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++) {
		cout << "-----------------" << endl;
		print(z);
		if (z[last] != z[k]) {
			last++;
			z[last] = z[k];
		}
		print(z);
	}
	z.resize(last + 1);
}
```

Name: **ID:**

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};
vector<int> y{7,8,9,10,11,12};
```

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

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:

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

Name: **ID:**

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.

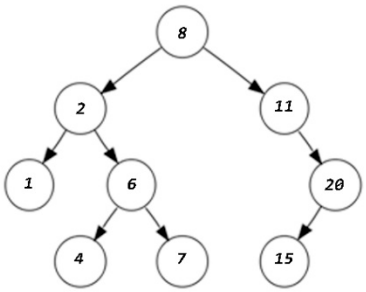
```
int leafCount(Node *node) {
```

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 2 have? (height being the number of edges from the root to the deepest leaf)

b. Draw a binary tree of height 2 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 2 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:

a. Draw a complete binary search tree with height 2 whose in-order traversal is 2,3,4,5,6,8.

b. What is the post-order traversal of this tree?

c. What is the pre-order traversal of this tree?

Problem 6:

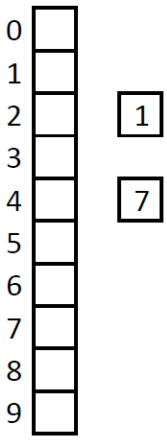
Consider the following open hash table implementation.

```
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];

};
```

Name: **ID:**

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?

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?

Name: **ID:**

Problem 7:
Consider the following array-based maxheap that has two operations, extractMax() and insert(int num).

15	10	14	7	9	8	11	4	3	5	6

a. How does the maxheap look after calling extractMax()?

b. How does it look after calling insert(12)?

c. Draw the equivalent complete binary tree of this maxheap.

Problem 8:
a. Suppose we have a maxheap which can be visualized with a ternary tree (up to three children). If we wanted to convert this to an array based maxheap, how would we find the left, middle, right child for any particular node? The parent for any particular node?

Name: **ID:**

b. What is the complexity of insertion into this maxheap? Complexity of extraction?

Problem 9:
You are hired to design a website called *brotionary.com*, a *dictionary.com* variant. You are given a list of **N** dictionary words (and their definitions) in a text file, and would like to preprocess it, such that you can readily provide information to users who visit your website and look up words. Assume that your dictionary is not going to be updated once it is preprocessed. We still want your system to "scale" -- that is, it should be able to efficiently take care of **a lot of queries** that may come in.

Assume a Word structure like the following is used to store each word and definition.

```
struct Word {
	string word;
	string definition;
};
```

a. First of all, the users should be able to look up words. Which of the following options would you take, and why?

Option A: Store the Words in a binary search tree.
Option B: Store the words in a hash table.

Name: **ID:**

b. You want to add functionality that prints words and their definitions within a certain range of **P** words. e.g all the words between and including aardvark and awkward. Which of the following options would you take, and why?

Option A: Add a sorted singly linked list with pointers to Words in the structure used in part a. Each time a range [x:y] is specified, we search for word x in the list, and then traverse the list to print each word, until we hit word y.
Option B: Add a sorted vector with pointers to Words in the structure used in part a. Each time a range [x:y] is specified, we search word x in the vector, and then traverse the vector to print each word, until we hit word y.

Name: **ID:**

c. In the right corner of the website, you want to display "**K** most popular words", where **K** is some integer, which are determined by queries that were received in the past hour, and is updated every hour. Assume queries are made on **M** distinct words, where **M** >>**K**. Which option would you take, and why?

Option A: In the beginning of every hour, create a hash table (initially empty) that stores (word, count)-pairs. Each time a query for a word x comes in, look up x in the hash table (or add a new one if one does not exist), and increase the count for x. At the end of the hour, iterate through all (word, count)-pairs in the hash table and store them in a maxheap, using their counts as the keys. Then extract **K** words from the heap.
Option B: In the beginning of every hour, create a vector (initially empty) that stores (word, count)-pairs. Each time a query for a word x comes in, look up x in the vector (or add a new one if one does not exist), and increase the count for x. At the end of the hour, sort the pairs in the vector in the decreasing order of their counts, and print the first **K** words