Name: **ID:**

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]);
        }
}
```

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);
    }
```

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

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?

Name: **ID:**

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 asked 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) {
```

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

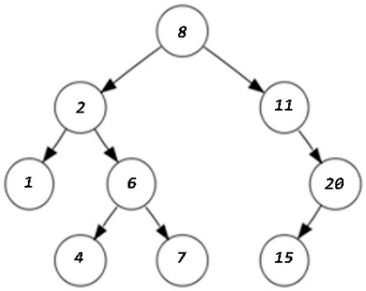
```
int edgeCount(Node *node) {
```

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) {
```

Name: **ID:**

Problem 3:
Consider the binary tree on the right:



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

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

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?

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

Name: **ID:**

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)

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?

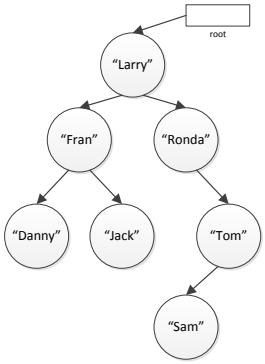
d. Draw a binary tree of height 3 whose pre-order traversal is S, M, B, R, T, E, I.

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

Name: **ID:**

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?

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.

Node index	Left index	Right index
0		
1		
2		
3		
4		
6		

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.

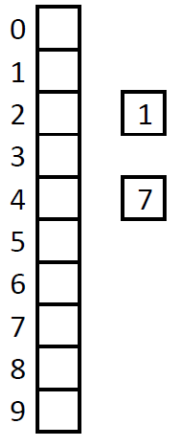
Name: **ID:**

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?

Name: **ID:**

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

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

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?