left && root->left->left->element == element then
            root := rotate_zig_zig_left(root);
        else ifroot->left->right && root->left->right->element == element the
            root := rotate_left_double(root);
    }
    else if root->element < element then
    {
        root->right := add_node(splaytree, root->right, element);
        if splaytree->root == root && root->right->element == element then
            root := rotate_right_single(root);
        else if root->right->right && root->right->right->element == element
            root := rotate_zig_zig_right(root);
        else if root->right->left && root->right->left->element == element th
            root ;= rotate_right_double(root);

    }
    else
    {
        return root;
    }
Function delete(splaytree,id:integer)
    if splaytree == NULL || splaytree->size == 0 then
        return 0;
    SplayNode *root := node_rotate_to_root(splaytree->root, splaytree->root,
    if root=0 then
        return 0;
    SplayNode *left := root->left;
    SplayNode *right = root->right;
    free(root);
    if left!=0 then
    {
        SplayNode *max := find_max(left);
        SplayNode *left_node := node_rotate_to_root(left, left, max->element)
        left_node->right := right;
```

```
52          splaytree->root := left_node;
53      }
54      else
55          splaytree->root := right;
56      splaytree->size--;
57      return 1;
```

Chapter 3: Testing Results

To measure and compare the performances of unbalanced binary search trees, AVL trees, and splay trees, especially the performance of the time they use to run, we use the sizes of input` N = 1000, 1200, 1400, ..., 9800, 10000` to test them. Also, there are three different ways of the test on a set of N distinct integers as:

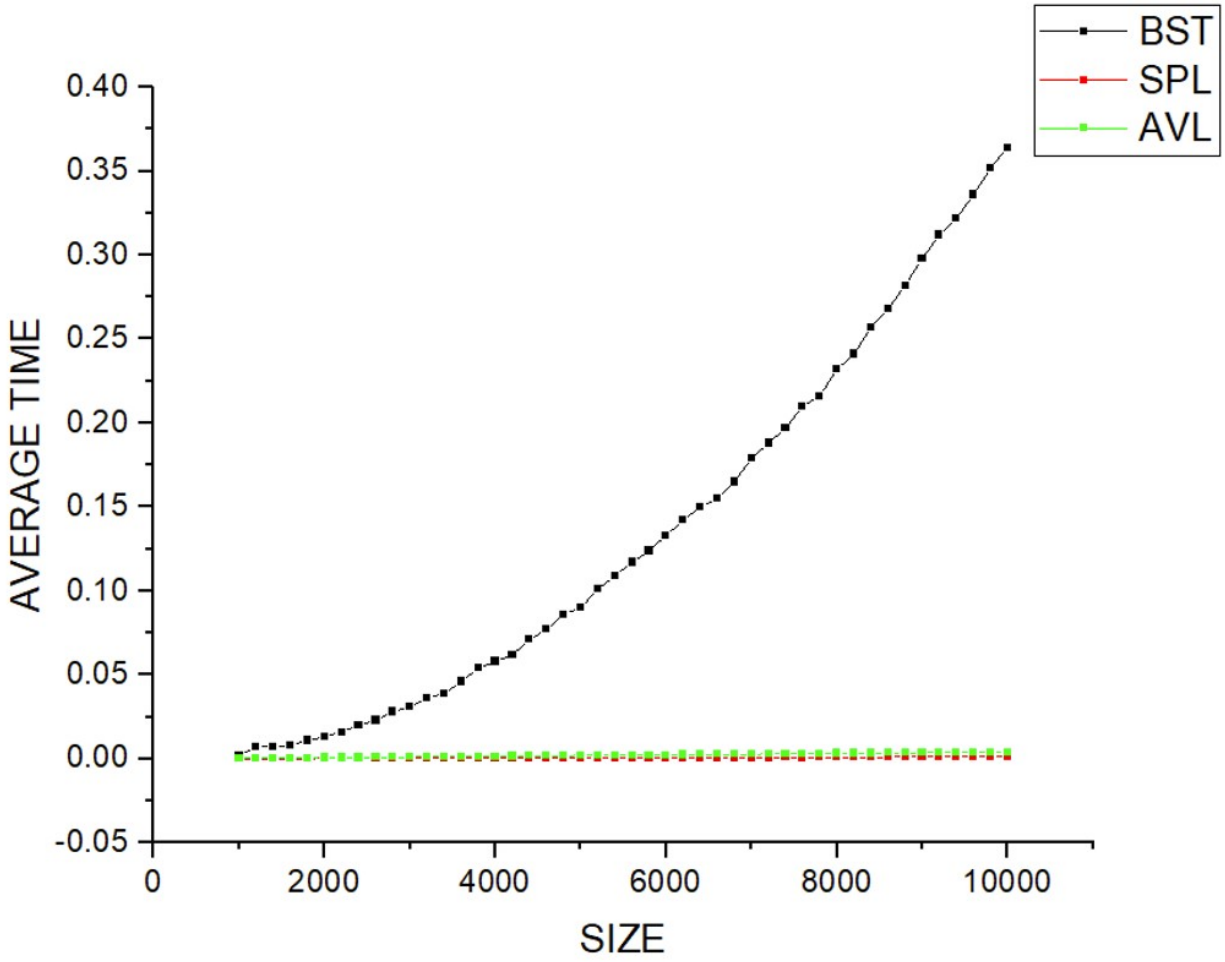
- (1) Insert N integers in increasing order and delete them in the same order; (Data type 1)
- (2) Insert N integers in increasing order and delete them in the reverse order; (Data type 2)
- (3) Insert N integers in random order and delete them in random order. (Data type 3)
- And in order to illustrate the difference, the Binary Search Tree will run only once, though the other two will run for 50 times. The average time they cost would be taken as the result.

Testing Result

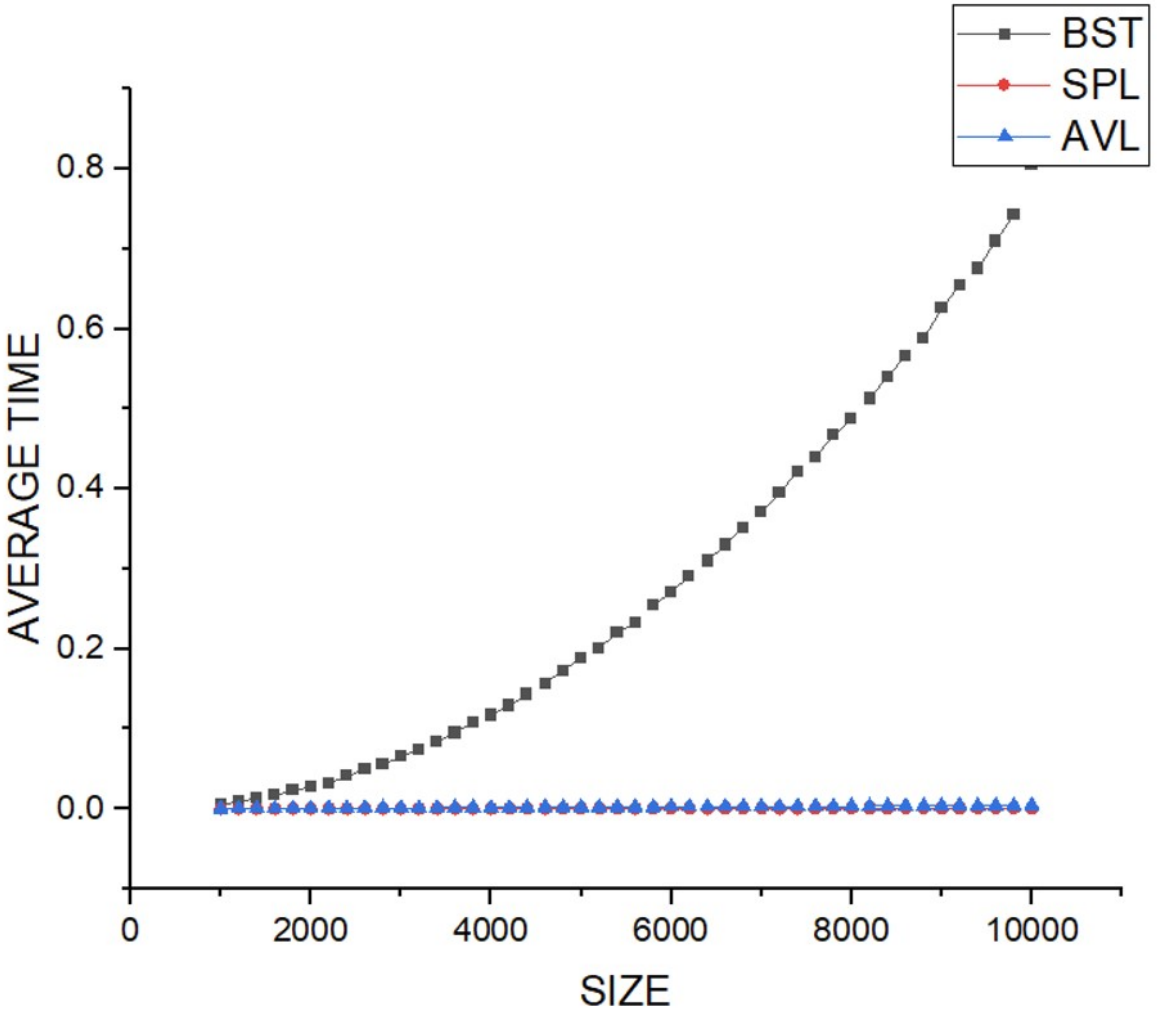
SIZE	Data Type 1			Data Type 2			Data Type 3		
	BST	SPLAY	AVL	BST	SPLAY	AVL	BST	SPLAY	AVL
1000	0.002	1.20E-04	3.20E-04	0.005	8.00E-05	3.00E-04	0	5.00E-04	4.80E-04
1200	0.007	2.20E-04	4.80E-04	0.009	1.00E-04	3.60E-04	0	6.00E-04	5.80E-04
1400	0.007	1.80E-04	4.40E-04	0.013	1.00E-04	4.40E-04	0	7.00E-04	7.00E-04
1600	0.008	2.00E-04	5.20E-04	0.017	1.20E-04	5.00E-04	0	8.60E-04	8.20E-04
1800	0.011	2.20E-04	6.00E-04	0.023	1.40E-04	5.80E-04	0	0.00102	9.40E-04
2000	0.013	2.40E-04	6.60E-04	0.027	1.80E-04	6.60E-04	0	0.00114	0.00108
2200	0.016	2.40E-04	7.40E-04	0.032	1.80E-04	7.40E-04	0.001	0.00126	0.00118
2400	0.02	3.00E-04	8.20E-04	0.041	1.80E-04	8.00E-04	0.001	0.0014	0.00132
2600	0.023	3.20E-04	9.20E-04	0.049	2.00E-04	9.00E-04	0.001	0.00154	0.00146
2800	0.028	3.40E-04	9.40E-04	0.056	2.20E-04	9.80E-04	0.001	0.00172	0.00158
3000	0.031	3.40E-04	0.00108	0.065	2.40E-04	0.00104	0.001	0.00186	0.00176
3200	0.036	3.60E-04	0.00114	0.074	2.60E-04	0.00114	0.001	0.00202	0.00186
3400	0.039	4.00E-04	0.0012	0.084	2.40E-04	0.00122	0.001	0.00216	0.00196
3600	0.046	4.40E-04	0.00138	0.095	2.80E-04	0.00128	0.001	0.00232	0.00214
3800	0.054	4.80E-04	0.0014	0.108	3.00E-04	0.00138	0.001	0.00252	0.00224
4000	0.058	4.80E-04	0.00146	0.117	3.00E-04	0.00146	0.001	0.0027	0.00248
4200	0.062	5.20E-04	0.00158	0.129	3.20E-04	0.00152	0.001	0.00284	0.00254
4400	0.071	5.60E-04	0.00172	0.143	3.20E-04	0.00162	0.001	0.00302	0.00268
4600	0.077	5.60E-04	0.00176	0.157	3.60E-04	0.00168	0.001	0.00316	0.00284
4800	0.086	5.80E-04	0.0018	0.173	3.60E-04	0.00178	0.002	0.0033	0.00294
5000	0.09	5.80E-04	0.00192	0.189	3.80E-04	0.00184	0.002	0.00346	0.00318
5200	0.101	6.60E-04	0.00202	0.201	4.00E-04	0.00194	0.002	0.00364	0.0033
5400	0.109	6.40E-04	0.00208	0.22	4.20E-04	0.00202	0.002	0.00378	0.00344
5600	0.117	6.60E-04	0.00216	0.233	4.40E-04	0.00212	0.002	0.00404	0.0036
5800	0.124	7.00E-04	0.00222	0.254	4.40E-04	0.00228	0.002	0.00406	0.00372
6000	0.133	7.20E-04	0.00228	0.271	5.00E-04	0.00234	0.002	0.00426	0.00378
6200	0.142	7.40E-04	0.0024	0.291	4.60E-04	0.00238	0.002	0.0045	0.00402
6400	0.15	7.40E-04	0.00254	0.31	4.80E-04	0.00244	0.002	0.00462	0.00408
6600	0.155	7.80E-04	0.00254	0.33	5.00E-04	0.00256	0.002	0.00482	0.0044
6800	0.165	7.80E-04	0.00266	0.352	4.80E-04	0.0027	0.002	0.00494	0.0045
7000	0.179	8.40E-04	0.00274	0.372	5.40E-04	0.0027	0.003	0.00516	0.00464
7200	0.188	8.40E-04	0.0028	0.395	5.60E-04	0.00282	0.003	0.00532	0.00476
7400	0.197	8.80E-04	0.00288	0.422	5.60E-04	0.0029	0.002	0.00548	0.0049
7600	0.21	8.60E-04	0.00304	0.44	5.60E-04	0.00298	0.003	0.00566	0.0051
7800	0.216	9.20E-04	0.0031	0.467	5.80E-04	0.00308	0.002	0.0058	0.00506
8000	0.232	9.80E-04	0.0032	0.488	6.00E-04	0.00322	0.003	0.00598	0.00542
8200	0.241	9.60E-04	0.00328	0.513	6.20E-04	0.00326	0.003	0.00626	0.00564
8400	0.257	9.60E-04	0.00334	0.541	6.40E-04	0.00338	0.003	0.00634	0.00568
8600	0.268	0.001	0.00344	0.566	6.40E-04	0.00346	0.003	0.00658	0.00576
8800	0.282	0.00102	0.00352	0.588	6.80E-04	0.00358	0.003	0.00668	0.00604
9000	0.298	0.00104	0.0036	0.626	6.60E-04	0.00368	0.003	0.00684	0.00622
9200	0.312	0.00106	0.00368	0.655	7.20E-04	0.0038	0.003	0.007	0.0062
9400	0.322	0.0011	0.00384	0.676	7.00E-04	0.00388	0.003	0.00718	0.00638
9600	0.336	0.0011	0.0039	0.71	7.20E-04	0.00396	0.003	0.00736	0.00668
9800	0.352	0.00112	0.00396	0.743	7.00E-04	0.00406	0.003	0.00758	0.00688
10000	0.364	0.00116	0.00404	0.806	7.60E-04	0.00422	0.003	0.00772	0.00702

Here follows some useful data comparation in line charts.

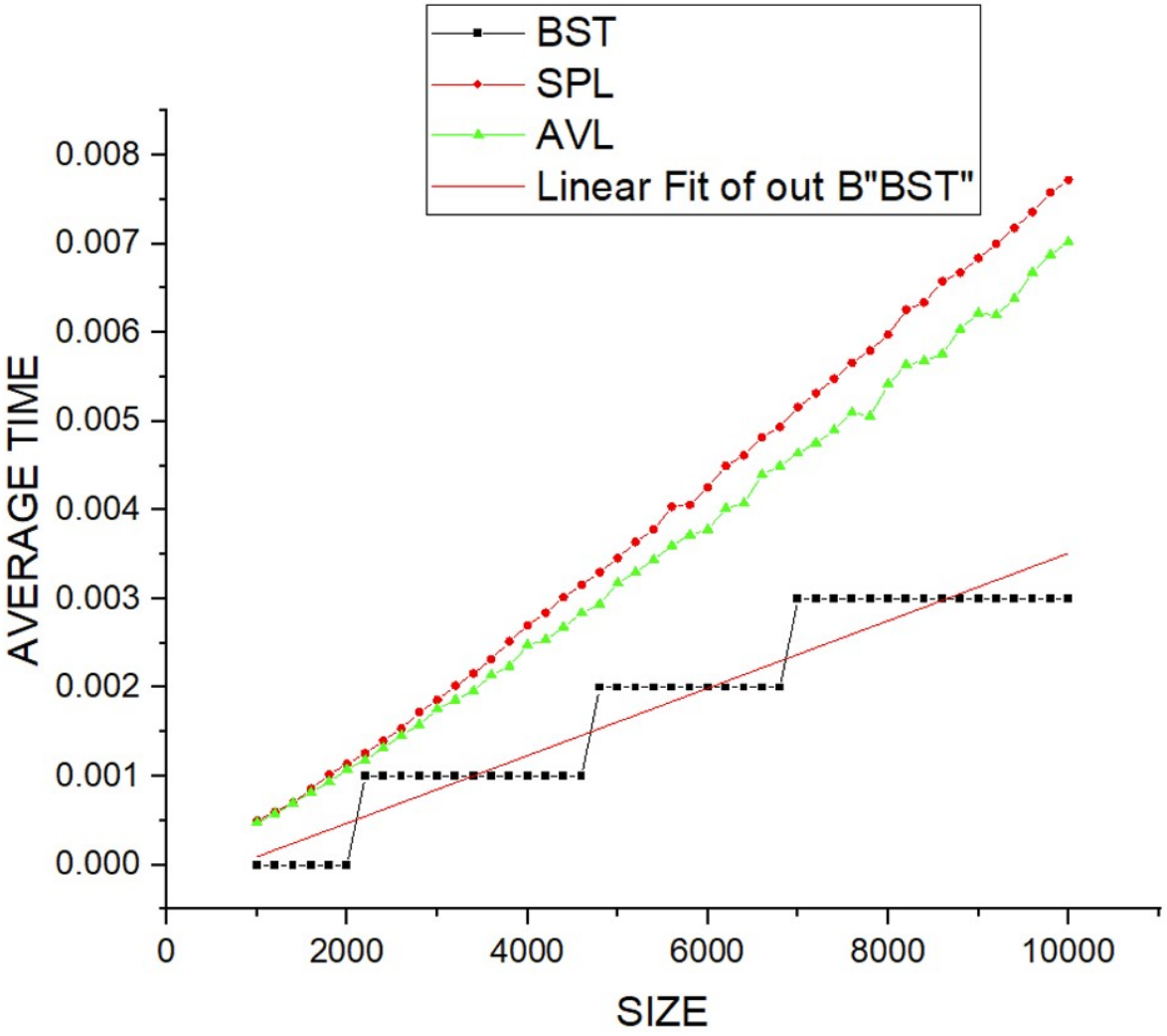
Compare the performances of unbalanced binary search trees, AVL trees, and splay trees using data type 1:



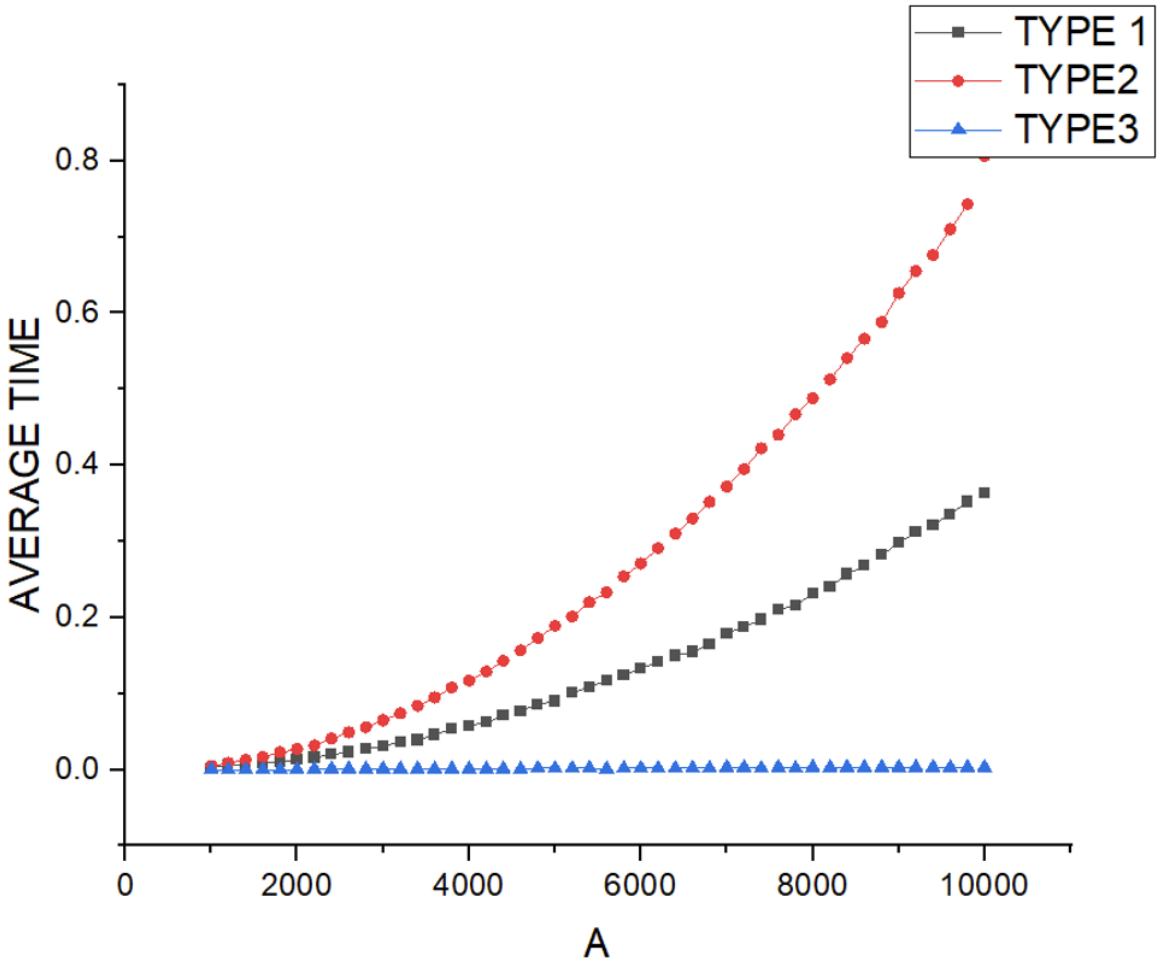
Compare the performances of unbalanced binary search trees, AVL trees, and splay trees using data type 2:



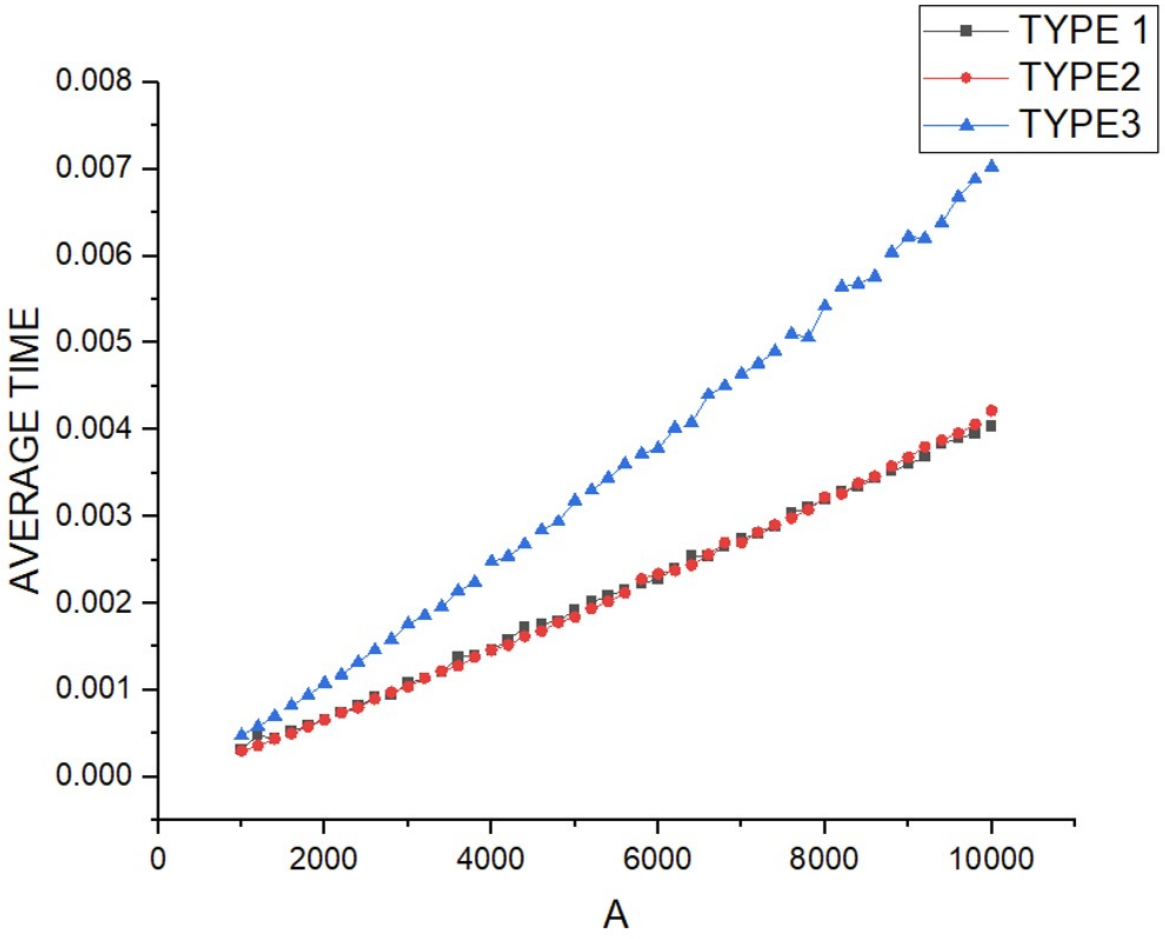
Compare the performances of unbalanced binary search trees, AVL trees, and splay trees using data type 3:



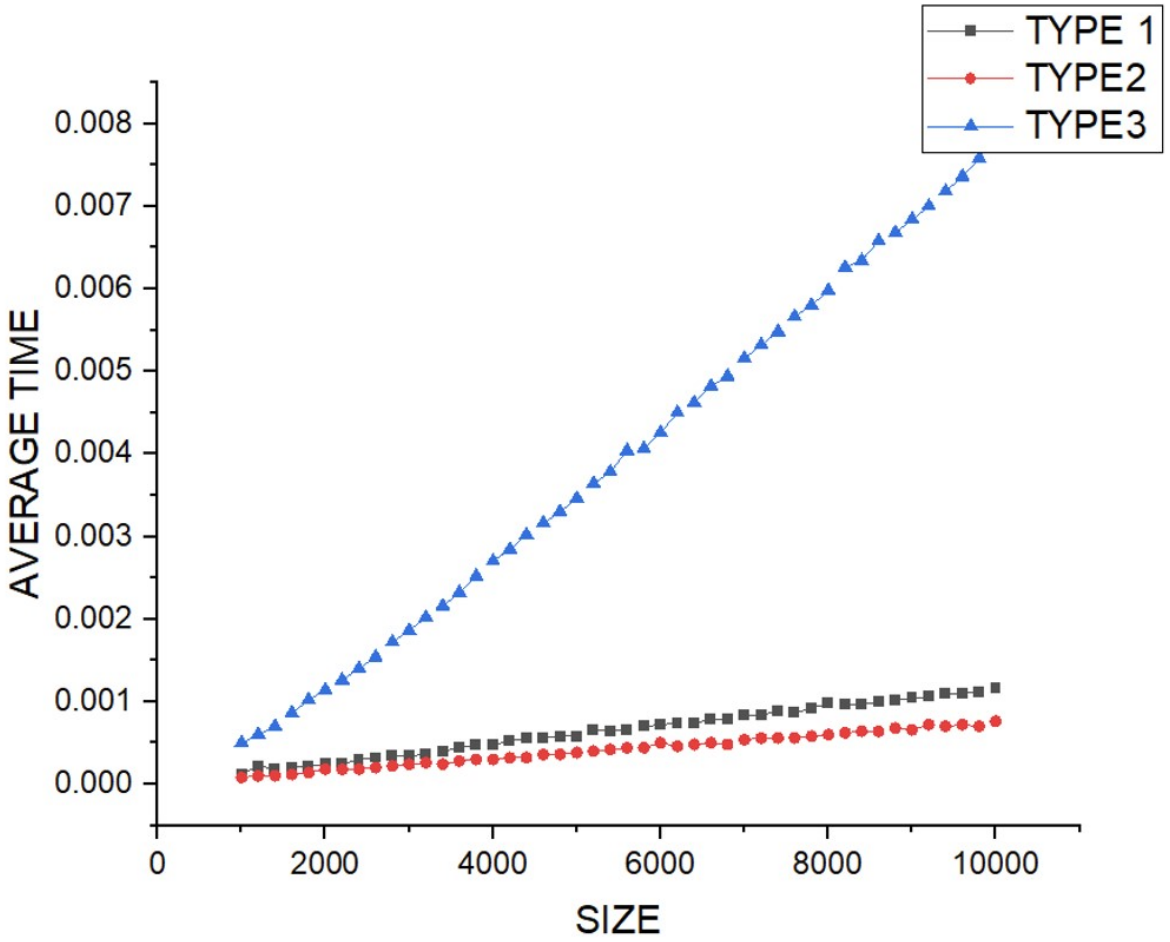
Compare the performances of the Binary Search Tree using different data type:



Compare the performances of the AVL-Tree using different data type:



Compare the performances of the Splay Tree using different data type:



Chapter 4: Analysis and Comments

Complexity Analysis

Space Complexity:

- Apparently, the space needed in all three trees is $O(n)$, for pointers of nodes and information in these nodes.

Time Complexity:

- We will analyse by different data types first. Then we will measure each kind of tree's performance on different data type.

Data Type 1:

- For BST, the time complexity is $O(n^2)$, as for insertion the needed operations are already $1+2+3+...+n = \frac{n*(n+1)}{2}$, and for deletion it's $n\,O(1)$ operations, which becomes $O(n)$.
- For AVL, the time complexity is $O(nlogn)$, as the insertion needs n operation on a $logn$ heighted AVL-tree, and the deletion is also $O(nlogn)$ for the same reason.
- For SPLAY, the time complexity is $O(nlogn)$, as the insertion needs n operation which is $O(1)$ (one insertion and one rotation) which will be $O(n)$, and the deletion is no more than $O(nlogn)$, as in class, we already proved it by using Amortized Analysis.
- As a result, in Graph 1 we see BST very slow and its data points form a parabola, and AVL & SPLAY are both fast and have similar performance.

Data Type 2:

- For BST, the time complexity is $O(n^2)$, as for insertion and deletion the needed operations are both $1+2+3+...+n = \frac{n*(n+1)}{2}$.
- For AVL, the time complexity is $O(nlogn)$, as the insertion needs n operation on a $logn$ heighted AVL-tree, and the deletion is also $O(nlogn)$ for the same reason.
- For SPLAY, the time complexity is $O(nlogn)$, as the insertion needs n operation which is $O(1)$ (one insertion and one rotation) which will be $O(n)$, and the deletion is no more than $O(nlogn)$, as in class, we already proved it by using Amortized Analysis.

- As a result, in Graph 2 we see BST very slow and its data points form a parabola, and AVL & SPLAY are both fast and have similar performance, just like Data Type 1.

Data Type 3:

- For BST, the time complexity is $O(nlogn)$, as the data is highly randomized, we can expect the average tree height $O(logn)$, so totally all operations use $O(nlogn)$ time.
- For AVL, the time complexity is $O(nlogn)$, this is the normal condition we analysed in class.
- For SPLAY, the time complexity is $O(nlogn)$, this is the normal condition we analysed in class.
- So in graph 3, we see BST the fastest because its constant coefficient (of time complexity) is lower than other two. The other two still share similar performances.

Analysis for Different Trees:

- For BST, as analysed above, the constant coefficient (of time complexity) of Data Type 2 is nearly 2 times of Data Type 1, so in theory it will use 2 times the time of Data Type 1. In graph 4 we can see that experiment result clearly. For Data Type 3, BST run very fast because it's average $O(nlogn)$, not $O(n^2)$ of the other two.
- For AVL, the performance of Data Type 1 and Data Type 2 is similar, for the final balanced tree after insertion is the same. For Data Type 3, as there are no similar pattern, the time used mainly depends on its high constant coefficient (of time complexity).
- For SPLAY, the performance of Data Type 1 and Data Type 2 is similar, for the final balanced tree after insertion is the same. However, Data Type 2 is faster, as deletion operation is always on the root. For Data Type 3, as there are no similar pattern, the time used mainly depends on its high constant coefficient (of time complexity).

A Final Conclusion:

- According to the graphs above, for the data type 1 and 2, the AVL-Tree and Splay Tree are much faster than the Binary Search Tree and the Splay tree runs fastest. However for type 3, the Binary Search Tree runs much faster than the other and the Splay Tree cost the longest time. What's more, for Binary Search Tree, the time

it cost of type 3 is the lowest and type 2 is the longset. For AVL-Tree and Splay Tree, the time cost by type 3 is much highest than the other two.

- Under more random data, AVL tree's performance is significantly better than Splay tree. Since the data and operations are very random, the time locality of data access is poor, and the caching strategy of the Splay tree is difficult to work; and it has no additional balancing measures, resulting in a large number of nodes to be accessed during searching and other operations. On the contrast, AVL-Tree's balance conditions are more strict, and the lower tree height ensures the time complexity of a single operation. However, when accessing sequential data, Splay trees have significant performance advantages. This data pattern allows the caching strategy of the Splay tree to play a great role, and the target node appears close to the root node, and the number of nodes that need to be accessed is small.

Appendix (readme.txt)

```
We suppose reading the source files for more comments. The files:

Main Program and Data Generator - main.cpp, gen.h & gen.cpp. Mainly
C++, by Zuo.
BST - BST.h & BST.cpp. Mainly C++, by Zuo.
SPLAY - splaytree.h & splaytree.cpp. Mainly C, by Han.
AVL - AVLtree.h & project1_AVLtree.cpp. Mainly C, by Pan.

This report is mainly written by Shi.

We recommend using DEV-C++ v5.15 to open the project, and TDM-GCC
9.2.0 to compile. The C++ standard shall be C++11.
The .o files are compiled by TDM-GCC 9.2.0 in 64-bit. You may need
delete them to re-compile.
```