Section Handout 8

Problem One: Sizing Your Vocabulary

Write a function

```
int wordsIn(Node* root);
```

that takes as input a pointer to the root of a trie, then returns the number of words contained in that trie.

There are many different ways that you can represent the nodes in the trie, each of which has different strengths and weaknesses. Talk about some of those options and their strengths and weaknesses.

For the purposes of this problem, and the ones after it, you can assume that the letters used in the words in the trie are all lower-case English letters ranging from `a` to `z`.

Problem Two: My Very First Words

Write a function

```
string firstWordIn(Node* root);
```

that takes as input a pointer to the root of a trie, then returns the alphabetically first word in that trie.

If the trie contains no words, this function should report an error.

Problem Three: Aphasia

Write a function

```
void removeFrom(Node*& root, const string& word);
```

that takes as input a pointer to the root of a trie (by reference) and a word, then removes that word from the trie. This function should deallocate any memory for trie nodes that are no longer needed. Additionally, if the trie ends up empty, the function should change `root` to be `nullptr` to signal this.

Again, you will need to think about how you want to represent the nodes in your trie. Different implementations will change how this step works.

The word in question might not be in the trie.

As a hint, draw pictures.

Problem Four: Scrambled Hashes

Below are three descriptions of hash functions that can be used to produce hash codes for English words. Each of these functions has a problem with it that would make it a poor choice for a hash function. For each of the hash functions, describe what the weakness is.

- Hash function 1: Always return 0.
- Hash function 2: Return a random `int` value.
- Hash function 3: Return the sum of the ASCII values of the letters in the word.

Problem Five: Rehashing

In lecture, we mentioned that in order to keep the runtime of a hash table low, we periodically need to ***rehash*** the elements by doubling the number of buckets and moving the elements from the old table into the new one. The `OurHashSet` type has the following private fields:

```
Vector<Vector<std::string>> buckets;
int numElems;
```

Here's the code for the insert function:

```
void OurHashSet::add(const string& value) {
    /* Determine the bucket to jump into. */
    int bucket = hashCode(value) % buckets.size();

    /* If this element is already present, we don't need to do anything. */
    for (string elem: buckets[bucket]) {
        if (elem == value) return;
    }

    buckets[bucket] += value;
    numElems++;
}
```

Implement a function `OurHashSet::rehash()` that performs a rehash, then update the above code from the `OurHashSet::add` function to rehash the table whenever the load factor (the ratio of the number of elements to the number of buckets) is two or greater.

Problem Six: A Classic Job Interview Question

Here's another classic job interview question that you are now equipped to answer. ***Useful hint: the answer to a ton of technical job interview questions is "use a hash table." I'm not kidding.***

You are given a list of integers. Write a function to find if any two of those integers add up to exactly 137. Your solution should be as fast as possible. Aim to solve it with an algorithm that, on average, runs in time O(n).

As a follow-up, consider the following problem, which is called the 3SUM problem: given a list of integers, determine whether any *three* of those integers sum up to exactly 137 (you're allowed to choose the same number multiple times if you'd like). See if you can solve this in time O(n^2).

Problem Seven: Linear Probing

There are a lot of different ways to build a hash table, each of which is designed to address collisions in a different manner. The type of hash table we used in class uses a system called ***closed addressing:*** any two items that collide with one another are put into the same bucket. There's another type of hashing called ***open addressing*** that in practice tends to be much, much faster. The simplest type of open addressing (which happens to be the very first sort of hash table ever invented!) is called ***linear probing***.

In linear probing hashing, the hash table consists of an array of table slots. To insert an element, we compute it hash code, then look in the table slot. We then do the following:

- If the slot is empty, then we just put the element into that table slot.
- If the slot is full and it holds the element we're inserting, there's nothing to do.
- Otherwise, move to the next slot in the table, wrapping around as necessary, and repeat this process.

Intuitively, you can think of linear probing hashing this way: each element has a slot that it would like to be in (the slot corresponding to its hash code), and if it can't fit there, it scoots over until it finds a free position. Similarly, to look up an element in a linear probing hash table, you'd compute the hash code for that element, then look in the slot in the table it corresponds to. From there, we'd do the following:

- If the slot is empty, then the element is definitely not present in the table.
- If the slot is full and the element there is the element in question, we've found it!
- Otherwise, move to the next slot in the table, wrapping around as necessary, and repeat this process.

Linear probing hash tables need to keep their load factors relatively low, since otherwise the table starts to have really long runs of elements that degrade lookup times. Typically, you'd pick a load factor like 0.75 and rehash if there are more than that many elements in the table.

Implement the following interface for a class that uses linear probing to implement a hash table:

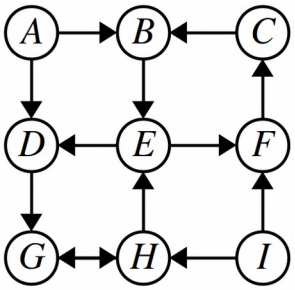
```
class LinearProbingTable {
public:
    LinearProbingTable();

    bool contains(int value) const;
    void add(int value);

    int size() const;
    bool isEmpty() const;
public:
    /* Your call! */
};
```

You can assume that all the integers that will be stored in the table are nonnegative, so you can reserve negative values as sentinels to indicate "not present." In practice, linear probing hash tables tend to *dramatically* outperform the closed-addressing system we talked about in class. Take CS166 for details!

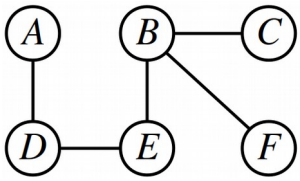
Problem Eight: Graph Searches

Below are six graphs. For each graph, say whether that graph is directed or undirected. Then, run a depth-first search and a breadth-first search starting at node *A* in each graph. When doing the depth-first search, at each point in time, if you have multiple choices of which node to visit next, choose the one that's alphabetically first. When doing the breadth-first search, similarly choose to enqueue nodes in alphabetical order if there's a tie. For the BFS, at each point show the node dequeued out of the queue and the state of the queue after processing that node.

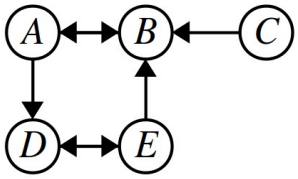
Graph 1



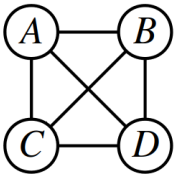
Graph 2



Graph 3

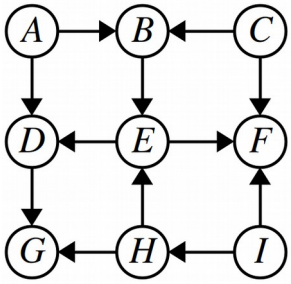


Graph 4



Problem Nine: Topological Sorting

Find two different topological orderings for the following DAG:

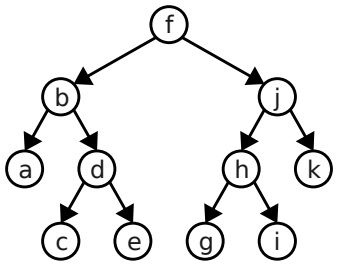


Problem Ten: Breadth-First Search and Binary Search Trees

All trees are graphs (though not all graphs are trees), so it's possible to use BFS to traverse a tree. Write a function

```
void breadthFirstSearch(Node* root);
```

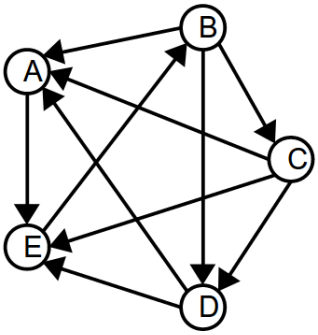
that accepts as input a pointer to the root of a binary search tree, then prints out all the nodes in the tree using a breadth-first search. What will the output of your function be on the following tree?



Under what circumstances will your function list all the nodes in a BST in sorted order?

Problem Eleven: Tournament Champions

A ***tournament*** is a graph representing contest among *n* players. Each player plays a game against each other player, and either wins or loses the game (let's assume that there are no draws). Each player is represented by a node, and each directed edge represents the outcome of a game, with the winner pointing to the loser. For example, in the tournament to the left, player A won her game against player E, but lost against players B, C, and D.



A ***tournament champion*** is a player in a tournament who, for each other player, either won her game against that player, or won a game against a player who in turn won his game against that player (or both). For example, in the graph on the left, players *B*, *C*, and *E* are tournament champions. However, player *D* is ***not*** a tournament champion, because he neither beat player *C*, nor beat anyone who in turn beat player *C*. Although player *D* won against player *E*, who in turn won against player *B*, who then won against player *C*, under our definition player *D* is ***not*** a tournament champion. *(Make sure you understand why!)*

There's a cool theorem about tournaments that says that every tournament must have at least one champion (curious why? take CS103!) Write a function

```
Set<String> championsOf(const Map<string, Set<string>>& tournament);
```

that takes as input a graph representing a tournament and returns a set of all the champions in that tournament.

Problem Twelve: Tournament Victory Chains

Here's another cool fact about tournaments (as defined above). In any tournament, there's always some way to line up all the people in the tournament so that each person won her game against the person immediately to her right. For example, in the tournament shown above, we can line up the players as *D*, *A*, *E*, *B*, *C*, since *D* beat *A*, *A* beat *E*, *E* beat *B*, and *B* beat *C*. This is sometimes called a ***victory chain***.

There's a very cool recursive algorithm for finding such a victory chain. If the tournament has no players, then the empty list of players is a victory chain. Otherwise, choose a person *p* and split the tournament into two subtournaments consisting of the players who beat *p* and the players who lost to *p*, respectively. Recursively get victory chains for those subtournaments. The overall chain can then be formed by starting with the chain for the subtournament of folks who won against *p*, then going to person *p*, then finishing with the chain for the subtournament of folks who lost against *p*. (Do you see why this works?)

Implement a function

```
Vector<string> victoryChainFor(const Map<string, Set<string>>& tournament);
```

that takes in a tournament and returns one of its victory chains.

Problem Thirteen: Eccentricities

The ***distance*** between two nodes in a graph is the length of the shortest path between them. The ***eccentriciy*** of a node in a graph is the maximum distance between that node and any other node in the graph (of the nodes that it can actually reach). Write a function

```
int eccentricityOf(const Map<string, Set<string>>& graph, const string& node);
```

that returns the eccentricity of the given node in the given graph. As a hint, think about using breadth-first search.