

Name: Stewart Dulaney

ID: 1545566

Problem 1:

Suppose you have two vector of integers x and y, each of which have N randomly distributed but distinct values. We want to merge x and y into a third vector z such that z has all the integers of x and y, additionally z should not have any duplicate values. For this problem we are not concerned with ordering in any of these vectors.

- a. Here is one algorithm. What is the Big-O of this algorithm?

```
void merge1(const vector<int>& x, const vector<int>& y, vector<int>& z) {
    z.clear();
    z.reserve(x.size() + y.size());
    for (int i = 0; i < x.size(); ++i)
        z.push_back(x[i]);
    for (int j = 0; j < y.size(); ++j) {
        bool duplicate = false;
        for (int i = 0; i < x.size(); ++i) {
            if (y[j] == x[i]) {
                duplicate = true;
                break;
            }
        }
        if (!duplicate)
            z.push_back(y[j]);
    }
}
```

$N \cdot 1$

$N \cdot N$

$N + N^2$

$O(N^2)$

- b. Here is another algorithm that uses a sorting function, assume that the sort function is implemented as quicksort. What is this algorithm's Big-O?

```
void merge2(const vector<int>& x, const vector<int>& y, vector<int>& z) {
    z.clear();
    z.reserve(x.size() + y.size());
    for (int i = 0; i < x.size(); i++)
        z.push_back(x[i]);
    for (int j = 0; j < y.size(); j++)
        z.push_back(y[j]);

    sort(z.begin(), z.end());

    int last = 0;
    for (int k = 1; k < z.size(); k++) {
        if (z[last] != z[k]) {
            last++;
            z[last] = z[k];
        }
    }
    z.resize(last + 1);
}
```

$2N \log N$

$N + N + 2N \log N + 2N$

$2N \log N + 4N$

$O(N \log N)$

- c. Which algorithm performs better given the provided description of inputs?

The algorithm for merge2 performs better b/c it has $O(N \log N)$ time complexity while merge1 is $O(N^2)$. In addition, since the values in x and y are distinct we know the loops in merge1 will execute the maximum # of iterations resulting in the worst case time complexity for merge1.

- d. Suppose the input vectors are:

```
vector<int> x{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
vector<int> y{21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39};
```

How will that change your analysis done in the previous parts?

The Big-O for merge2 is $O(N^2)$ if the input is already sorted b/c it uses QuickSort. Thus, merge1 and merge2 have the same Big-O for this input and will have similar performance. You could analyze the lower order terms in order to determine which algorithm performs better more precisely.

Name: Stewart Dulaney

ID: 1545566

Problem 2:

Suppose we are working with a general tree structure where each Node can have any number of children. A Node is defined as:

```
struct Node {  
    int val;  
    vector<Node*> children;  
};
```

- a. Write a **recursive** function that counts the number of Nodes in any (sub)tree passed into this function, including the one being passed into the function. Hint: recall that when designing recursive functions you want to restrict the reach of the function as much as possible, in this case each recursive level should only handle the node that is passed to it. That implies that someone else does most of the work. In particular, as a Node I can just ask my children how many Nodes they have, add the counts my children provide, include myself, then report to my parent that total.

```
int nodeCount(Node *node) {  
    if ( node == nullptr ) {  
        return 0;  
    }  
    int count = 1; // count the current node  
    for (int i = 0; i < node->children.size(); i++) {  
        count += nodeCount( node->children[i] );  
    }  
    return count;  
}
```

- b. Write a function that returns the number of edges in a tree using the function you implemented above:

```
int edgeCount(Node *node) {  
    return ( nodeCount( node ) - 1 );  
}
```

A tree with n vertices/nodes has $n-1$ edges.

- c. Write a **recursive** function that counts the number of leaves that are in a given tree. Note that a leaf is a node with zero children.

```
int leafCount(Node *node) {  
    if ( node->children.size() == 0 ) {  
        return 1;  
    }  
    int count = 0;  
    for (int i = 0; i < node->children.size(); i++) {  
        count += leafCount( node->children[i] );  
    }  
    return count;  
}
```

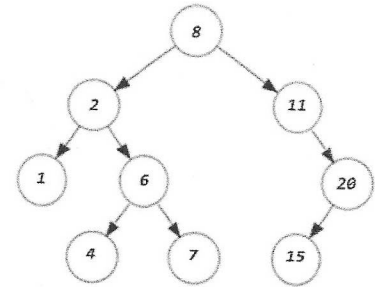
Name: Stewart Dulaney

ID: 1545566

Problem 3:

Consider the binary tree on the right:

a. What is the pre-order, post-order, in-order traversal for this tree?



pre-order: 8, 2, 1, 6, 4, 7, 11, 20, 15

post-order: 1, 4, 7, 6, 2, 15, 20, 11, 8

in-order: 1, 2, 4, 6, 7, 8, 11, 15, 20

b. What can you always say about the root node with a pre-order traversal? Post-order?

In a pre-order traversal, the root node will always be processed first.

In a post-order traversal, the root node will always be processed last.

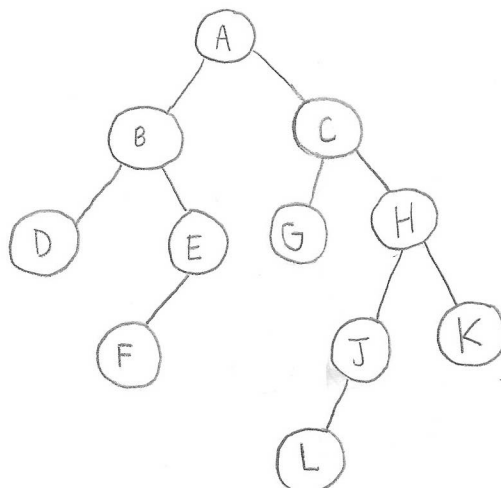
c. If given an in-order traversal and you are told which value corresponds to the root node, what can you say about the values that appear to the left of the root node's value with regards to their placement in the tree? Values that appear to the right?

The values that appear to the left of the root node's value are from the left sub-tree and the values that appear to the right are from the right sub-tree.

d. It is possible to construct a unique binary tree when given both its pre-order and in-order traversals (The same is true for post/in, but not pre/post). Given the following traversals, draw the binary tree:

Pre-order: A B D E F C G H J L K
In-order: D B F E A G C L J H K

Annotations:
Pre-order: root (A), left (B D E F), right (C G H J L K)
In-order: left (D B F E), root (A), right (G C L J H K)



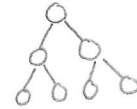
Name: Stewart Dulaney

ID: 1545566

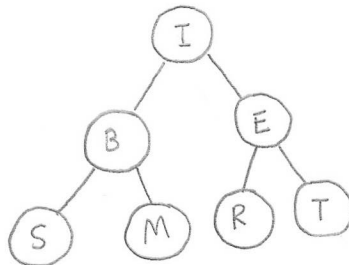
Problem 4:

- a. How many nodes does a full binary tree of height 3 have? (height of a tree being the number of nodes from the root to the deepest leaf, inclusive)

$$2^3 - 1 = \boxed{7}$$



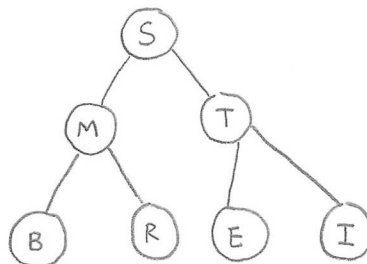
- b. Draw a binary tree of height 3 whose post-order traversal is S, M, B, R, T, E, I.



- c. What is the level-order traversal of the tree you created above?

I, B, E, S, M, R, T

- d. Draw a binary tree of height 3 whose pre-order traversal is S, M, B, R, T, E, I.



- e. What is the in-order traversal of the tree you created above?

B, M, R, S, E, T, I

Name: Stewart Dulaney

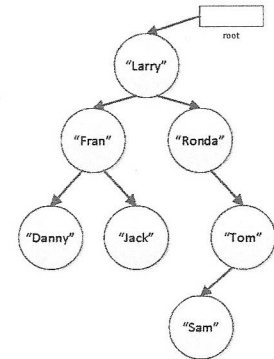
ID: 1545566

Problem 5:

Typically, we implement trees using linked-list/Node like structures because it is intuitive. However, that doesn't mean that we can't use other implementations, such as arrays. Under certain conditions it may be more efficient to do so. But, we'll encounter that later on. For now let's just consider the Binary Tree presented on the right without any other additional constraints.

Suppose we decided to encode the tree in the following way:

"Larry"	"Fran"	"Ronda"	"Danny"	"Jack"	"	"Tom"	"	"	"	"	"Sam"	"
0	1	2	3	4	5	6	7	8	9	10	11	12



One challenge in encoding any tree as an array is how to handle the lack of a child. In this case since we're storing strings we're encoding that case with an empty string.

- a. Tree traversals can be used to transfer values from a tree into linear structures like arrays. Which traversal was used to generate the array above?

Level-order traversal

- b. With the tree to obtain a child we just dereferenced the pointer to that child. We cannot do this with the current array encoding. However, because of the regular structure of binary trees (even trinary, quad, N-ary trees) we can find formulae to compute the index of each element's child given that element's index. Determine these two for the left and right child's index given the parent's index. The table provided is for your convenience. Recall that integer division truncates the result.

$\text{int leftChildIndex} = 2 * \text{parentIndex} + 1;$

$\text{int rightChildIndex} = 2 * \text{parentIndex} + 2;$

Note this calculation doesn't work for the node at index 6 b/c it seems the array given might be missing some entries. Unless I am missing something the array should be size 15 instead of 13 and then this calculation will work.

Node index	Left index	Right index
0	1	2
1	3	4
2	5	6
3	7	8
4	9	10
6	11	12

- c. One advantage that we gain here is the ability to also determine the index of an element's parent without any additional memory. Determine an equation that will compute the index of an element's parent given its own index.

$\text{int parentIndex} = (\text{childIndex} - 1) / 2;$

→ This note applies here too.

Also note this formula works for both left and right children b/c of integer division.

Name: Stewart Dulaney

ID: 1545566

Problem 6:

Consider the following open hash table implementation.

```
const int HASH_SIZE = 10;

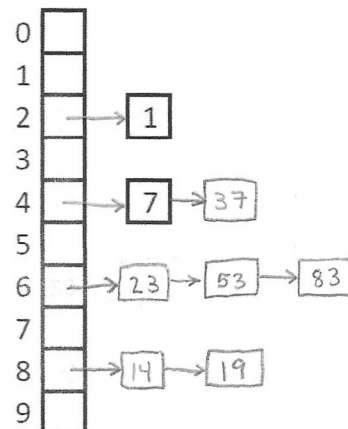
class OpenHashTable {
public:
    int hashFunc(int x) {
        return (x * 2) % HASH_SIZE;
    }

    void insert(int key) {
        int index = hashFunc(key);
        hash_array[index].push_back(key);
    }

private:
    list<int> hash_array[HASH_SIZE];
};
```

- a. Draw the state of hash_array to the right after the following calls.
The first two elements are drawn in there for you.

```
OpenHashTable oh;
oh.insert(7);
oh.insert(1);
oh.insert(23);
oh.insert(14);
oh.insert(19);
oh.insert(53);
oh.insert(37);
oh.insert(83);
```



- b. Is hashFunc() a good hash function? Why or why not?

This is not a good hash function b/c even though the hash table isn't full it has ^{already} resulted in a lot of collisions. In particular, a collision occurs when two inputs have the same digit in the one's place. However, the size of the array is also partially to blame for the high frequency of collisions. In summary, this hash function is not great b/c it does not disperse items throughout the hash array well.

Name: Stewart Dulaney

ID: 1545566

- c. Using the current state of the hash table, what is the load factor and average number of checks for this hash table?

$$L = \frac{\text{Max \# of values to insert}}{\text{Total buckets in the array}} = \frac{8}{10} = \boxed{0.8}$$

$$\text{Average \# of checks} = 1 + \frac{L}{2} = 1 + \frac{0.8}{2} = \boxed{1.40}$$

- d. How many checks on average would a closed hash table using linear probing have using the same load factor?

$$L = 0.8$$

$$\begin{aligned} \text{Average \# of checks} &= \frac{1}{2} \left(1 + \frac{1}{1-L} \right) = \frac{1}{2} \left(1 + \frac{1}{1-0.8} \right) = \frac{1}{2} (1 + 5) \\ &= \frac{1}{2} (6) = \boxed{3} \end{aligned}$$

- e. For original open hash table, how much bigger would I need to make this hash table if I wanted an average number of checks being roughly 1.05?

$$\text{Average \# of checks} = 1 + \frac{L}{2} = 1.05$$

$$\frac{L}{2} = 0.05$$

$$L = 0.10$$

$$L = \frac{\text{Max \# of values to insert}}{\text{Total buckets in the array}}$$

$$0.10 = \frac{8}{\text{Total buckets in the array}}$$

$$\text{Total buckets in the array} = \frac{8}{0.10} = \boxed{80}$$