CompSci 261P

Data Structures

Project 2 - **Binary Search Trees**

Qixiang Zhang

qixianz@uci.edu

70644128

March 1 2019

Outline

1. Project Overview
2. Implementation Details
3. Algorithms
4. Tests and Results
5. Summarize

1 - Project Overview

The data structures to be implemented are:

1. A standard (unoptimized) **binary search tree**
2. **AVL** Tree
3. **Treap**
4. Treap variation I - a **faster treap**
5. Treap variation II - using **AVL deletion**

For each data structure you implement, you must at minimum implement the following operations:

1. Create
2. Search
3. Insert
4. Delete

These two questions will be answered with empirical tests:
*(**operations** include "insert", "search", and "delete")*

Factor 1 - Population (number of elements):

1. How does the population affect the time complexity of all operations (insert, search, delete)?

Factor 2 - Insertion order:

2. How does pre-sorting the keys affect the time complexity of all operations?

Assumptions:

1. The inputs are integer keys without associative values for convenience.
2. Duplicate keys are not allowed. (Insertion, search, and deletion only include unique keys)

2 - Implementation Details

Language: C++ (works on ALL OpenLab machines, makefiles are provided)

Data Members	Description
value, *left, *right, *parent	- The basic data members for every tree node: key value, left child pointer, right child pointer, parent pointer
balance_factor	- Left sub-tree height minus right sub-tree height.
root	- The root of a tree.
population	- Number of keys stored in the tree
priority	- Available only for treap and its variants, a randomized number served as guidance for rotation (balance the tree)

Behaviors	Description
create(), create(count)	- Create an empty tree; for treap and its variants, need a count parameter to generate the randomized priority values
search_private(key, node) search(key)	- Search through the tree using the key value - No return value

insert_private(key, node) insert(key)	- Insert the key into the correct location of the tree using the key and the priority value - No return value
remove_private(key, node) remove(key)	- Delete the key in the tree - No return value
get_population()	- print the current number of population in the tree - No return value
display(flag)	- print the current tree preorder, inorder, postorder, 2d based on the flag. - No return value

3 - Algorithms

Common for all binary search trees and variants:

```
struct Node                          // initialize each node (private)
        int value;
        Node* left;
        Node* right;
        Node* parent;
        Node(int v)
              value = v;
              left = nullptr;
              right = nullptr;
              parent = nullptr;
int get_max_height(Node* np)    // get the tree height for a given node (private)
      if(!np)
            return 0;
      else
            int left    = get_max_height(np->left);
            int right = get_max_height(np->right);
            if ( left >= right )
                  return left + 1;
            else
                  return right + 1;
 void search_private(int v, Node* np){        // search operation taken a key value and given starting node (private)
        if(np){
              if(np->value > v) search_private(v, np->left);
              else if(np->value < v) search_private(v, np->right);
              else if(np->value == v) cout << "= node " << v << endl;
              Else cout << "!! NOT FOUND " << v << endl;
        }
        Else cout << "Search not possible - Tree is empty!!" << endl;
 void search(int v) {search_private(v, root); }         // always start searching from the root of the tree (public)
 void postorder(Node* np){            // printing the tree nodes in postorder starting from a given node
        if (!np) return;
        postorder(np->left);
        postorder(np->right);
        cout << np->value << " ";
 void preorder(Node* np){             // printing the tree nodes in preorder starting from a given node
        if (!np) return;
        cout << np->value << " ";
        preorder(np->left);
        preorder(np->right);
```

```
Void inorder(Node* np){                    // printing the tree nodes in inorder starting from a given node
        if (!np) return;
        inorder(np->left);
        cout << np->value << " ";
        inorder(np->right);
    void get_population(){cout << "current population: " << this->population << endl;}          // getter method for the population
    void free_tree(Node* np)                // destruct the tree starting from a given node
        //Basically use the postorder traversal algorithm and delete the nodes one by one
```

Binary Search Tree - BST

Pseudo code:

```
helper_findmin(Node* np)       // helper finding the min node starting from np
    Node* min_node = np;
    while( min_node->left )
        min_node = min_node->left;
    return min_node;
helper_remove_inorder_successor( Node* np )     // helper to remove node that has 2 children
    locate the inorder successor of np's right children
    swap the values of np and the ssr
    call remove to remove the ssr starting from the right children
insert_private( int value, Node* np )    // private
    Create a new node - node_insert
    Initialize 2 pointers one as a head at the np, the other as the parent as nullptr
    Use binary search to find the correct location where to insert the node
    Determine if the insert node is the left or right child (leaf) of the parent pointer
    Insert the node, update the parent's child pointer and the node's parent pointer
    Give the confirmation message
insert ( int value )   // public
    If there is no root, create a new node with given value and insert as a root.
    Else call the insert_private function to insert the node
    ++population
remove_private( int value, Node* np )        // private
    Initialize 2 pointers one as a head at the np, the other as the parent as nullptr
    Use binary search to find the node
    If the node has no children, delete it
        Else if the node has 1 child, swap with it, and then delete it
        Else if the node has both children, find the in-order successor and swap with it, then delete it
remove( int value )        // public
    Call the remove_private function
    --population
```

Important Notes:

This binary search tree is the simplest tree with fewest data members among other binary search trees and the variants. However, it set the foundation to the further development of the later advanced binary search tree algorithms.

AVL - AVL

Pseudo code:

```
max_height(Node* np)
    // get the max height for a given node
update_bf( Node* np)
    // update the balance factor for a given node
rotateLeft( Node* A )
    // get A's right
    Node* B = A->right;
    // break A's right link
    A->right = nullptr;
    // update A's parent's pointer if any
    if(A->parent){
        if(A->parent->left == A)
            A->parent->left = B;
        else A->parent->right = B;
        // update B's parent
        B->parent = A->parent;
    }
    else B->parent = nullptr;
    // update A's parent
    A->parent = B;
    // move B's left child as A's right child if any
    if(B->left){
        Node* C = B->left;
        C->parent = A;
        A->right = C;
    }
    B->left = A;
    update_bf( B );
    update_bf( A );
    return B;
}
rotateRight ( Node* A )
    // mirror of rotateLeft function

rotateLeftThenRight ( Node* A )
    A->left = rotateLeft( A->left );
    return rotateRight(A);
}
rotateRightThenLeft ( Node* A ){
    // mirror of rotateLeftThenRight function
```

```
rebalance_tree( Node* np )
    - Update the balance factor for the node
    - If the balance factor is 2, we want to rotate the tree right since the left is heavier
    - Else if the balance factor is -2, we want to rotate the tree left since the right is heavier
    - Lastly, call the rebalance_tree on the nodes above if the node has a parent and not a root
helper_findmin(Node* np)      // Same as the binary search tree
helper_remove_inorder_successor( Node* np ){      // Same as the binary search tree
insert_private( int value, Node* np ):   // private Same as binary search tree; Call rebalance_tree ( parent )
insert ( int value ):  // public Same as the binary search tree
remove_private( int value, Node* np )      // private Same as the binary search tree; Call rebalance_tree ( parent )
remove( int value )      // public Same as the binary search tree
```

Important Notes:

Getting the rotations to work properly is the heart of this AVL design. If implemented poorly, much of the run time and the memory can be eaten up by tail recursions. The code above is mostly implemented iteratively to avoid tail recursions.

Treap - TP0

Pseudo code:

```
max_height(Node* np)
    // get the max height for a given node
update_bf( Node* np)
    // update the balance factor for a given node
rotateLeft( Node* A )
    // Same as AVL
rotateRight ( Node* A )
    // Same as AVL
```

```
siftDown( Node* np ) {
    while( np->left && np->right ){
        if( np->left->priority > np->right->priority )
            np = rotateRight(np)->right;
        else
            np = rotateLeft(np)->left;
    }
    remove_private(np->value, np);
}
```

void siftUp(Node* np): **// move the node up based on the priority value**
while(np && np!=root && np->priority > np->parent->priority):
if (np->parent->left == np): np = **rotateRight**(np->parent)
else: np = **rotateLeft**(np->parent)
insert_private(int value, int priority, Node* np): **// private**
Use the standard binary search tree insertion procedure
In the end, call the siftUp to bubble up like a heap using rotations based on the node's priority value
remove_private(int value, Node* np) **// private**
Use the standard binary search tree deletion procedure
In the end, call the siftDown to bubble down like a heap using rotations
Create(int count) **// public**
Create a root null pointer
Generate random a total of given size series of integers as priority values

Important Notes:
Different from AVL and standard binary search tree, the insertion takes additional parameter: the priority value. Like a heap so I implemented the siftUp and siftDown functions for the insertion and deletion operations, respectively.

Treap Variant I - A Faster Treap! - TPf

Important Notes:
Essentially, with Professor Dillencourt's permission, I was granted the permission to implemented this data structure out of curiosity. In class, I wondered if there are any shortcuts for the rotations that AVL tree or treap can do. Here, I found that **presorting makes the insertion faster and will result in the exact same tree as the regular treap**. This data structure does not require any rotations. The only thing that is different from binary search tree is that the elements are pre-sorted before insertion (randomized).

Treap Variant II - Fast Treap with AVL deletion ! - TPa

Important Notes:
As the title suggested, this data structure takes in data just like a fast treap (pre-sorted). Different from the fast treap (TPf), TPa uses the AVL deletion procedure when deleting a node (call the rebalance function). Therefore, implementing this needs to take account of the balance factor for each node.

4 - Tests and Results

Test 0 - Does TPf produce the same tree as TP0 (regular treap)?

The answer is yes!

We can prove that the regular treap and the faster treap can produce the same tree by looking at the postorder printout.

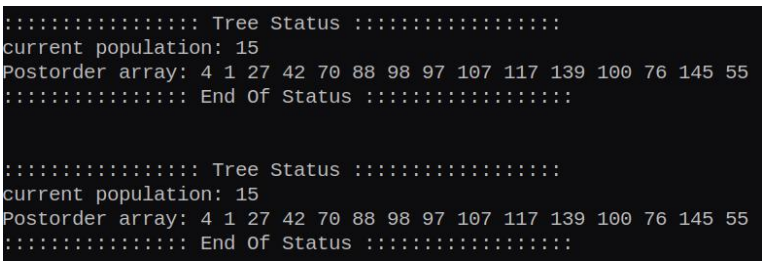
In this test, tp0 is inserted with priority without sorting; tpf is inserted after sorting. We get the same tree!

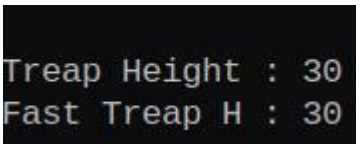
```
for( int i=0; i<size; ++i ) tp0.insert(elements[i].first, elements[i].second);
sort( elements.begin(), elements.end(), mycompare() );
for( int i=0; i<size; ++i ) tpf.insert(elements[i].first, elements[i].second);
```

Instead of follow the regular treap insertion algorithm, this faster treap insert just like a binary search tree. However, the important thing is to pre-sort the items (**randomize** it). There would be no more rotations in this treap!

Here is a screen shot for testing a small sample size of 15 nodes (on the left)

Another sample for testing another sample size of 10000, and we get the same tree height of 30! (on the right).

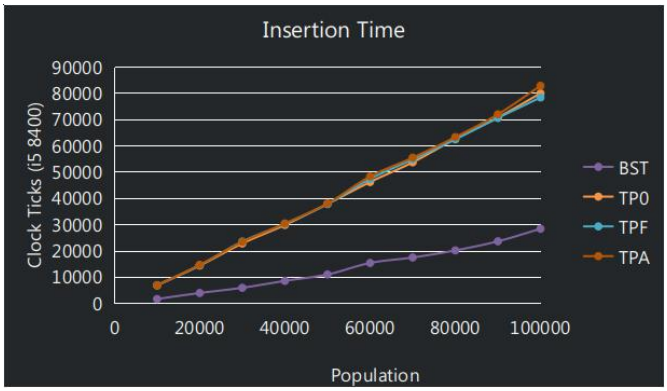


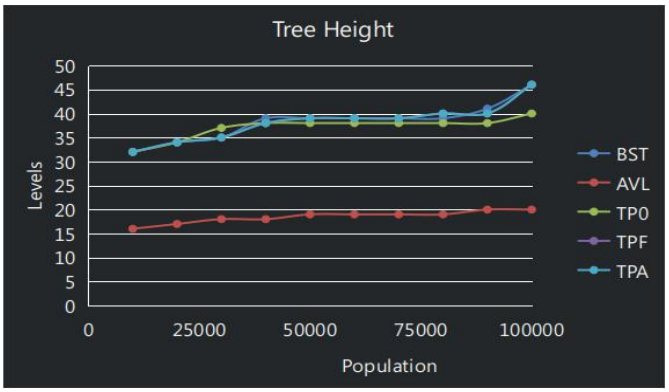


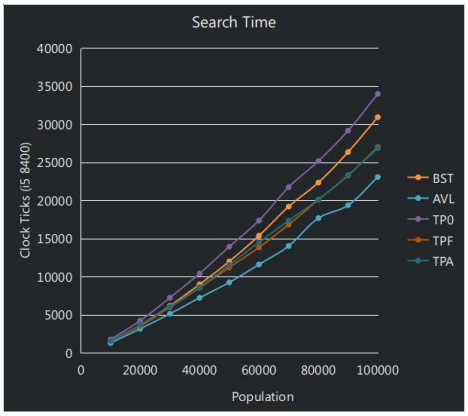
The test files are under the directory /tp0=tpf

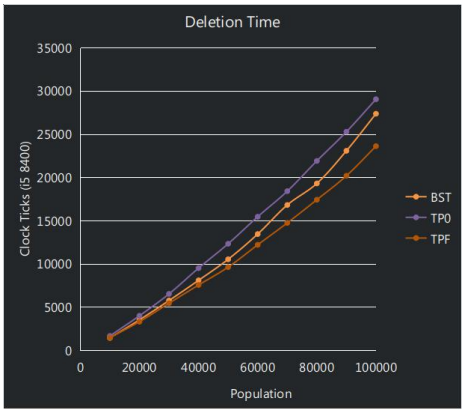
Test 1 - How do these data structures differ in time complexity for all operations?

In this test, the test is set up using various populations and operations mix. The runtime and tree heights are recorded as below:







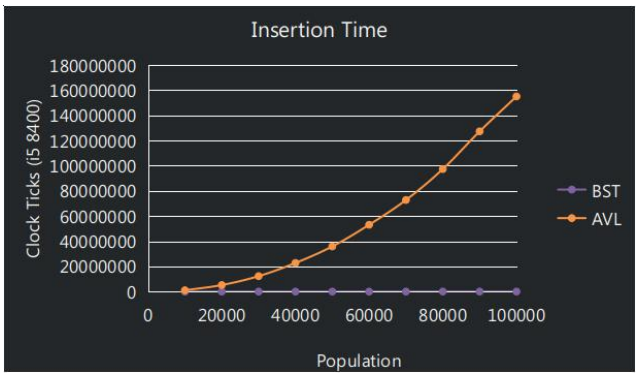


Insertion	BST	AVL	TP0	TPF	TPA
10000	1552	1045221	6724	6814	6742
20000	3833	5222001	14262	14413	14555
30000	5818	12289810	22691	23404	23271
40000	8444	22769145	29736	30129	30309
50000	10885	35900547	37757	37703	37919
60000	15262	53107655	46079	47253	48208
70000	17346	72789171	53591	54694	55373
80000	20048	97359306	62494	62332	63191
90000	23514	127307660	70655	70477	71882
100000	28287	155067437	79854	78278	82714

Deletion	BST	AVL	TP0	TPF	TPA
10000	1419	1120334	1649	1453	1262098
20000	3510	5488251	3983	3295	5996323
30000	5744	12696771	6513	5436	13947893
40000	8079	23232854	9493	7541	25944787
50000	10514	36540475	12309	9611	41396380
60000	13431	54214970	15433	12188	58929273
70000	16792	74199604	18390	14718	80896383
80000	19286	99935673	21889	17400	108045051
90000	23058	127624303	25259	20172	139878439
100000	27344	161956228	29021	23594	167203021

Height	BST	AVL	TP0	TPF	TPA
10000	32	16	32	32	32
20000	34	17	34	34	34
30000	35	18	37	35	35
40000	39	18	38	38	38
50000	39	19	38	39	39
60000	39	19	38	39	39
70000	39	19	38	39	39
80000	39	19	38	40	40
90000	41	20	38	40	40
100000	46	20	40	46	46

Search	BST	AVL	TP0	TPF	TPA
10000	1534	1283	1758	1574	1538
20000	3638	3140	4210	3497	3657
30000	6160	5113	7229	5969	6067
40000	8979	7207	10393	8519	8600
50000	11985	9228	13916	11189	11520
60000	15359	11582	17350	13818	14471
70000	19174	14009	21721	16818	17344
80000	22326	17658	25153	20054	20103
90000	26331	19352	29138	23272	23302
100000	30907	23058	33961	26994	26828



Binary search tree has the shortest insertion time. It does not have to perform rotations, heapifications, or randomization. AVL has the longest insertion time because it has to constantly maintain the balance factor for almost every node between -1 and 1. Treap variant the fast treap inserts elements faster than normal treaps because it does not perform heapifications.

The tree heights seem to be highly correlated with the search time for all data structures. AVL has the least search time since it is almost a perfect balanced binary search tree. It would have the least tree heights. On the contrary, treap and treap variants have the worst search time.

AVL has the worst deletion time. Treap variant TPf has the best deletion time since it does not perform heapifications or rotations. It has the same delete operation as a standard binary search tree. It is faster since the elements are inserted in a random order which statistically can be better than a normal binary search tree.

Test 2 - How does pre-sorting the keys affect the time complexity of all operations?

In this test, the test is set up using sequential and non-sequential data and operations mix. The runtime and tree heights are recorded as below:

10000	BST(s)	BST(n)	AVL(s)	AVL(n)	TP0(s)	TP0(n)	TPF(s)	TPF(n)	TPA(s)	TPA(n)
Insertion	147201	1552	608171	1045221	5342	6724	146006	6814	145853	6742
Height	10000	32	14	16	34	32	10000	32	10000	32
Search	273019	1534	776	1283	1008	1758	269533	1574	266005	1538
Deletion	153	1419	495428	1120334	453	1649	150	1453	154	1262098

30000	BST(s)	BST(n)	AVL(s)	AVL(n)	TP0(s)	TP0(n)	TPF(s)	TPF(n)	TPA(s)	TPA(n)
Insertion	1321344	5818	5611272	12289810	3009	22691	1318538	23404	1319639	23271
Height	30000	35	15	18	36	37	30000	35	30000	35
Search	2512605	6160	2531	5113	3248	7229	2522608	5969	2479927	6067
Deletion	465	5744	5063217	12696771	1613	6513	476	5436	485	13947893

50000	BST(s)	BST(n)	AVL(s)	AVL(n)	TP0(s)	TP0(n)	TPF(s)	TPF(n)	TPA(s)	TPA(n)
Insertion	3614803	10885	16634736	35900547	27295	37757	3618148	37703	3635605	37919
Height	50000	39	16	19	39	38	50000	39	50000	39
Search	7021721	11985	4513	9228	5989	13916	6869116	11189	7024815	11520
Deletion	768	10514	15186892	36540475	2821	12309	752	9611	772	41396380

For binary search trees, everything got worse except for deletion. Since the elements are inserted and deleted in the same order, deletion time become way less than usual. However, this data structure is the same as a linked list. The search time becomes linear instead of logarithmic. Same can be applied to treap's variants.

For AVL and regular treap, everything has gotten better! Sequential insertion to both data structures seem to have a good effect on building the tree. My guess is that the number of rotations have decreased since the keys are inserted in a sorted order.

5 - Summarize

Conclusion:

1. The Big-O notation and amortized time may not always apply

From this project, we can see that the runtime for insertion, search, and deletion varies differently and cannot be easily generalized by the Big-O notation or amortized time. There are other factors including the population and the specification on the input keys.

2. Choosing a good binary search tree

Choosing the best binary search tree depends on the need and usage of the data. If the search time is strictly limited, AVL is recommended since it has the best search time (since it is almost a perfectly balanced binary search tree - tree height bounded by 1.44 x lg(number of nodes)). Basic binary search tree is a very easy algorithm to implement but it is always a good idea to randomize the input data before the insertion (like a fast treap!). Treap is also a good choice when the number of data inputs are unknown ahead of time. Regular Treap can assign randomized priority values to each individual elements regardless the insertion order. However, if the data is known before hand, it is always best to randomize it and just insert all using the fast treap algorithm.

3. Project Extension

Randomize the input really boosts the performance of binary search tree. However, to further improve the maintenance of the data set for long term modification, maybe calling the rebalance function of AVL on the treap may not be a bad idea. The TPa was originally designed to serve this purpose. However, it would require more empirical experiments to make this deduction.