CompSci 261P Project #2: Binary trees and their variants

Program: Master of Computer Science
Name: Yi-Dar Liu
ID#: 32845675

CONTENT

A. INTRODUCTION 1
B. DESCRIPTION OF BINARY TREE ALGORITHMS 2
C. THEORETICAL ANALYSIS OF RUNNING TIMES 8
D. EXPERIMENTS 11
E. COMPARATIVE EXPERIMENTAL ANALYSIS 14
F. CONCLUSIONS 31
G. REMARKS 32
H. REFERENCES 32

A. INTRODUCTION

Binary tree is a common data structure used to store, query, and modify ordered data. This report explores four binary trees: Standard Binary Search Tree (BST), AVL Tree, Treap, and Splay Tree. Since operation cost of a binary tree is proportional to tree height, how to keep a tree balanced has been an important topic in the computer science academia. Unlike Standard BST, AVL Tree, Treap, and Splay Tree have their own ways to achieve this goal. For convenience, we call Standard BST just “BST” and the other three algorithms “BST variants.” Because balancing incurs extra cost, we wonder if balanced trees perform really better than BST. If the answer is yes, which balancing scheme is the most efficient?

This report is organized as follows: Section B describes the four binary tree algorithms. Section C analyzes their running times. Section D details the experiments conducted in this project. Section E shows and explains the empirical results. Section F concludes the insights gained from Section E. Section G supplements additional remarks about this project. Section H lists the references cited in this report.

General Terminology and Notation:

- [i] means ith reference in the “H. REFERENCES” section.

B. DESCRIPTION OF BINARY TREE ALGORITHMS

In a binary tree, search cost is proportional to tree height. Therefore, minimizing tree height is equivalent to minimizing search cost. If each path from leaf node to root node is (almost) equal, the worst-case search cost will be log n. In this case, we call the tree “balanced.” It is the most favorable situation. However, it is also likely that all nodes are linked as a list. In this case, the search cost becomes O(n). That is the last scene every computer scientist would like to see.

Many strategies have been developed to keep a tree balanced. Some store extra balance information in the tree nodes, and if the balancing conditions are violated, the trees are modified to restore balance. We call them “self-balancing trees.” AVL Tree and Red-Black Tree belong to this group. Others capitalize on randomization techniques in hope that the trees can be automatically balanced without sophisticated adjustment schemes. We call them “randomized trees.” Treap and Skip-list are two of the examples. Still others associate each operation with some special modification routines to achieve amortized bounds of O(log n) running time. We call them “self-adjusting trees.” Splay Tree is the most famous one.

In this project, we select one representative from each category to compare their relative efficiencies: AVL Tree, Treap, and Splay Tree, respectively. Their balancing methods are summarized in the following table:

	Category	Balance Information	Balancing Condition	Balancing Method
BST	Standard tree	None	None	None
AVL Tree	Self-balancing tree	Node height	The height of the left subtree and that of the right subtree differ by at most one (deterministic)	AVL rotation
Treap	Randomized tree	Node priority value	Tree is heap-ordered with respect to priority values (random)	AVL rotation
Splay Tree	Self-adjusting tree	None	None	Splay

Below we provide more detailed description and pseudo-code for each algorithm. All of them use the same BST search and insertion. They vary only in balancing methods and deletion.

In the pseudo-code, we use the following terminology:

A tree structure stores the following information:

- root: root node of the tree
- n: number of nodes in the tree

A node structure stores the following information:

- k: key value
- v: value associated with the key
- h: height, defined as the longest path from leaf to the node; root.h = tree height, leaf.h = 0; only used in AVL Tree
- p: priority value, a double-type number between 0 and 1; only used in Treap
- left: pointer to the left child; leaf.left = NULL
- right: pointer to the right child; leaf.right = NULL
- parent: pointer to the parent; root.parent = NULL
- left_flag: 1 if the node is the left child of its parent, 0 if right child

BST

In BST, key values satisfy the relationship: left child's key < parent's key < right child's key.

Search:

When searching for a key, we start from the root. If the key value is lower than the node key, we go to the left child; otherwise, we go to the right child.

```
def search(k):
    node ← root
    while node != NULL
        if k < node.k: node ← node.left
        else if k > node.k: node ← node.right
        else: return node
    k is not in data
    return node
```

Insert:

If null child is found, then insert the key into that slot.

```
def insert(k, v):
    node ← search(k)
    store <k, v> in node
    return node
```

Delete:

Let v be the node that contain the key to delete. If v has a null left child, replace v by v's right child. Otherwise, if v has a null right child, replace v by v's left child. If neither child of v is null, copy inorder successor of v, say x, into v and delete x.

```
def delete(k):
    node ← search(k)
    if node = NULL:
        k is not in data
        return node
    if node.left = NULL and node.right = NULL: directly delete node
    else if node.left = NULL: node ← node.right
    else if node.right = NULL: node ← node.left
    else:
        find node's inorder successor x
        copy x to node
        return delete(x.k)
    return node.parent
```

AVL Tree

AVL Tree is similar to BST except that each node stores height information. Every time the tree is modified, it has to update nodes' height information. For each node, the height of the left subtree and that of the right subtree can differ by at most one. If the invariant is violated, it does rotations to restore balance.

The pseudo-code of AVL rotation is as follows:

```
def AVL_rotate(x):
    if x.right.h – x.left.h ≥ 2:
        if x.right.right.h – x.right.left.h ≥ 0:
            return rotate_left(x)
        else:
            return rotate_right(x.right)
            return rotate_left(x)
    if x.right.h – x.left.h ≤ -2:
        if x.left.right.h – x.left.left.h ≤ 0:
            return rotate_right(x)
        else:
            return rotate_left(x.left)
            return rotate_right(x)

def rotate_right(x):
    y ← x.left
    x.left ← y.right
    y.right ← x
    update x.h, y.h, x.parent, y.parent, y.right.parent
    return y

def rotate_left(x):
    y ← x.right
    x.right ← y.left
    y.left ← x
    update x.h, y.h, x.parent, y.parent, y.left.parent
    return y
```

Search:

Same as BST.

Insert:

Initial insertion is the same as BST, but it needs some extra work: (1) update height information along the path from the inserted node to the root; (2) if the balancing condition of a node is violated (subtrees' height difference is more than 1), rotate the node.

```
def insert(k, v):
    node ← BST.insert(k, v)
    while node != NULL:
        if |node.right.h – node.left.h| ≥ 2:
            node ← AVL_rotate(node)
        else:
            update node.h
        node ← node.parent
```

Delete:
As with insertion, in addition to BST deletion, it also needs to do extra work to ensure tree balance.

```
def delete(k):
    node ← BST.delete(k)
    while node != NULL:
        if |node.right.h – node.left.h| ≥ 2:
            node ← AVL_rotate(node)
        else:
            update node.h
        node ← node.parent
```

Treap

Like AVL Tree, Treap also attaches extra information to a node as the balancing condition. Unlike AVL Tree, which stores a node's height, Treap stores a node's priority value, a uniformly-distributed random number between 0 and 1. Another difference is that once a priority value is assigned, it will never change, as opposed to node's height in AVL Tree, which needs to be updated every time the tree is modified.

In addition to satisfying the key order, Treap also needs to be heap-ordered, which means child's priority value must be lower than parent's priority value. Once the invariant is violated, it carries out AVL-style rotation to restore order.

Search:
Same as BST.

Insert:
Initial insertion is the same as BST, but it needs some extra work: (1) assign a priority value to the inserted node; (2) if the balancing condition is violated (the node's priority value is higher than that of its parent), rotate the node until the tree is completely heap-ordered.

```
def insert(k, v):
    node ← BST.insert(k)
    node.p ← random()
    while node != root and node.p > node.parent.p
        if node.left_flag = True:
            node ← rotate_right(node.parent)
        else:
            node ← rotate_left(node.parent)
```

Delete:

If the deleted key is at a leaf node, just remove the node. Otherwise, use AVL rotations to rotate the node down until it becomes a leaf, and delete it.

```
def delete(k):
    node ← search(k)
    while node != leaf:
        if node.left = NULL:
            node ← rotate_left(node)
        else if node.right = NULL or node.left.p > node.right.p:
            node ← rotate_right(node)
        else:
            node ← rotate_left(node)
    delete node
```

Splay Tree

Unlike AVL Tree and Treap, it does not store any extra information to check balancing condition. Instead, after each operation, it performs a specific series of rotations, called “splays,” to make a node the root.

The pseudo-code of splay is as follows:

```
def splay(x):
    //zig
    if x.parent = root:
        if x.left_flag = True:
            return rotate_right(x.parent)
        else:
            return rotate_left(x.parent)
    else
        //zig-zig
        if x.left_flag = x.parent.left_flag:
            if x.left_flag = True:
                y ← rotate_right(x.parent.parent)
                return rotate_right(y)
            else:
                y ← rotate_left(x.parent.parent)
                return rotate_left(y)
        else: //zig-zag
            if x.left_flag = True:
                y ← rotate_right(x.parent)
                return rotate_left(y.parent)
            else:
                y ← rotate_left(x.parent)
                return rotate_right(y.parent)
```

Search:

```
def search(k):
    node ← root
    while node != NULL or node.k != k:
        if k < node.k: node ← node.left
        else if k > node.k: node ← node.right
    if node = NULL:
        k is not in data
        node ← node.parent
    while node != root:
        node ← splay(node)
```

Insert:

```
def insert(k, v):
    node ← BST.insert(k, v)
    while node != root:
        node ← splay(node)
```

Delete:

```
def delete(k):
    node ← BST.delete(k)
    while node != root:
        node ← splay(node)
```

C. THEORETICAL ANALYSIS OF RUNNING TIMES

Below is the summary of running times of each binary tree algorithm.

Notation:
- n: tree size, i.e., number of nodes in the tree
- Worst: worst-case time per operation
- Typical Worst: typical worst-case time per operation
- Average: expected time per operation under the assumption that keys are inserted in random order
- Amortized: amortized time, i.e. worst average time per operation

	Search	Insert	Delete
BST	Worst: O(n) Average: $O(\log n)$	Worst: O(n) Average: $O(\log n)$	Worst: O(n) Average: $O(\log n)$
AVL Tree	Worst: $O(\log n)$ Average: $O(\log n)$	Worst: $O(\log n)$ Average: $O(\log n)$	Worst: $O(\log n)$ Average: $O(\log n)$
Treap	Typical Worst: $O(\log n)$ Average: $O(\log n)$	Typical Worst: $O(\log n)$ Average: $O(\log n)$	Typical Worst: $O(\log n)$ Average: $O(\log n)$
Splay Tree	Amortized: $O(\log n)$ Average: $O(\log n)$	Amortized: $O(\log n)$ Average: $O(\log n)$	Amortized: $O(\log n)$ Average: $O(\log n)$

Although all BST variants claim to achieve the O(log n) bound of running time, the inherent meaning of the bound is different: AVL Tree is in terms of worst-case time, Treap typical worst-case time, Splay Tree amortized time. Besides, although search, insert, and delete operations' costs look the same in terms of big O notation, they should have different constant terms. For example, the expected number of rotations (O(1) time per rotation) may vary among the three kinds of operations, as the following table shows:

A = always if the key is not at the root, O = often, S = sometimes, N = never

Require rotations? (expected number of rotations)	Search	Insert	Delete
BST	N	N	N
AVL Tree	N	S (≤ 2)	S (≤ O(log n))
Treap	N	S (≤ 2)	O (≤ 2)
Splay Tree	A ($O(\log n)$)	A ($O(\log n)$)	A ($O(\log n)$)

Based on the aforementioned theoretical costs, we would like to predict their relative efficiencies in our empirical study. First, we define our cost measure as follows:
- traversing cost = number of nodes accessed for lookup
- update cost = number of nodes accessed for update
- rotation cost = number of rotations
- cost of a BST operation = traversing cost
- cost of an AVL Tree operation = traversing cost + update cost + rotation cost
- cost of a Treap operation = traversing cost + rotation cost
- cost of a Splay Tree operation = traversing cost + rotation cost

Although rotating a node is normally costlier than accessing a node, for simplicity, we assume the costs for accessing a node, updating a node, and rotating a node are equal.

Using this definition of cost measure, we make the following predictions:

BST:

- Tree height: If the order of keys is random, BST is called "Random BST," and it is shown that its expected height will be c*log n for some constant c. If the order of keys is sequential, the tree will become a linked list, so its height will be n.
- Search cost: On average, it will be less than tree height; in the worst case, it will be equal to tree height.
- Insertion cost: Insertion is involved with traversing from the root to leaf, so it will be the most expensive operation. Because BST does not need to do any other adjustments than inserting the node, the average insertion cost should be less than tree height.
- Deletion cost: Sometimes it needs to find the deleted node's inorder successor, so deletion cost should be higher than search cost. However, the inorder successor is not necessarily at leaf, so deletion cost should be lower than insertion cost.

In summary, on average, search cost < deletion cost < insertion cost < tree height = c*log n for some c if the order of keys is random. Worst case will be walking the longest path from the root to leaf, and the three kinds of operations will have cost as much as height in this case.

AVL Tree:

- Tree height: Because AVL Tree is always balanced, its tree height will be always log n. It also implies that the path from the root to any leaf node is roughly log n long.
- Search cost: It will never be more than log n.
- Insertion cost: Aside from inserting the node, it needs to update height along the path from the new node to the root. Moreover, when the balancing condition is violated, it needs to do at most two rotations. Therefore, the insertion cost will be between 2*log n and 2*log n + 2.
- Deletion cost: Like insertion, it also needs to update node heights in deletion. Besides, it may need to do as many as O(log n) rotations. We expect deletion cost to be bounded by 3*log n.

In summary, in either average or worst case, search cost ≤ tree height = log n. In addition, average insertion cost = 2*log n + 0~2, and worst-case deletion cost ≤ 3*log n.

Treap:

- Tree height: Because the priority values are uniformly-distributed random variables, the resulting tree of Treap would look like Random BST, and its height will be c*log n for some constant c.
- Search cost: On average, it will be less than tree height; in the worst case, it will be equal to tree height.
- Insertion cost: It is proved that the expected number of rotations in update is bounded by 2, so the average insertion cost will be bounded by tree height plus 2. Because insertion requires traversing from the root to leaf and may need rotations, we expect it to be the most expensive operation.
- Deletion cost: If the node to delete is not at leaf, it needs to be rotated down to leaf before deleted. Accordingly, average deletion cost should be average insertion cost minus 0~2 (number of rotations in insertion), and the worst deletion cost should be equal to tree height. Besides, average deletion cost should be average search cost plus 0~2 (number of rotations in deletion) as well.

In summary, in the worst case, search cost = deletion cost = tree height < insertion cost. On average, search cost < tree height = c*log n for some c, and deletion cost = search cost + 0~2 = insertion cost – 0~2, where insertion cost ≤ tree height + 2.

Splay Tree:

- Tree height: Because its amortized cost is O(log n), we expect its tree height to be c*log n for some c when the number of operations is large enough.
- Search/insertion/deletion cost: Because each operation is involved with finding the node and bringing it back to the root, its cost will be twice as much as that of BST of the same tree shape.

The following table summarizes our predictions. All numbers represent bounds on expectation.

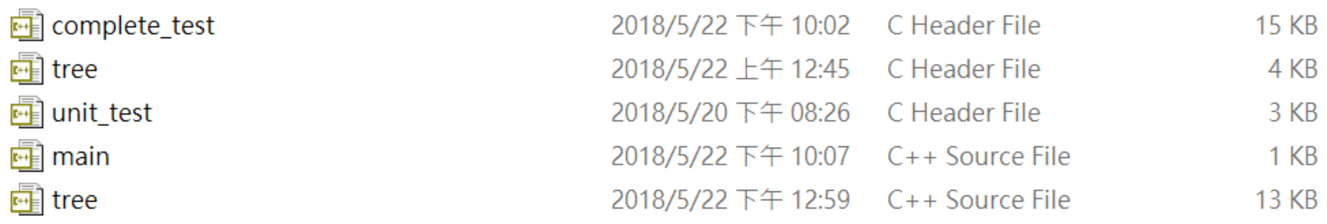
	Height	Search	Insert	Delete	Relationship in average case	Relationship in worst case
Random BST	c*log n	c*log n	c*log n	c*log n	search < delete < insert < height	search = insert = delete = height
AVL Tree	log n	log n	2*log n + 2	3*log n	search < height	search = height
Treap	c*log n	c*log n	c*log n + 2	c*log n	search < height insert ≤ height + 2 delete = search + 0~2 = insert – 0~2	search = delete = height < insert
Splay Tree	c*log n	2c*log n	2c*log n	2c*log n	search < delete < insert	search = insert = delete = 2*height

The table shows that Treap has lower update cost than AVL Tree and lower search cost than Splay Tree. Moreover, it is expected to have height of O(log n) regardless of key order. Therefore, we expect Treap to be among the top performers in most cases.

D. EXPERIMENTS

We use C++ to implement the data structures. The code is organized as follows:

- main.cpp: used to execute complete test
- tree.h and tree.cpp: implement the four binary tree algorithms
- unit_test.h: perform simple tests to debug tree.cpp
- complete_test.h: define the experiment settings and perform the actual experiments which will be explained later



complete_test	2018/5/22 下午 10:02	C Header File	15 KB
tree	2018/5/22 上午 12:45	C Header File	4 KB
unit_test	2018/5/20 下午 08:26	C Header File	3 KB
main	2018/5/22 下午 10:07	C++ Source File	1 KB
tree	2018/5/22 下午 12:59	C++ Source File	13 KB

In Project 1, we design a hash table for Professor to store UCI MCS students' midterm scores. However, student IDs in the hash table is unordered. Suppose Professor would like to arrange final exam seats by student IDs. In this case, hash table is hard to use. It is better to use a binary tree to store student IDs. The data structure also allows Professor to perform predecessor or successor query easily. If a student's handwriting is unclear, Professor can use the query to find a possible ID number.

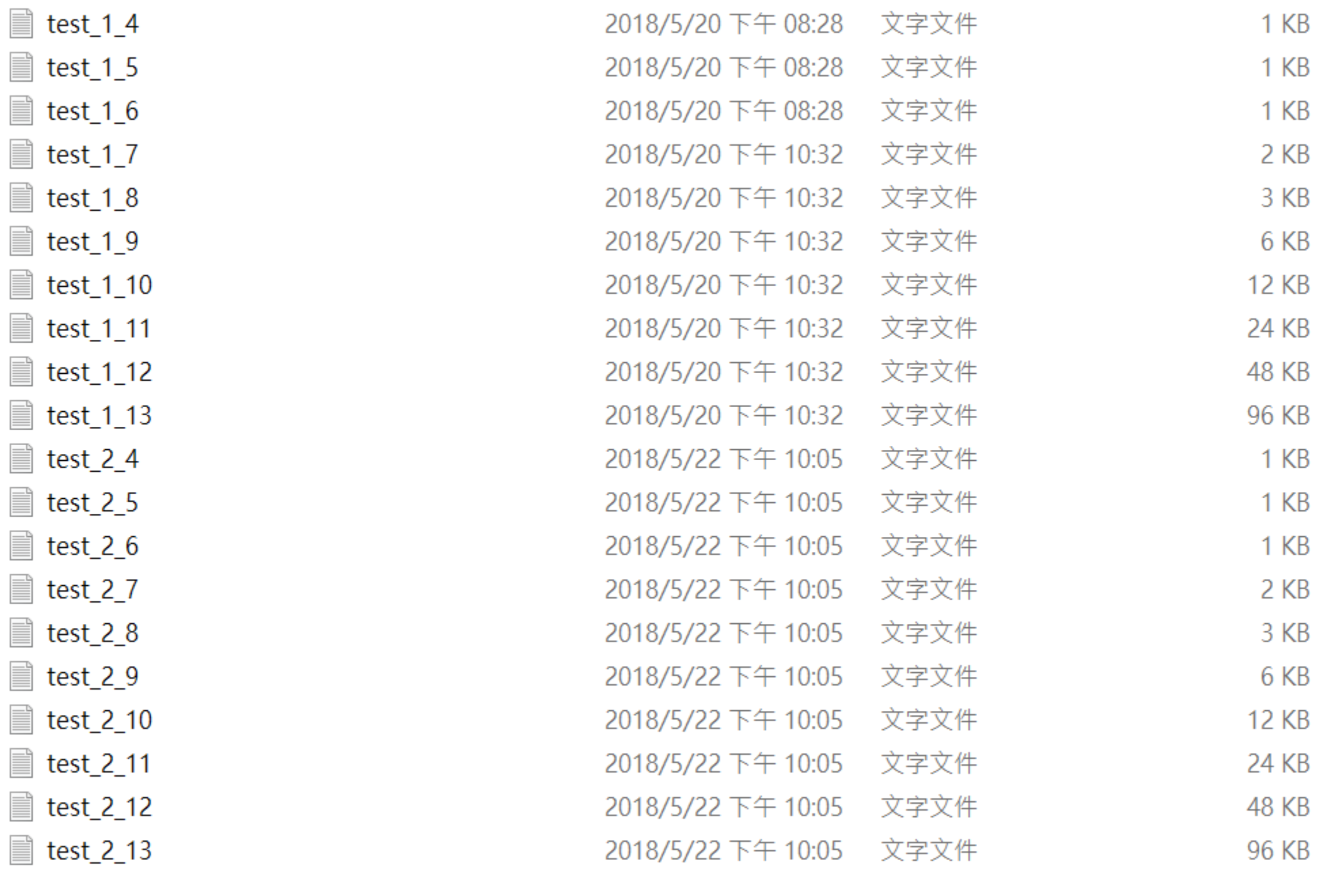
We imagine creating a binary tree for CS261P to record UCI MCS students' final exam scores. Students are identified by 8-digit IDs, which serve as the keys k. Assume the keys are unique, independent, and uniformly distributed over the range between 00000000 and 99999999. The students' CS261P final exam scores are the values v associated with the keys. Assume that the final exam's full mark is 100 points, so score ranges from 0 (assume no extra point loss for wrong answers) to 100. We also assume that MCS students are quite smart, so scores are normally distributed with mean = 75 and standard deviation = 12.5. We use C++ <random> header file to generate these random numbers. The following is an example of how we generate student IDs (keys) and scores (values):

```
#include <random>       // std::default_random_engine
std::default_random_engine generator;
std::uniform_int_distribution<int> k_distribution(0, 99999999);
std::normal_distribution<double> v_distribution(75, 12.5);
k = k_distribution(generator); //generate a key
v = value(v_distribution(generator), d->vmin, d->vmax); //generate a value, value() is a function to transform a double-type number to an integer between specified minimum (d->vmin = 0) and maximum (d->vmax = 100)
```

We generate test cases of different class sizes n from $2^4 - 1 = 15$ to $2^{13} - 1 = 8191$ (assume that students can take the program online, so the class size can be expanded without limits). The files are named as "test_d_i.txt" where:

- d: 1 if the keys are unordered, 2 if the keys are arranged in the increasing order
- $i = \text{ceiling}(\log_2 n)$

For example, "test_1_4.txt" is the test case with $n = 2^4 - 1 = 15$ uniformly distributed keys without specific order, and "test_2_4.txt" arranges the keys in the increasing order. The first line of the files is sample size n, followed by n pairs of "key, value", which are separated by semicolons. The following screenshot shows all the data used for test cases:



test_1_4	2018/5/20 下午 08:28	文字文件	1 KB
test_1_5	2018/5/20 下午 08:28	文字文件	1 KB
test_1_6	2018/5/20 下午 08:28	文字文件	1 KB
test_1_7	2018/5/20 下午 10:32	文字文件	2 KB
test_1_8	2018/5/20 下午 10:32	文字文件	3 KB
test_1_9	2018/5/20 下午 10:32	文字文件	6 KB
test_1_10	2018/5/20 下午 10:32	文字文件	12 KB
test_1_11	2018/5/20 下午 10:32	文字文件	24 KB
test_1_12	2018/5/20 下午 10:32	文字文件	48 KB
test_1_13	2018/5/20 下午 10:32	文字文件	96 KB
test_2_4	2018/5/22 下午 10:05	文字文件	1 KB
test_2_5	2018/5/22 下午 10:05	文字文件	1 KB
test_2_6	2018/5/22 下午 10:05	文字文件	1 KB
test_2_7	2018/5/22 下午 10:05	文字文件	2 KB
test_2_8	2018/5/22 下午 10:05	文字文件	3 KB
test_2_9	2018/5/22 下午 10:05	文字文件	6 KB
test_2_10	2018/5/22 下午 10:05	文字文件	12 KB
test_2_11	2018/5/22 下午 10:05	文字文件	24 KB
test_2_12	2018/5/22 下午 10:05	文字文件	48 KB
test_2_13	2018/5/22 下午 10:05	文字文件	96 KB

We consult reference [1] and conduct the following experiments:

Trees of Stable Size

First build a binary tree of size n by selecting n unique integer keys randomly from a uniform distribution. We would like to learn how algorithm efficiencies are affected by relative frequencies of insert, search, and delete operations. Here we use “activity ratio” to control relative frequencies. It is defined as the ratio of the number of updates (deletions and insertions) to total number of operations. High activity ratio means a dynamic environment where updates are more frequent, while low activity ratio means a static environment where searches are more frequent. Let activity ratio be α. We perform 3n cycles of the following 20 operations:

1. $10*\alpha$ sequences of a random deletion and then a random insertion
2. $20*(1-\alpha)$ random successful searches

Because each deletion is followed by an insertion, tree size will remain at $n\pm1$ during the whole process. For verification purpose, we also separately set up a key reserve which stores all keys in the tree. The key reserve is implemented using C++ <set> header file.

The following code snippet shows how we generate a key to delete/search. We select a uniform random double-type number between 0 and 1 (u), multiple it by tree size (tree->get_element_num()), and then index into the key reserve (k_set) to find the key (k).

```
#include <random>
#include <set>
std::default_random_engine generator;
std::uniform_real_distribution<double> u_distribution(0.0,1.0);
set<int> *k_set;
k_set = new set<int>();
set<int>::iterator k_it;
double u;
int k, h;
u = u_distribution(generator);
h = (int)(tree->get_element_num()*u);
k_it = k_set->begin();
std::advance(k_it, h);
k = *k_it; //k is the key to delete/search
```

The following code snippet shows how we generate a key to insert. We iteratively generate a uniform random integer-type number between 00000000 and 99999999 (k) until it is not in the key reserve (k_it == k_set->end()).

```
std::default_random_engine generator;
std::uniform_int_distribution<int> k_distribution(d.kmin, d.kmax); //d.kmin = 0, d.kmax = 99999999
do{

        k = k_distribution(generator);
        k_it = k_set->find(k);

}while(k_it != k_set->end());
```

In our experiment, α ranges from 0.1 to 0.9 with increment = 0.1. We record worst and average cost for each operation as well as worst and average height during the 3n*20 operations.

Trees of Growing and Shrinking Size

We insert n keys into the tree and then delete all of them from it. We try two kinds of insertion/deletion order: random and sequential. We record the worst and average costs of each insertion and each deletion.

Our cost measure is defined in "C. THEORETICAL ANALYSIS OF RUNNING TIMES." It is different from [1] in two aspects:

1. [1] records traversing cost and rotation cost separately, while we add them together. Nonetheless, we can easily estimate the number of rotations using the relationships among the operation costs.
2. Their empirical results support that AVL Tree outperforms Splay Tree. However, their experiment does not take into account update cost, so their results may overestimate AVL Tree's performance. We include the cost in our performance measure.

Other implementation details:

- We use the following code to generate a priority value for a node (new_node->p) in Treap:

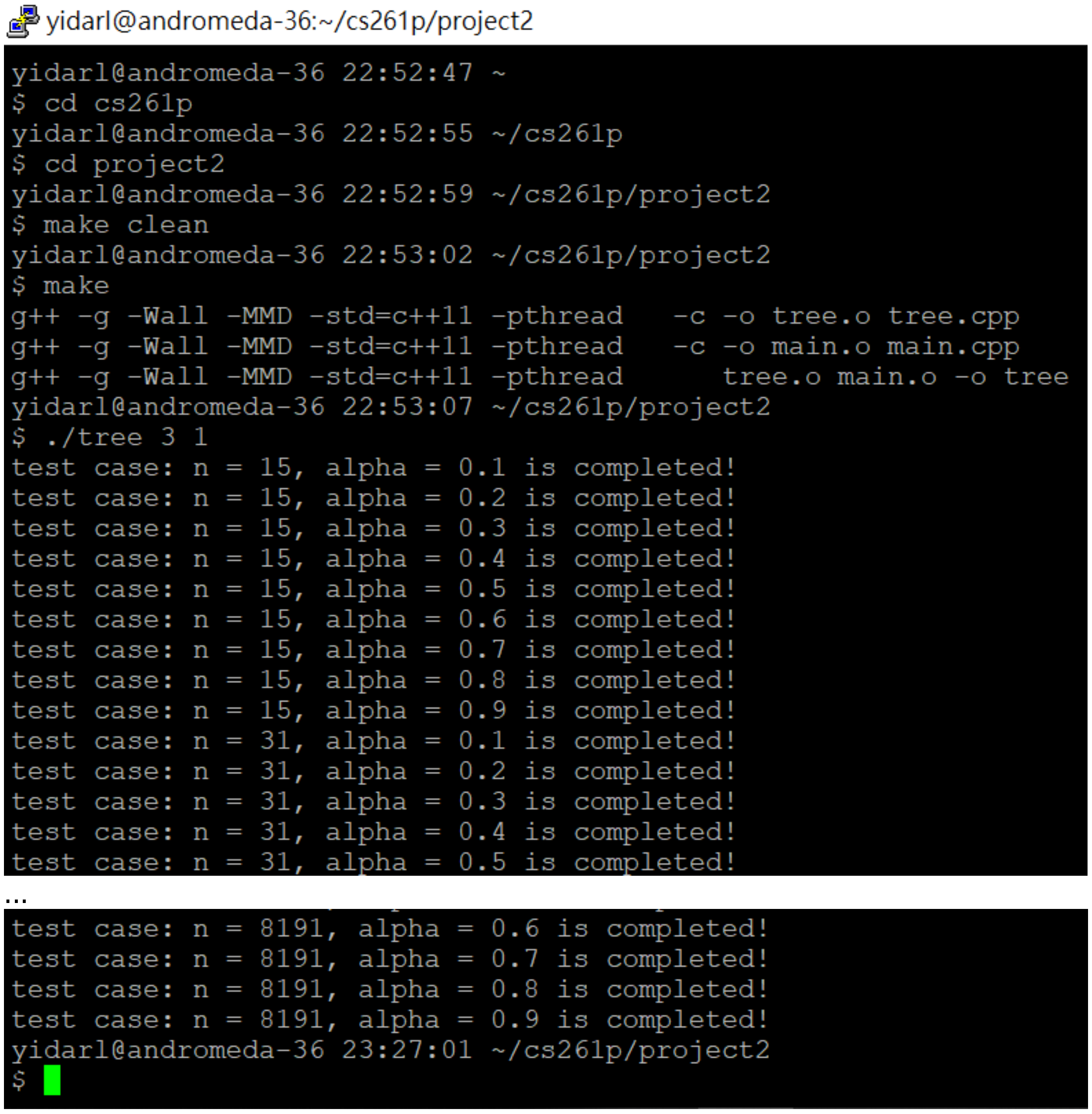
```
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(0.0,1.0);
new_node->p = distribution(generator);
```

E. COMPARATIVE EXPERIMENTAL ANALYSIS

To run our program in Linux, just put the code in a folder and use “make” command to compile the code. A program called “tree” will be generated in the folder. It takes two arguments:

- argument 1: used to annotate the test case files and the result file in case that several versions of complete tests are done
- argument 2: 1 if we would like to generate test cases, 0 if they are already available (file names must be in the form of “test_[argument 1]_i.txt”, i = 4, 5, ... , 13)

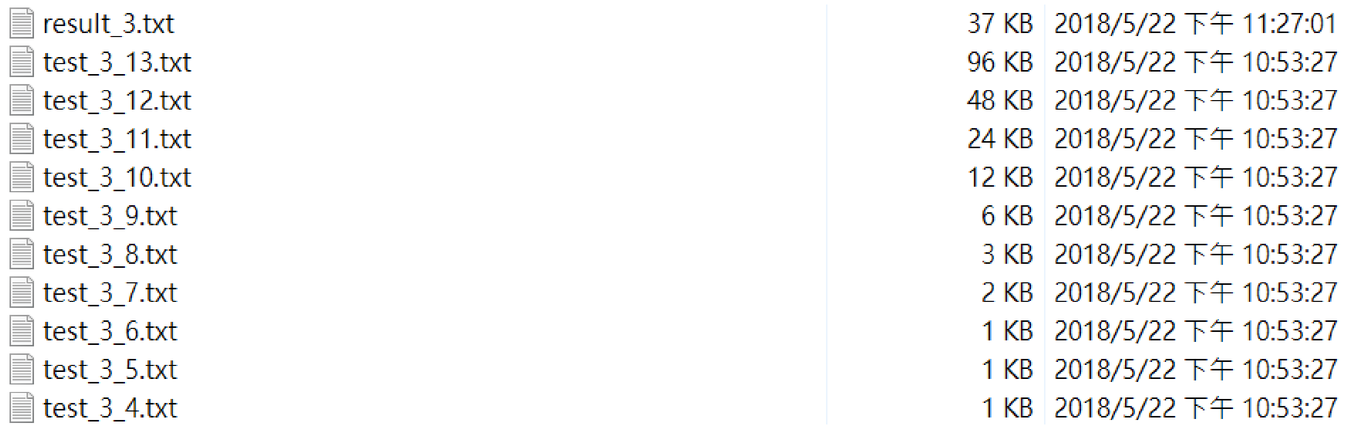
The following screenshot shows a trial run of our program in the UCI ICS openlab. Please note that it will print out a success message after completing a test case.



yidarl@andromeda-36:~/cs261p/project2

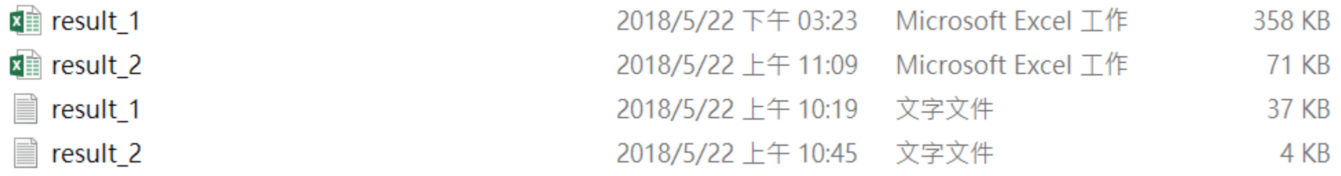
```
yidarl@andromeda-36 22:52:47 ~
$ cd cs261p
yidarl@andromeda-36 22:52:55 ~/cs261p
$ cd project2
yidarl@andromeda-36 22:52:59 ~/cs261p/project2
$ make clean
yidarl@andromeda-36 22:53:02 ~/cs261p/project2
$ make
g++ -g -Wall -MMD -std=c++11 -pthread   -c -o tree.o tree.cpp
g++ -g -Wall -MMD -std=c++11 -pthread   -c -o main.o main.cpp
g++ -g -Wall -MMD -std=c++11 -pthread      tree.o main.o -o tree
yidarl@andromeda-36 22:53:07 ~/cs261p/project2
$ ./tree 3 1
test case: n = 15, alpha = 0.1 is completed!
test case: n = 15, alpha = 0.2 is completed!
test case: n = 15, alpha = 0.3 is completed!
test case: n = 15, alpha = 0.4 is completed!
test case: n = 15, alpha = 0.5 is completed!
test case: n = 15, alpha = 0.6 is completed!
test case: n = 15, alpha = 0.7 is completed!
test case: n = 15, alpha = 0.8 is completed!
test case: n = 15, alpha = 0.9 is completed!
test case: n = 31, alpha = 0.1 is completed!
test case: n = 31, alpha = 0.2 is completed!
test case: n = 31, alpha = 0.3 is completed!
test case: n = 31, alpha = 0.4 is completed!
test case: n = 31, alpha = 0.5 is completed!
```
...
```
test case: n = 8191, alpha = 0.6 is completed!
test case: n = 8191, alpha = 0.7 is completed!
test case: n = 8191, alpha = 0.8 is completed!
test case: n = 8191, alpha = 0.9 is completed!
yidarl@andromeda-36 23:27:01 ~/cs261p/project2
$ 
```

The program will generate test cases based on the aforementioned experiment settings (if argument 2 is 1), import them, perform the experiments, and export the test results to a text file called “result_[argument 1].txt”. As can be seen from the screenshot, the execution time of the program is around 30 minutes. The following files are the outputs of the previous trial run:



result_3.txt	37 KB	2018/5/22 下午 11:27:01
test_3_13.txt	96 KB	2018/5/22 下午 10:53:27
test_3_12.txt	48 KB	2018/5/22 下午 10:53:27
test_3_11.txt	24 KB	2018/5/22 下午 10:53:27
test_3_10.txt	12 KB	2018/5/22 下午 10:53:27
test_3_9.txt	6 KB	2018/5/22 下午 10:53:27
test_3_8.txt	3 KB	2018/5/22 下午 10:53:27
test_3_7.txt	2 KB	2018/5/22 下午 10:53:27
test_3_6.txt	1 KB	2018/5/22 下午 10:53:27
test_3_5.txt	1 KB	2018/5/22 下午 10:53:27
test_3_4.txt	1 KB	2018/5/22 下午 10:53:27

In our case, argument 1 is 1 (the order of keys is random) and 2 (the order of keys is sequential). We imported the resulting “result_1.txt” and “result_2.txt” into spreadsheets and drew plots with Excel built-in tools. The following screenshot shows the program outputs and our spreadsheets used to generate the plots in this section.



result_1	2018/5/22 下午 03:23	Microsoft Excel 工作	358 KB
result_2	2018/5/22 上午 11:09	Microsoft Excel 工作	71 KB
result_1	2018/5/22 上午 10:19	文字文件	37 KB
result_2	2018/5/22 上午 10:45	文字文件	4 KB

[1] mentions that the realistic activity ratio is 0.2, which means generally update accounts for 20% of operations and search the remaining 80%. Therefore, when we show a plot of costs against varying tree sizes, the activity ratio is fixed at 0.2. On the other hand, when we show a plot of costs against varying activity ratios, the tree size is fixed at $2^{12} – 1 = 4095$, which is the same as the setting of [1]. However, different combinations of tree size and activity ratio are also possible given the data we have.

Individual Results of Trees of Stable Size

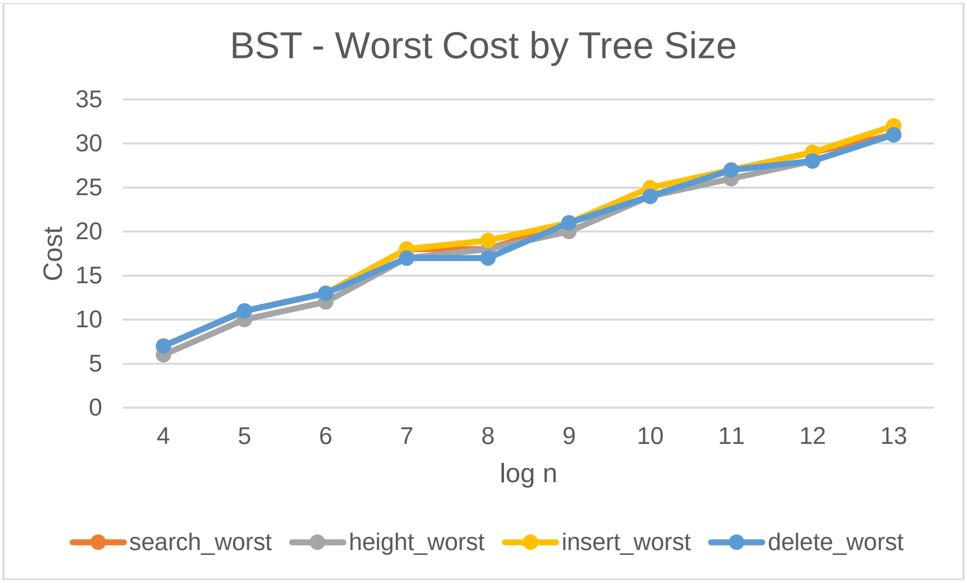
This part shows each algorithm’s worst and average operation costs under different tree sizes. We focus on whether they are consistent with our theoretical predictions in “C. THEORETICAL ANALYSIS OF RUNNING TIMES.”

Notation:

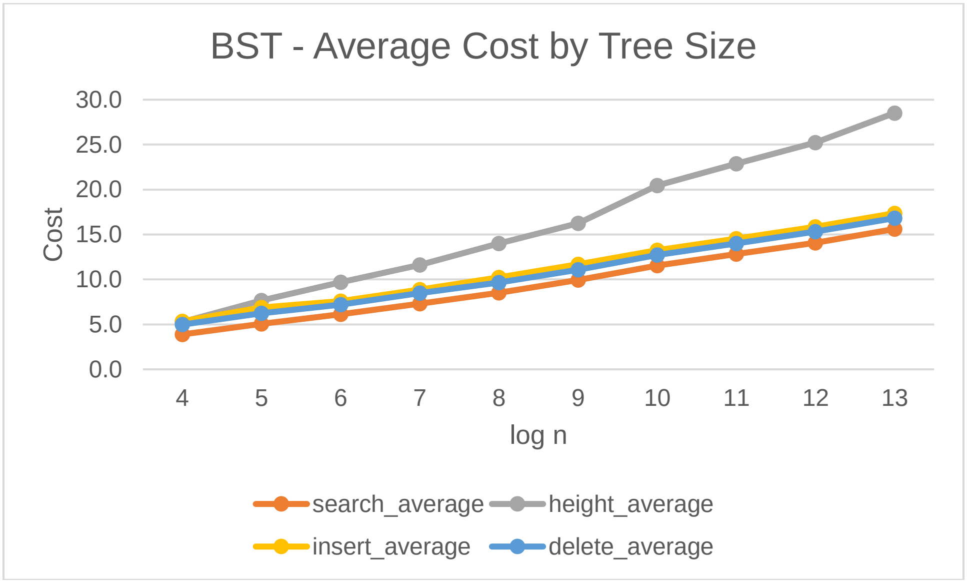
- n: tree size
- activity ratio (alpha) = number of deletions and insertions / (number of deletions and insertions + number of searches)
- search_average: average cost of a successful search
- height_average: average tree height after each operation in a tree of stable size
- insert_average: average cost of inserting a key
- delete_ average: average cost of deleting a key
- search_worst: worst cost of a successful search
- height_worst: worst tree height after some operation in a tree of stable size
- insert_worst: worst cost of inserting a key
- delete_worst: worst cost of deleting a key

<u>BST: (activity ratio = 0.2)</u>

Because the order of keys is random in the experiment of Trees of Stable Size, BST becomes Random BST in this case.

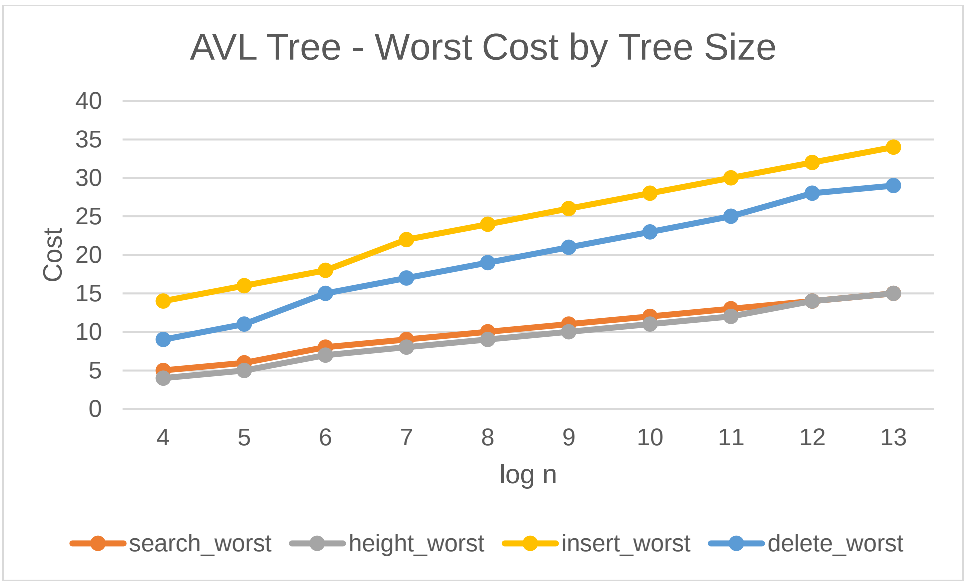


As expected, when the order of keys is random, the worst cost for every operation is O(log n), where the constant factor is estimated to be 2.3.

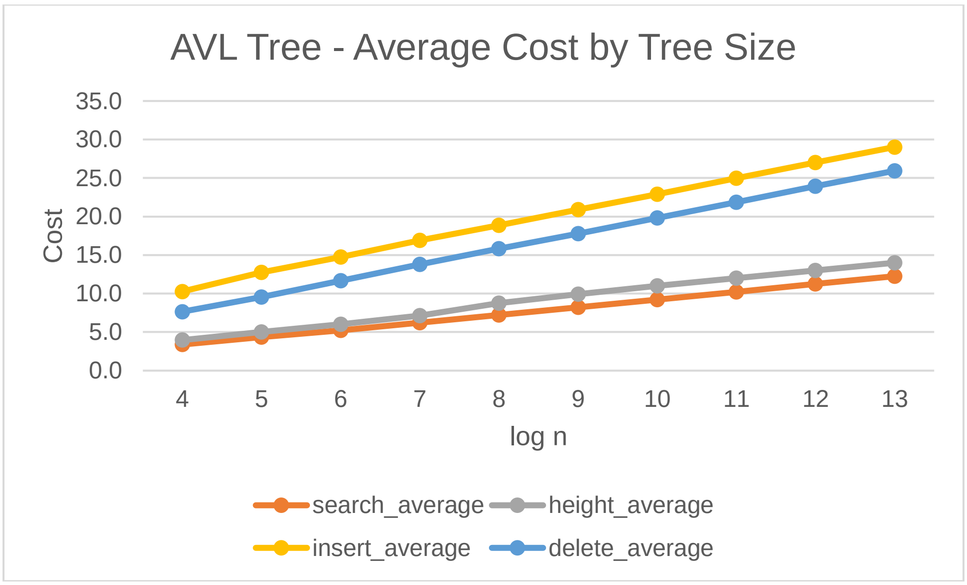


If the order of keys is random, the average tree height is between 1.3 and 2.2 times log n, while all operation costs are close to or a little more than log n on average. Although they are quite close, it is still apparent that search cost < deletion cost < insertion cost. Sometimes deletion requires searching for the inorder successor, so its cost is higher than search cost; insertion is involved with traversing the tree from the root to leaf, so it is the most expensive operation.

AVL Tree: (activity ratio = 0.2)

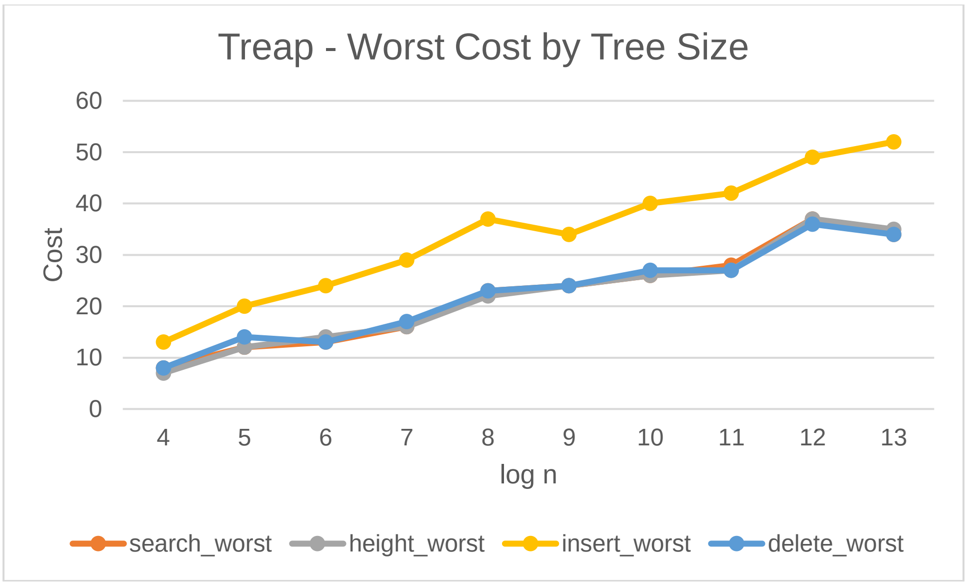


As expected, the worst search cost is almost equal to the worst tree height, which is log n. Besides, both insertion and deletion costs are more than twice of height. The differences should be rotation costs.

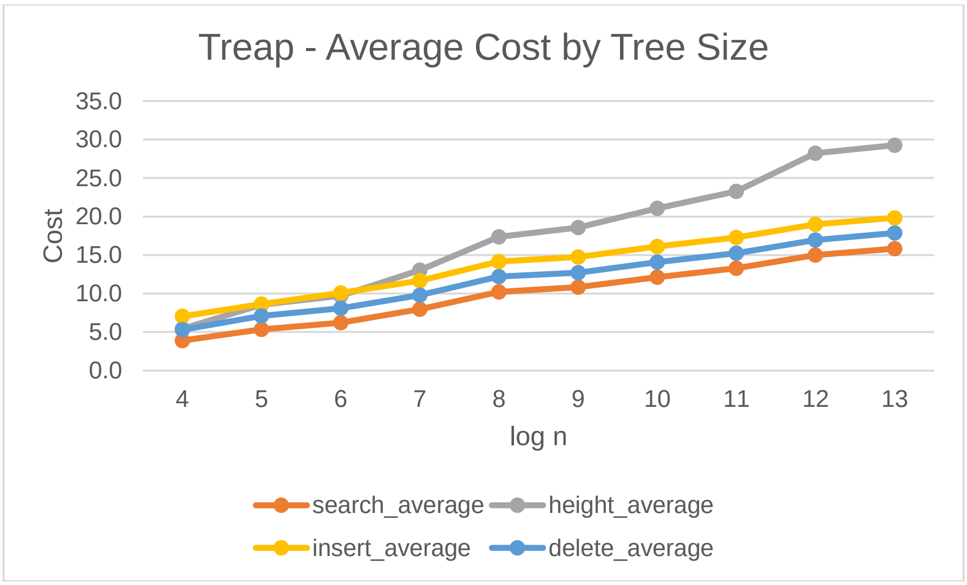


As expected, since AVL Tree is always balanced, its average-case plot is similar to its worst-case plot except that average search cost is less than average tree height. It is worth noticing that the gap between insertion cost and twice of tree height and the gap between deletion cost and twice of search cost remain stable when tree size increases. It shows that expected rotation cost is O(1) in both insertion and deletion.

<u>Treap: (activity ratio = 0.2)</u>

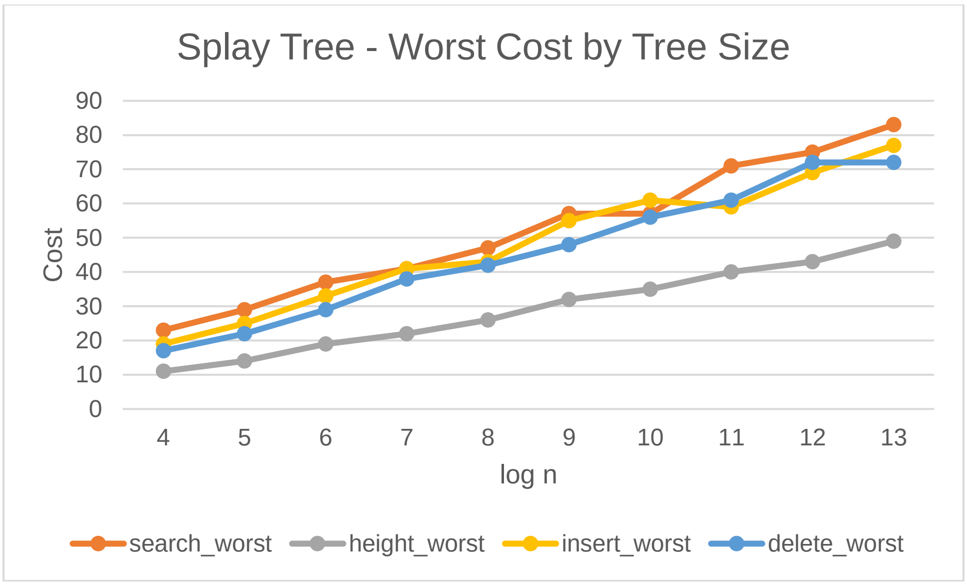


All operations' worst costs are proportional to log n. Worst costs of search and deletion are almost the same as worst tree height, and insertion is the most expensive operation.

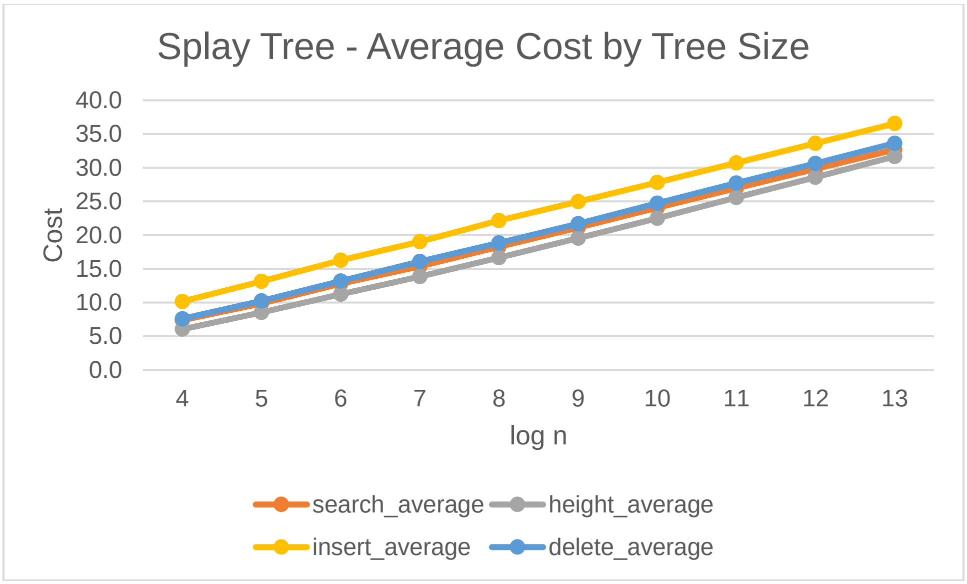


As expected, all operations' average costs are proportional to log n. It is apparent that insertion cost never exceeds tree height plus 2. Moreover, the gap between search cost and deletion cost remains between 1.4 and 2 when tree size increases, and so does the gap between deletion cost and insertion cost. The results corroborate the fact that the expected number of rotations in update is at most 2.

Splay Tree: (activity ratio = 0.2)



All operations' worst costs are proportional to log n and are almost twice as much as worst tree height.

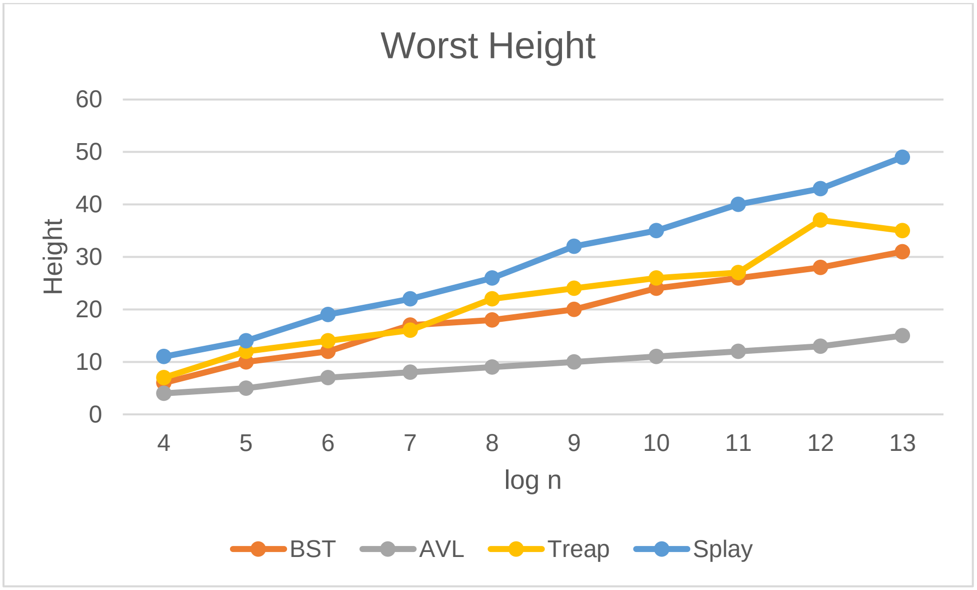


In contrast to the worst case, average operation costs are only slightly more than average tree height, which is proportional to log n. It demonstrates the power of "amortization."

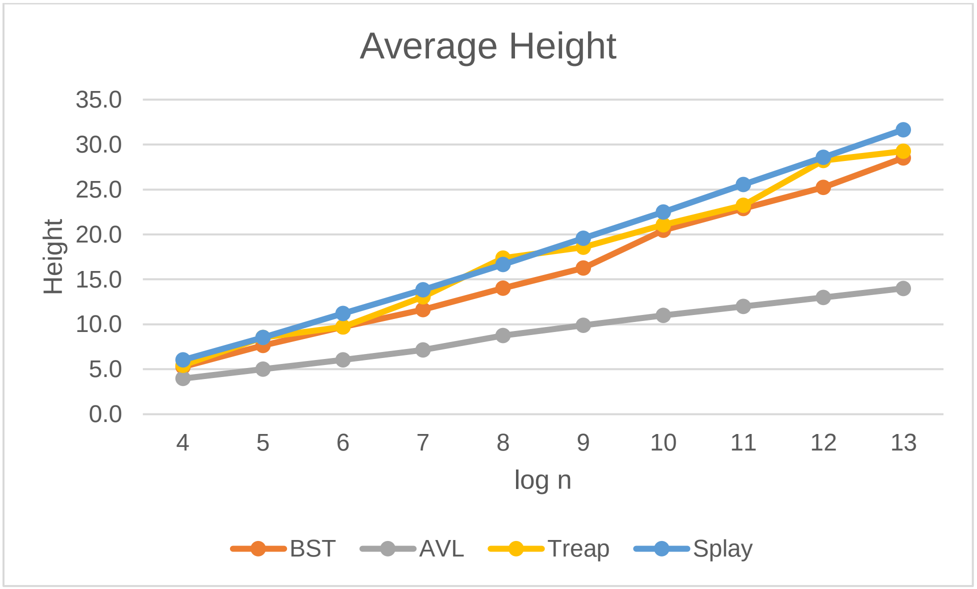
Trees of Stable Size

In this part, we compare the relative efficiencies of the algorithms as a function of tree size. All graphs are based on activity ratio = 0.2, which means 10% of operations are delete, another 10% insert, and the remaining 80% are search. During all operations, tree size remains at n±1.

Height: (activity ratio = 0.2)

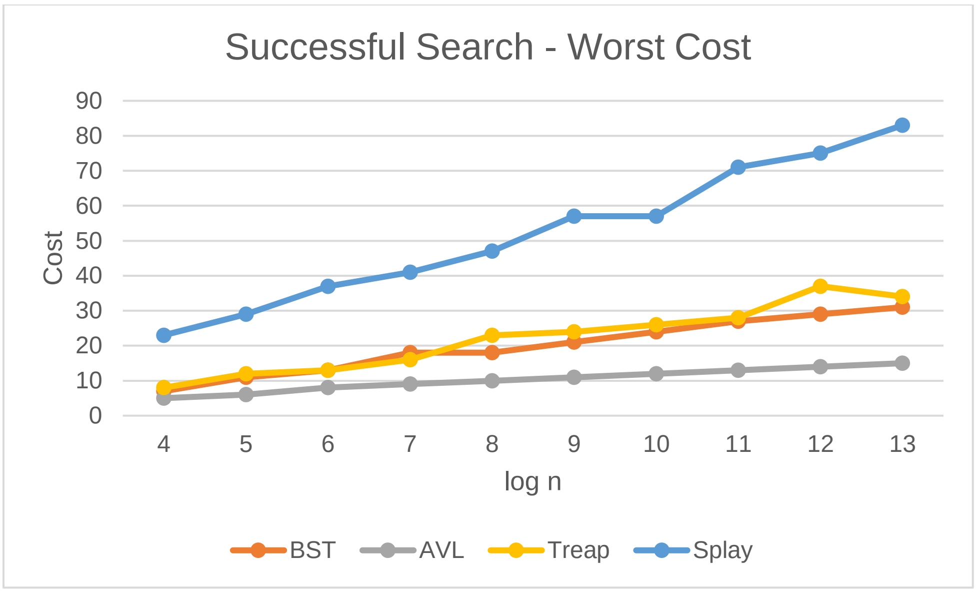


As expected, worst heights of BST and Treap are close, and AVL Tree has the minimum worst height of log n. Splay Tree has the worst worst-case height.

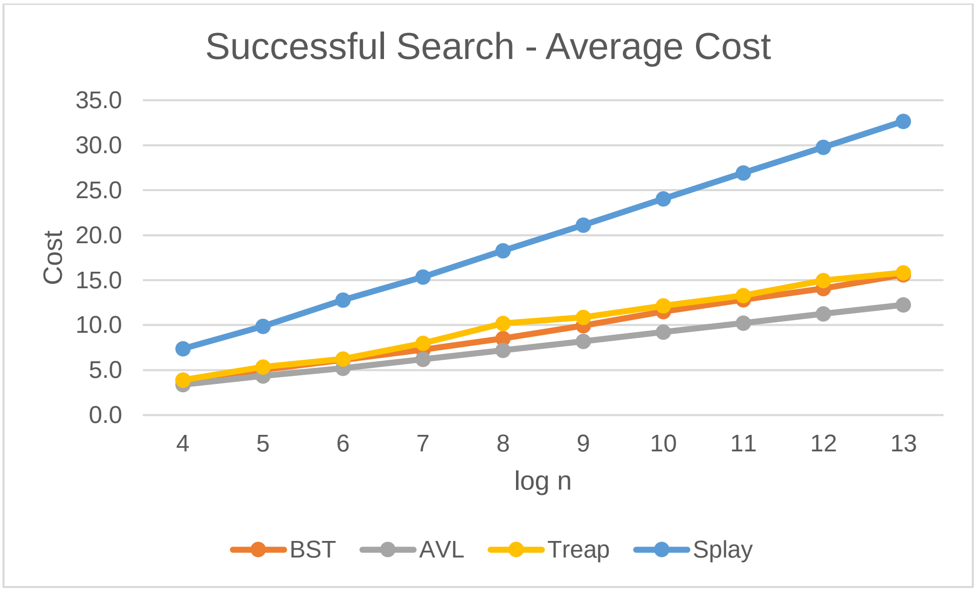


Despite its poor performance in worst height, Splay Tree has average height comparable to those of BST and Treap due to the amortization effect. Generally, AVL Tree has about half height of the other three trees.

<u>Successful Search: (activity ratio = 0.2)</u>

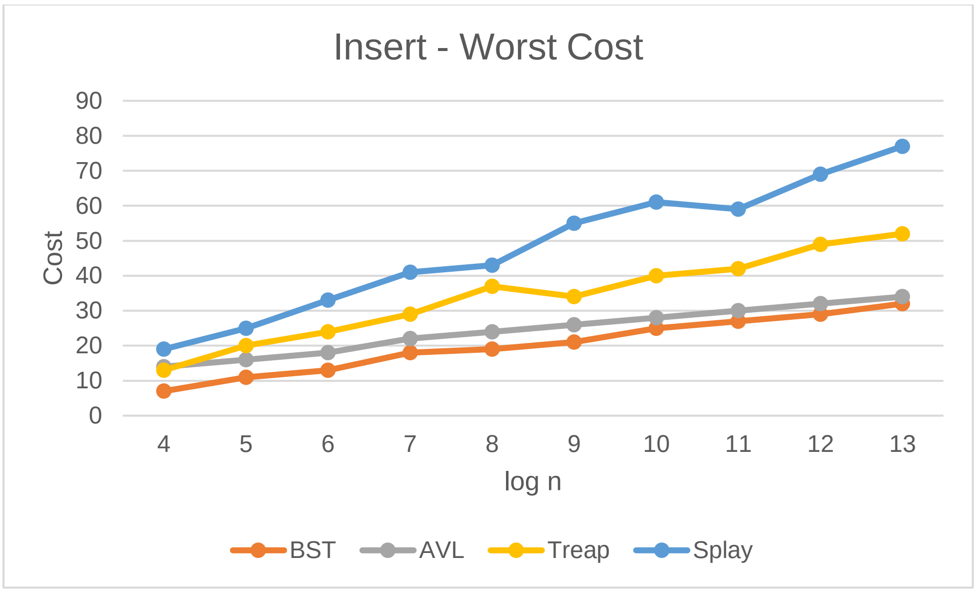


The result is similar to that of height, except that the gap between Splay Tree and the others widens. It is because Splay Tree splays the query node in every search, while the others do nothing but traverse the tree. Therefore, Splay Tree has more extra cost in this operation.

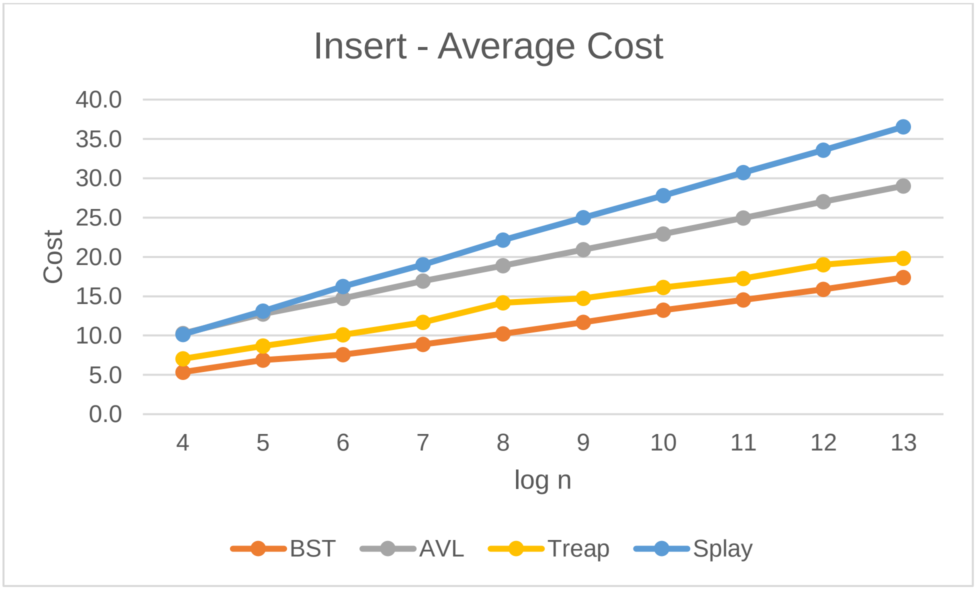


The plot is similar to above, except that the leading margin of AVL Tree over BST and Treap becomes smaller.

<u>Insert: (activity ratio = 0.2)</u>

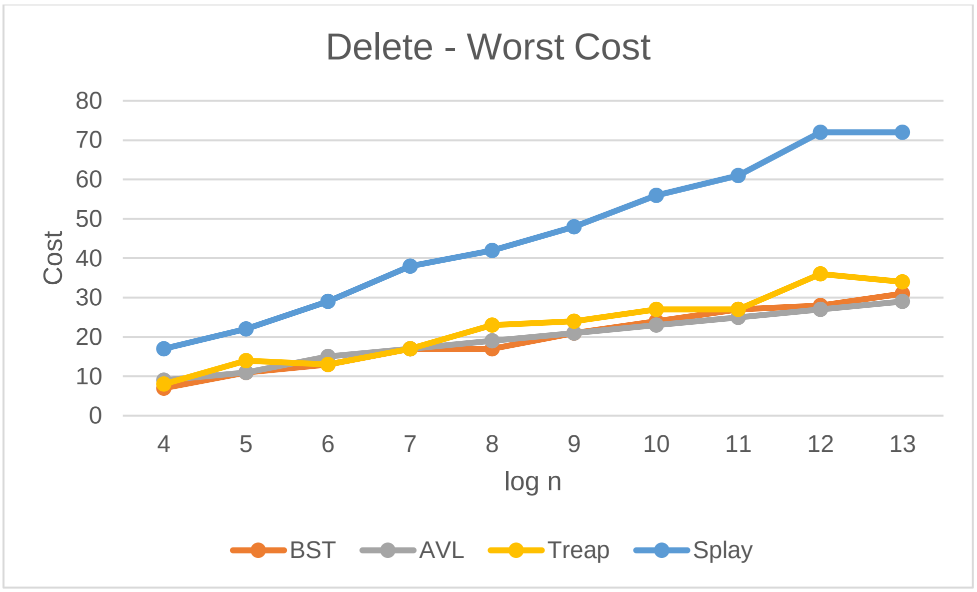


Surprisingly, BST is the best performer under all tree sizes. It shows that when keys are inserted in random order, simple operations are the most efficient. Although AVL Tree can maintain its height at minimum, the cost of updating node heights and rotations offset the benefits and let it lose the game to naive BST. Treap is worse than AVL Tree because sometimes its height may be quite large. Splay Tree is still a loser because its insertion is involved with taking a round-way trip from the root to leaf.

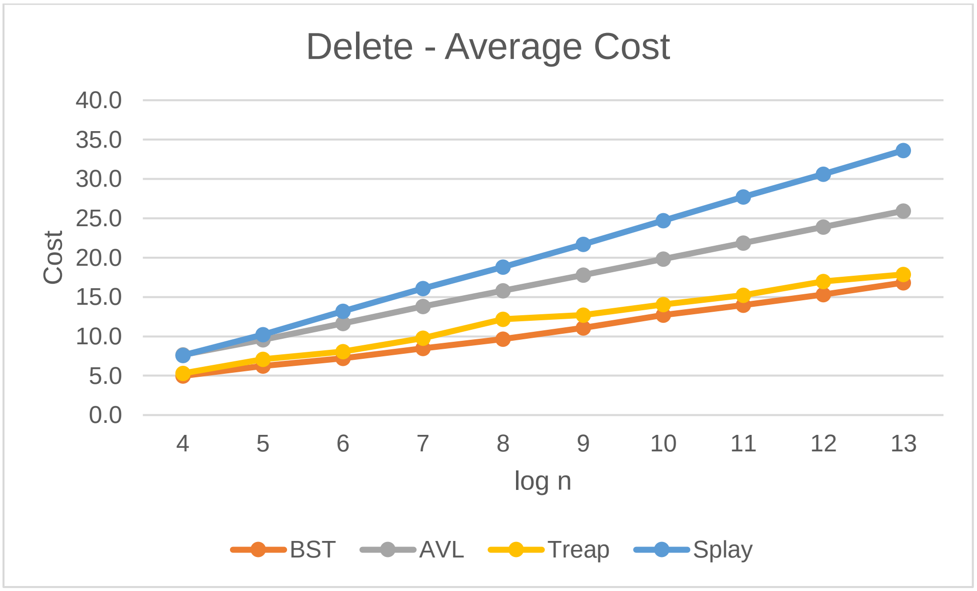


It is clear that BST < Treap < AVL Tree < Splay Tree. Treap's insertion cost is higher than that of BST due to intermittent rotations. AVL Tree's insertion cost is higher than that of Treap because it has additional update cost, which cancels the gain from minimal search cost. Splay Tree is still the worst for the same reason mentioned above.

Delete: (activity ratio = 0.2)



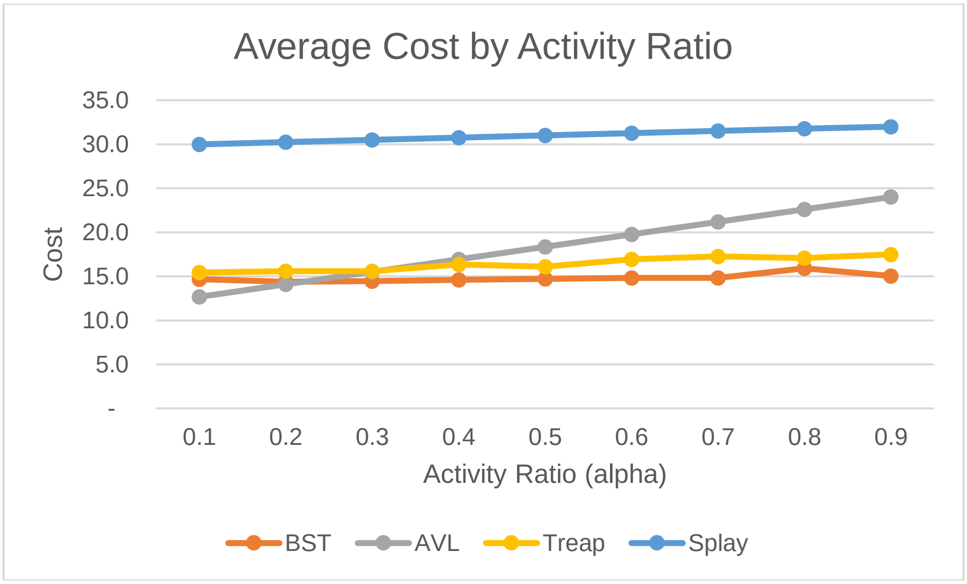
BST, AVL Tree, and Treap’s worst deletion costs are intertwined. It may be because AVL Tree has about half height of BST and Treap, but also has double costs (traversing cost and update cost). Splay Tree still has the worst worst-case deletion cost.



The result is analogous to that of the average insertion cost.

Running Time by Activity Ratio: (n = 4095)

In this part we compare the algorithms' relative efficiencies under different relative frequencies of update and search operations. The average cost is obtained by summing up all operations' costs and then dividing it by the total number of operations (3n*20).

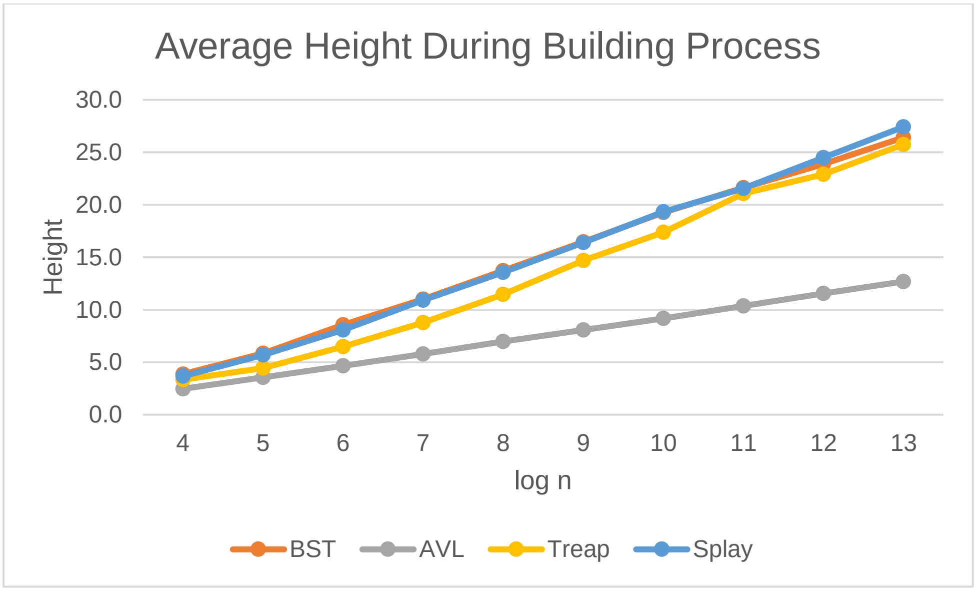


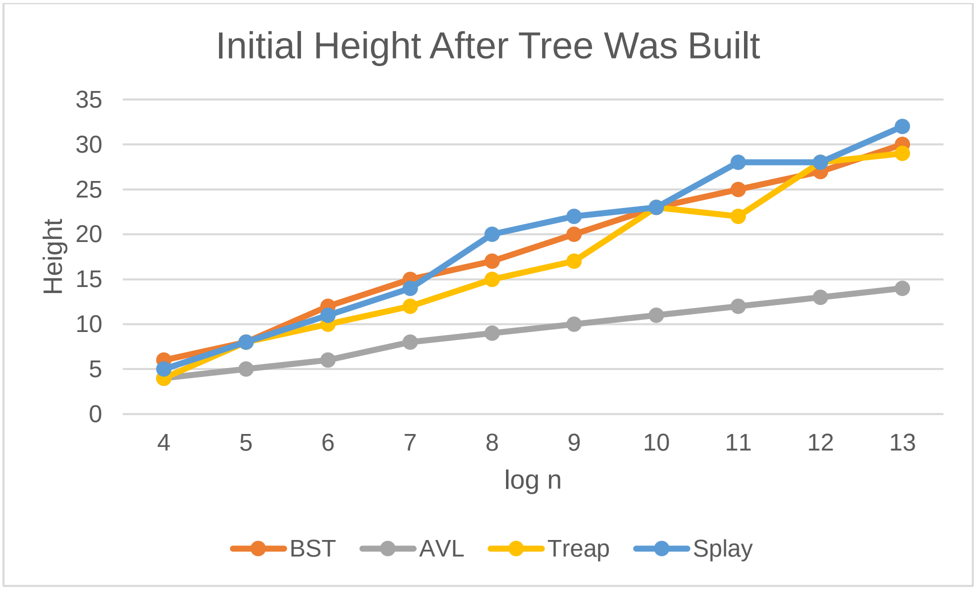
First, we observe that Splay Tree is the worst performer under all situations. It is because it requires splays in all operations, while AVL Tree and Treap carry out rotations only in the case of update. Second, we notice that AVL Tree's average cost is proportional to activity ratio, whereas the others remain relatively unchanged as activity ratio increases. It is because AVL Tree's update cost is more than twice of its search cost. In a static environment (update accounts for at most 20% of operations), AVL Tree performs the best. However, once the activity ratio ≥ 0.3, BST and Treap have lower average costs than AVL Tree, and BST's cost is even lower than that of Treap. It suggests that there is no free lunch: keeping balanced in a dynamic environment is expensive.

Trees of Growing and Shrinking Size

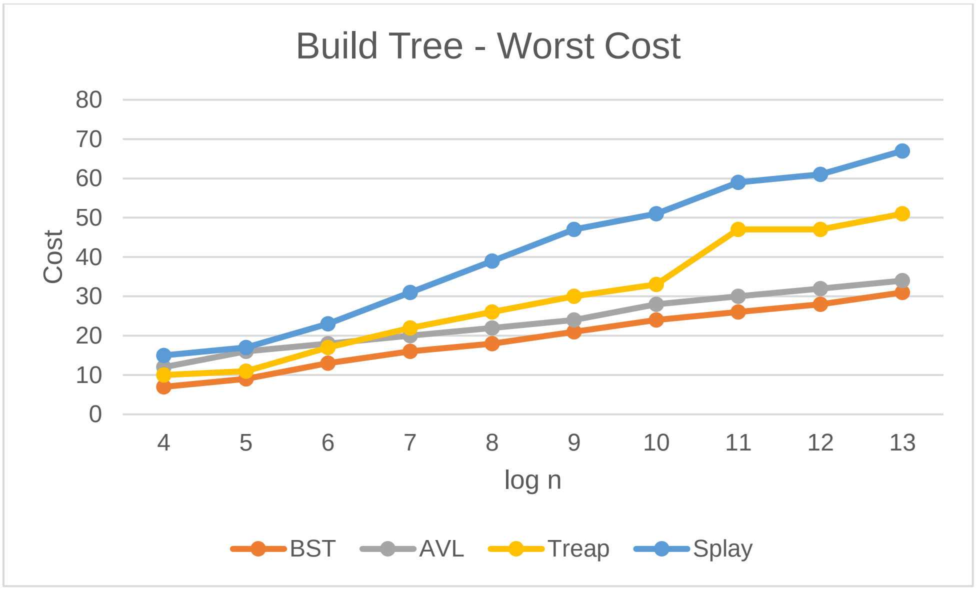
In this part we compare the algorithms' costs associated with building and destroying a tree, that is, given n keys, continuously inserting them into the tree and then continuously deleting them from the tree. The cost is on per operation basis.

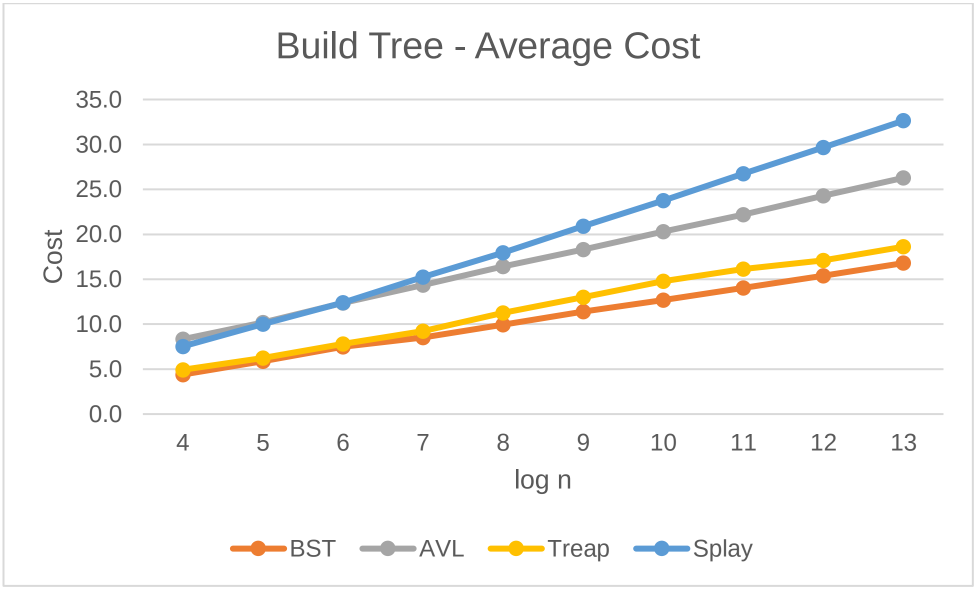
Insertion order is random:





As expected, AVL Tree has the minimum height, while the other binary trees' heights do not differ too much from one another.

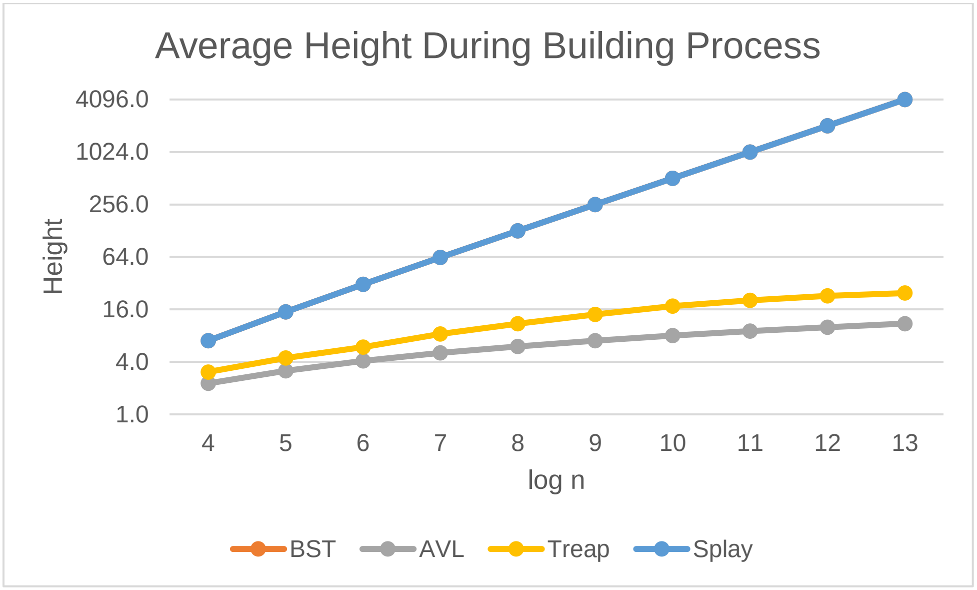


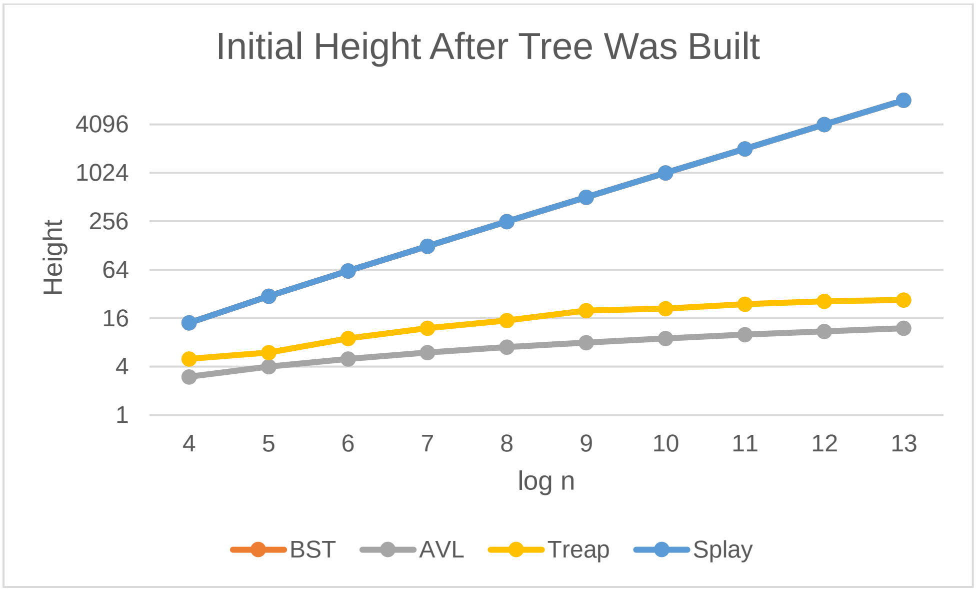


The results are very similar to those of stable tree size. The only difference is that because the costs are averaged over increasing tree sizes from 1 to n, they are lower.

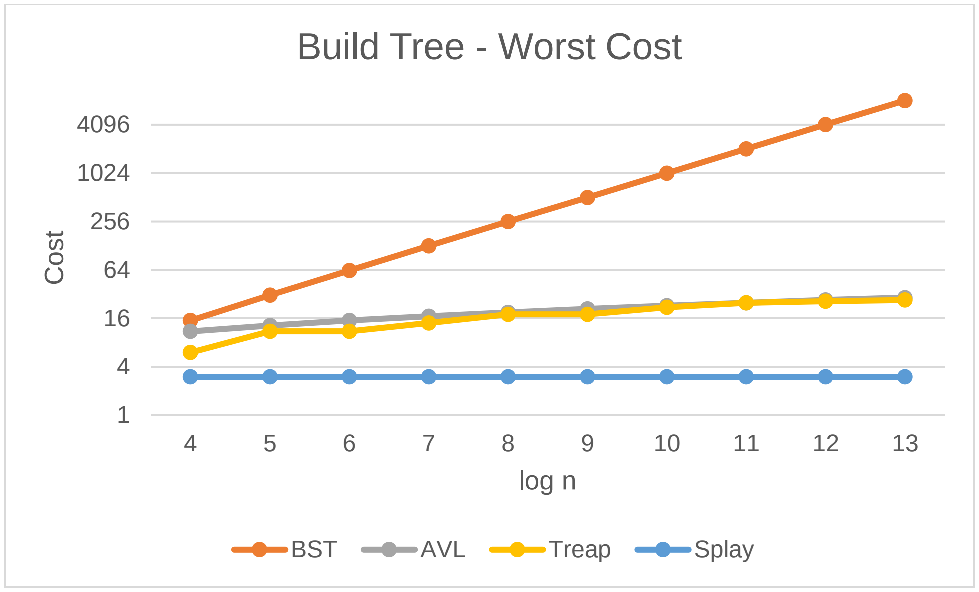
<u>Insertion order is sequential:</u>

Next, we would like to examine that if the inserted keys are ordered per se (presumably the worst case for BST), how things will change.

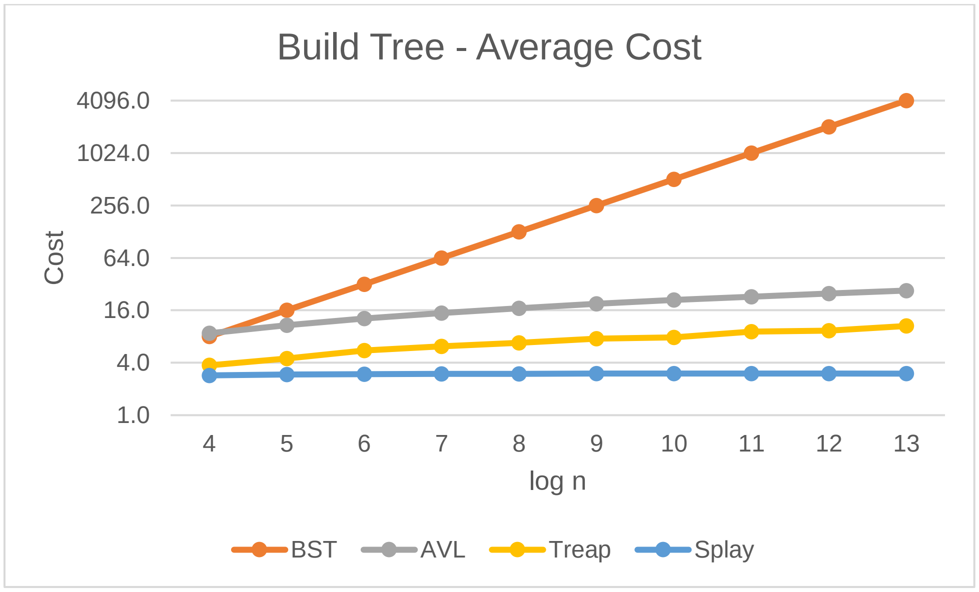




Please note that y-axis is in logarithm scale. BST line is invisible because it is covered by Splay Tree line. As expected, BST has the worst height of n. However, we find that Splay Tree can also have the same bad behavior. AVL Tree still has the lowest height, i.e. log n, in this extreme case. Treap performs quite well even though it adopts a randomized approach, as opposed to AVL Tree's deterministic balancing approach.



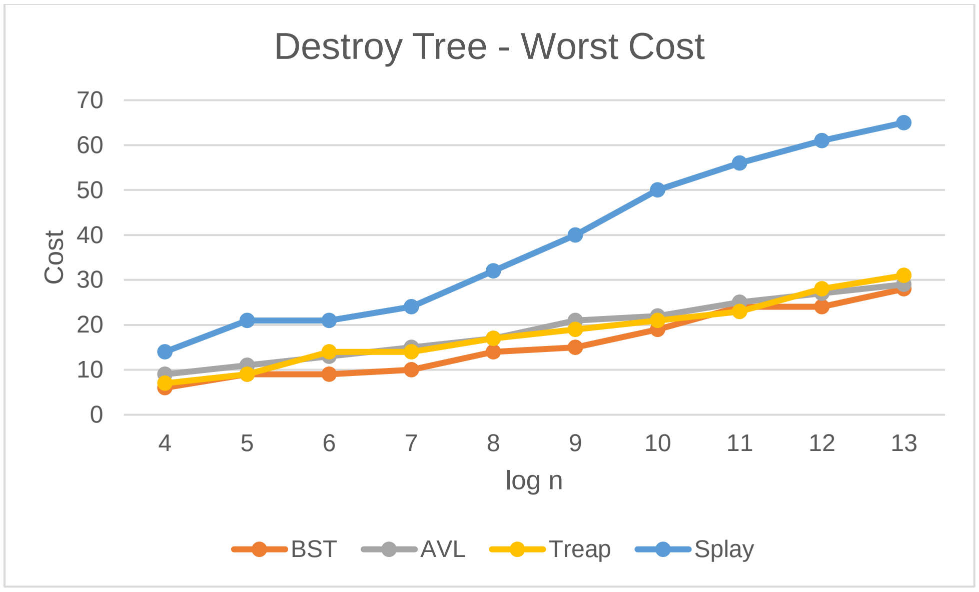
Please note that y-axis is in logarithm scale. As expected, BST has the worst-case cost of n. Surprisingly, although the resulting tree of Splay Tree has a height of n, its worst-case cost is the lowest among all algorithms. It should be attributed to the order of keys. If it is sequential, next key to insert is very likely to be close to the key just inserted. Because Splay Tree has splayed the previous node to the root, next insertion point (could be at leaf of a shallow subtree) will also be near the root. For the same reason, the number of rotations required to splay the new node should be minimal. In this item, AVL Tree and Treap perform almost equally well.

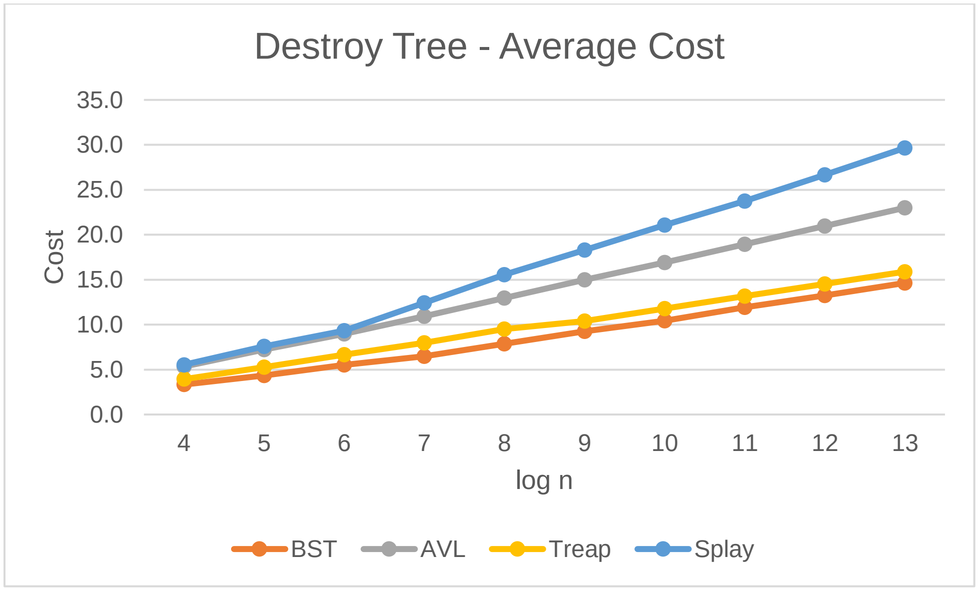


Please note that y-axis is in logarithm scale. The plot is very similar to the previous one except that the superiority of Treap over AVL Tree is unequivocal. It reveals that the cost of updating node heights does take a toll on AVL Tree’s performance.

Next, we look at deletion cost of destroying a tree.

Deletion order is random:

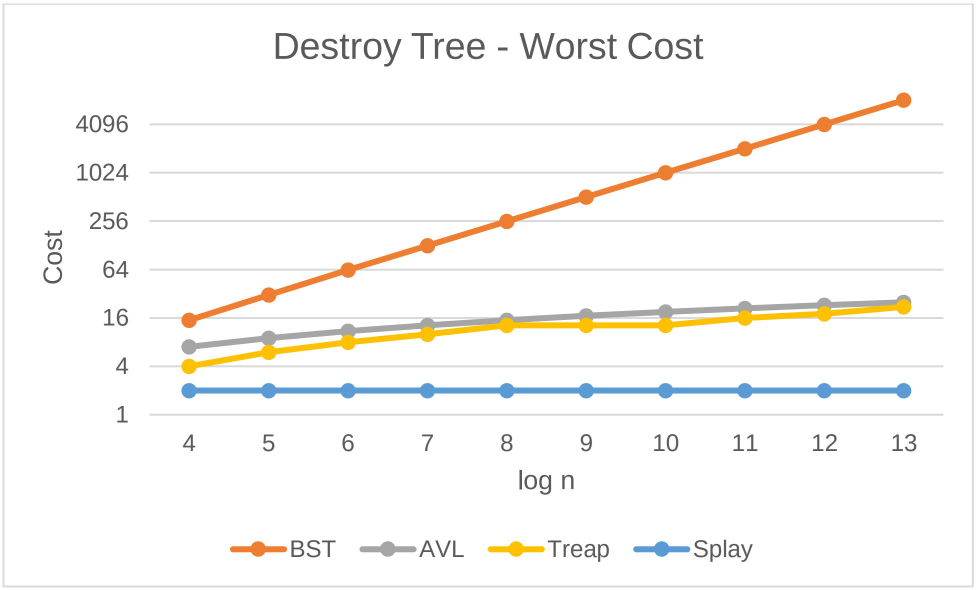




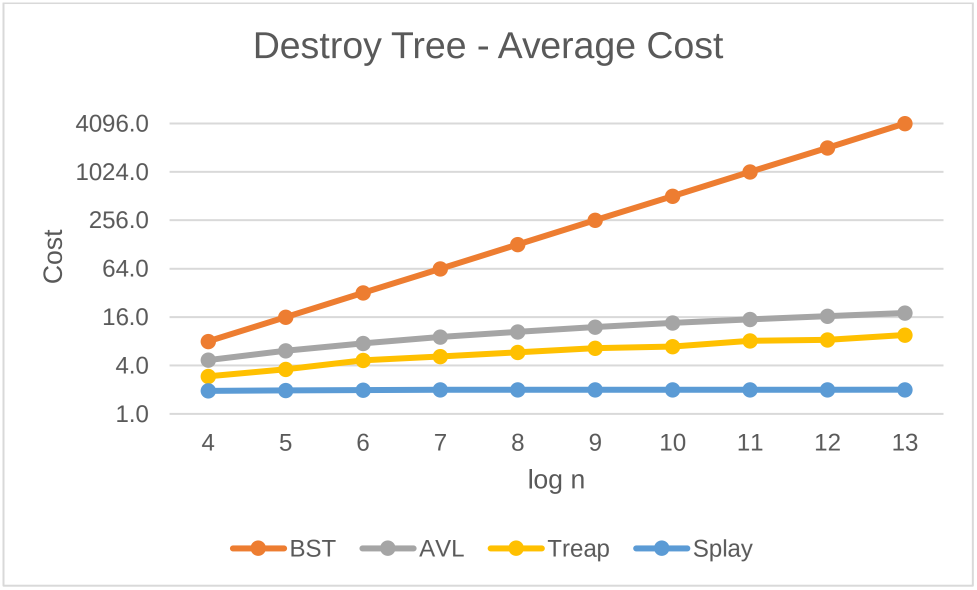
The results are very similar to those of stable tree size. The only difference is that because the costs are averaged over decreasing tree sizes from n to 1, they are lower.

<u>Deletion order is sequential:</u>

We find that deletions in the increasing or decreasing order will produce completely different results. Deleting from the maximum will favor Splay Tree, and deleting from the minimum will favor BST. Here we adopt the former because we wonder if things can take a turn for Splay Tree under such ideal situation.



Please note that y-axis is in logarithm scale. When we choose to delete the most recently inserted key, Splay Tree steals the show in this contest. Because the most recently inserted key is at the root, it takes Splay Tree only O(1) time to find the key regardless of tree size. On the contrary, for BST, each key to delete is at the end of the linked list, so it is not surprising that its worst deletion cost is n.



Please note that y-axis is in logarithm scale. The result is very similar to above except that the superiority of Treap over AVL becomes more apparent. It is because the former avoids the cost of updating node information in each deletion.

F. CONCLUSIONS

The following table summarizes our findings from the comparative empirical analysis:

	Advantages	Disadvantages	Suitable Situation
BST	• Simple and can be very efficient when keys are random	• Sensitive to the order of keys; in particular, when it is sequential, tree height becomes O(n)	Dynamic environment where insertion/deletion order is random
AVL Tree	• Guaranteed to have height of log n in any cases	• Need to update node heights every time the tree is modified	Static environment where search is more frequent than update (activity ratio ≤ 0.2)
Treap	• Not sensitive to the order of keys; can perform like Random BST even when the order of keys is sequential • Low maintenance cost compared with AVL Tree and Splay Tree	• Not guaranteed to be balanced; sometimes tree shape can be bad	General case winner, can perform pretty well under any order of keys and any activity ratio
Splay Tree	• Do not need to store extra information in nodes • Can perform astonishingly well when the order of keys is sequential • Can find the most recently accessed key in O(1) time	• Search is also accompanied with splays, so its search cost is particularly high • Not guaranteed to be balanced; sometimes tree shape can be bad	The order of keys is sequential or need to find frequently-accessed keys quickly

We find that Treap has the following prominent advantages over the other three algorithms:

- Can still have height of O(log n) even when the order of keys is sequential, as opposed to BST, whose height becomes O(n) in this case
- Once the location to insert or the node to delete is found, can complete the update in O(1) time, as opposed to AVL Tree, which needs to update node heights in O(log n) time
- Do not need to adjust tree shape in search, as opposed to Splay Tree, which requires O(log n) rotations unless the query key is at the root

Based on the above reasons, we believe that Treap is an efficient data structure which can be applied in most situations. Therefore, we would recommend Professor to use Treap to store UCI MCS student's final exam scores.

G. REMARKS

Some points not considered in this project include:

- Other factors which may affect running times: Our performance measure assumes that the costs of accessing a node, updating a node, and rotating a node are equal. However, their relative magnitudes should be accessing a node < updating a node < rotating a node. More delicate analysis can be performed by choosing different constants for each instruction.
- Other input variables which may affect the number of nodes accessed: In our experiment, we assume that each key has equal probability of being accessed. If some keys are accessed more frequently than others, Splay Tree may be an ideal choice. (However, [1] finds that Splay Tree's performance is still not impressive in this case.) We believe that Professor should be fair to all students and would not access a few students' information particularly often.

H. REFERENCES

1. J. Bell and G. Gupta, "An Evaluation of Self-adjusting Binary Search Tree Techniques," *Software - Practice and Experience*, 23(4), 369–382 (1993).
2. Lecture Slides and Assigned Readings