Here are some practice problems for the second midterm in CS10. Some of these are taken from previous terms, which might have emphasized somewhat different things, but we did cover all these.

1. We looked at separate chaining and linear probing for implementing hash tables. What are the relative advantages and disadvantages of each?

2. Make up a small example of a set of Strings and a hash function that yield collisions, and show what happens in both a "separate chaining" hash table and an "open addressing" hash table in your scenario.

3. Consider a hash table that uses an array of 11 elements and a hash function $h(i) = i \bmod 11$. Give a list of 4 different numbers that would map to two different buckets in chaining and one big cluster in linear probing. Describe a simple modification to the hash function that would spread them all out.

4. Robocallers-R-Us needs your help. They have a map from people to 9-character phone numbers, and a set of area codes to target. Now they want a map from area codes to the people in them. For example:

   person-to-phone: { "Alice" $\mapsto$ "765-555-1234", "Bob" $\mapsto$ "650-555-9999", "Charlie" $\mapsto$ "765-555-1111", "Denice" $\mapsto$ "650-555-6666", "Elvis" $\mapsto$ "901-555-0000", ... }
   area-codes-of-interest: { "765", "650", ... }
   $\Rightarrow$
   area-code-to-people: { "765" $\mapsto$ { "Alice", "Charlie", ... }, "650" $\mapsto$ { "Bob", "Denice", ... }, ... }

   Declare and write a method that takes a person-to-phone map and an area codes set, and returns an area-code-to-people map. Assume that `phone.substring(0,3)` will correctly extract an area code.

5. Given the heap `[-7,-3,1,12,4,14,5]`, draw the intermediate and final heaps (as arrays or trees) following these steps:
   ```
   remove();
   add(-7);
   add(-7);
   ```

6. Write a short piece of code that uses a Stack to test whether a word is a palindrome (i.e., reads the same forward and backward). For example, "a", "aha", and "abba" are palindromes, while "ah" and "abbs" are not. The code should proceed through the word once, from beginning to end.

   ```
   static boolean isPalindrome(String s) {
   ```

7. (a) Insert the following keys, in the order given, into an initially empty binary search tree, and draw the final tree.

   `10, 38, 20, 30, 15, 12, 40, 18, 35, 28, 25, 19`

   (b) Suppose that the Binary Search Tree's delete method is executed on the above tree to delete the key whose value is 20. Draw the tree at the end of this delete operation.

   (c) Consider the following class to represent a node in a Binary Search Tree:

   ```
   public class BSTnode {
     public int key;   // key stored in the node
     public BSTnode left;   // reference to left child
     public BSTnode right;   // reference to the right child
   }
   ```

Consider the following method that, given a non-null reference x to a node of a Binary Search Tree, returns the largest key in the subtree rooted at x.

```
public int getLargestKey(BSTnode x)
```

Implement the above method using recursion. You are not allowed to write or use any method other than the above method (so, no helper methods are allowed).

8. Suppose we have a box of the following kind: when you open it, you will find some cash lying and, possibly, some more boxes. When you open any of these boxes, you will similarly find some cash and some more (possibly zero) boxes. The problem that we want to solve is stated as follows: given a box b, count all of the cash in b and in the boxes nested inside b. To capture this problem, I have defined the Box class below. The instance variable cash is the money that you will readily see up on opening this box, and the instance variable boxes is a reference to the array of boxes that are readily seen up on opening this box (boxes is null if this box does not contain any boxes).

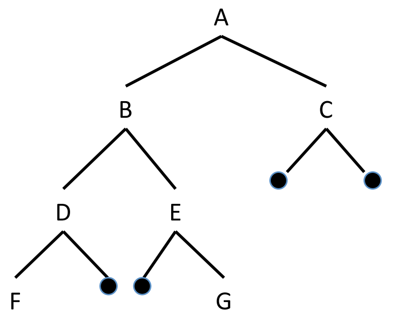
```
public class Box {
   public int cash;   // dollars readily seen up on opening this box
   public Box [] boxes;   // sequence of boxes readily seen up on opening this box
}
```

Now consider the following method:

```
// returns the total amount in box b and in all of the boxes nested within b
public int wealthInside(Box b)
```

Implement the above method using recursion. You are not allowed to write or use any other methods (so, no helper methods are allowed). It is important that your method be correct, efficient, comprehensible, and elegant. (Use the other side of this sheet if you need more space.)

9. Write a binary tree method that prints to `System.out` one line for each node in the tree, with each line of the form
`parent [, child1 [, child2]]`
where the brackets indicate things that may or may not be there (it may have 0, 1, or 2 children). For example:



```
A,B,C
B,D,E
D,F
F
E,G
G
C
```

10. You have been recruited by a search engine company whose local copy of the web is taking up too much disk space. Inspired by Huffman, you decide to compress the web, though based on words rather than characters. That is, the most common word will have the shortest compressed representation, the second most common the second shortest, and so forth. To save even more space, you will use a "lossy" compression: words that are too rare will be replaced with the wildcard word "foo". (So "foo" might then become a very common word.)

(a) Write a method to tabulate and return a representation of the frequencies of words in a given `List<String>`.

(b) Write a method that takes such a representation and returns a lossy version, containing only words that appear at least a given `minAppearances` number of times, and with the frequency of "foo" appropriately augmented (it might or might not already appear).

(c) Apply the Huffman coding approach to build a code tree based on lossy frequencies (`minAppearances` = 2) for the words in the following test documents:

this is a small test
this test is small but not as small as that
that test is almost as small as this one

When there are ties in frequencies, break them in alphabetical order (the one closer to "a" is smaller). No code is needed, just the tree.

11. Update the following heap to insert the value 5 and restore the min-heap property.

```
[3,6,4,7,9,12,8,10]
```

12. Assume that we are implementing a stack using an array `a`. The n items in the stack are in positions 0 through $n-1$. What are the run times for push and pop when the top of the stack is at position 0? At position $n-1$? Why? Assume that we are implementing a stack using a singly linked list? What are the run times for push and pop when the top of the stack is the first item in the list? The last item in the list? Why?

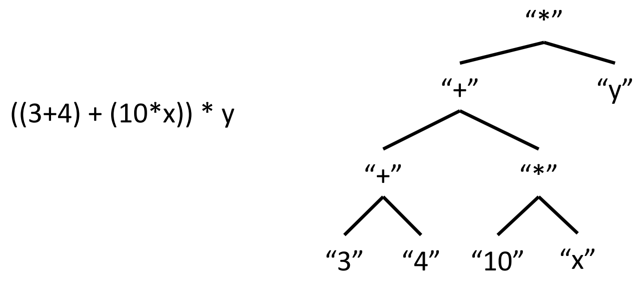
13. Given a list of teams, conduct a tournament in which for each round, the best-seeded remaining team (smallest number) plays the worst-seeded remaining team (largest), the second-best-seeded the second-worst, etc. E.g., with 8 teams, the first round would be 1 vs. 8, 2 vs. 7, etc. Supposing that the best seeds always wins, then the second round would be 1 vs. 4, 2 vs. 3; the final would be 1 vs. 2; the champion would be 1. Here, a helper method `winner()` determines the winner of each game (i.e., it's not always the best seed).

```
class Team implements Comparable<Team> {
  String name;
  int seed;

  // Returns either team1 or team2, depending on outcome of game
  public static Team winner(Team team1, Team team2) { /* body unimportant */ }

  // "<" according to seed
  public int compareTo(Team o) { return seed - o.seed; }
```

Complete the `champion()` method, conducting as many rounds as necessary in order to return the champion from a list of entrants, determining the winner of each game by `Team.winner`. You may assume that the number of entrants is a power of 2. Use a "min-max priority queue", which is like the PQ you have been using, except that both min and max objects can be determined/removed. The declaration is at the end of the exam; assume the implementation is available for "new"ing.

```
  public static Team champion(List<Team> entrants) {
```

14. A simple arithmetic expression can be represented as a binary tree, in which the leaves are values (here, either numbers or variable names) and the inner nodes are operations (here either "+" or "*"). For example:

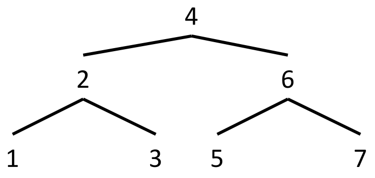


Write a method to evaluate such an expression, yielding the value. Note that everything in the tree is stored as a String. The values of the variables are passed in a `Map<String,Double>`; e.g., {"x" $\mapsto$ 2.0, "y" $\mapsto$ 0.5}. The values of the numbers may be obtained from their String representations by the method `Double.parseDouble`; e.g., `Double.parseDouble("3")` gives the number 3.0 from the string "3". Assume that all variables are in the map and that everything else can be parsed as a valid double. Only handle the arithmetic operations corresponding to strings "+" and "*".

```
public static double eval(BinaryTree<String> t, Map<String,Double> vars) {
```

15. Huffman trees are built "bottom up", repeatedly merging two trees into one. Another way to build trees is "top down", repeatedly finding a representative to serve as the data for the root and splitting the remaining elements to be built into its left and right subtrees. For example, if we used the median as the representative, we would build a nicely balanced binary search tree:

[5,1,3,7,4,2,6]: rep 4, left built from [1,3,2] and right from [5,7,6]
[1,3,2]: rep 2, left built from [1] and right from [3]
[5,7,6]: rep 6, left built from [5] and right from [7]
[1], [3], [5], [7]: leaf nodes



Write a method to build a tree top-down. Assume that a method to identify a representative is already available (i.e., don't fill in its body), and that the elements provide a "compareTo" method to enable checking whether they are "<" or ">" the representative (assume no ties). The list is not modified by the representative finder method (i.e., it only returns the rep, doesn't remove it), and should not be modified by your method.

```
public static <T> T rep(List<T> elems) {
  // Returns one of the elems, without modifying the list.
  // Assume given. In the example, it returns the median element, but it could be anything.
}

public static <T extends Comparable<T>> BinaryTree<T> buildTree(List<T> elems) {
```

16. The registrar needs help from CS 10 students. Banner has the courses for each department, and the students enrolled in each course, but now they need to know for each department which students are taking any of that department's courses. For example:

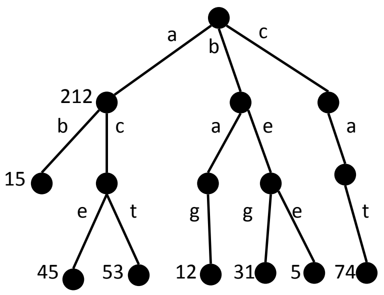
dept-to-courses: { “COSC” ↦ { “COSC 31”, “COSC 50”, . . . }, “CHEM” ↦ { “CHEM 5”, “CHEM 6”, . . . } }
course-to-students: { “COSC 31” ↦ { “Alice”, “Bob”, . . . }, “COSC 50” ↦ { “Alice”, “Charlie”, . . . }, “CHEM 5” ↦ { }, “CHEM 6” ↦ { “Alice”, “Dory”, “Elvis” } }
⇒
dept-to-students: { “COSC” ↦ { “Alice”, “Bob”, “Charlie”, . . . }, “CHEM” ↦ { “Alice”, “Dory”, “Elvis” } }

Declare and write a method that takes a department-to-courses map and a course-to-students map, and returns a department-to-students map.

17. Show the heap resulting after two calls to `extractMin` on the following heap. You may express the heap as either an array or a binary tree.
[3, 8, 5, 9, 10, 15, 12, 13]

18. A *multiset* is like a set, except it allows duplicates and keeps track of how many duplicates there are of each element. For example, a class with a couple of “Alice”s, a single “Bob”, and a bunch of “Elvis”es would have the multiset: { “Alice” (2x), “Bob” (1x), “Elvis” (15x) }. Operations on the multiset include adding an element, removing an element (just one, if there are duplicates), and checking how many copies of an element there are (if any). Name two different possible implementations of this ADT and give a sentence for each, briefly summarizing how to add/remove/check. Give one more sentence characterizing efficiency trade-offs between the two implementations.

19. A tree data structure can be useful to support “autocomplete” — type the first few letters of a word and quickly find out what words start with those letters. Just as with a Huffman tree, each letter in the input corresponds to taking a branch, though now there are branches for ‘a’ to ‘z’ instead of just ‘0’ and ’1’. In the example, “ace” goes left for ‘a’, right for ‘c’, and left for ‘e’. To make the most valuable autocomplete suggestions, we keep track of word frequencies. In the example, “ace” has a frequency of 45, “act” 72, and the word “a” 212. Only non-zero frequencies are shown, corresponding to words represented in the structure by following paths from the root.



We can generalize our binary tree data structure to represent such a tree:

```
public class Autocomplete {
  int freq = 0;
  Map<Character,Autocomplete> children;

  Autocomplete() {
    children = new HashMap<Character,Autocomplete>();
  }
}
```

So instead of two instance variables for the children (`left` and `right`), each node keeps a map from a character to the associated child (a good old-fashioned binary tree would just have two different characters as keys). In the example, the root’s children map has three entries, for ‘a’, ‘b’, and ‘c’. For reference, recall that `s.charAt(i)` returns the character at position `i` in string `s`.

The following methods are all member functions of the class.

(a) Write a method to extract from a tree a map of words and their frequencies. (This would typically be done in a subtree after having typed a few characters.) For the root in the example:
{ “a” ↦ 212, “ab” ↦ 15, “ace” ↦ 45, “act” ↦ 53, “bag” ↦ 12, “beg” ↦ 31, “bee” ↦ 4, “cat” ↦ 74 }

```
Map<String, Integer> allWords() {
```

(b) Write a method to insert into a tree a word and its associated frequency. In the example, inserting the word "ad" would require making a 'd' child for the node with frequency 212, which already has children for 'b' and 'c'. Inserting the word "be" doesn't require anything other than entering the frequency in the associated node. And inserting the word "dad" would require inserting three nodes, basically parallel to "cat".

```
void insert(String s, int freq) {
```

(c) Write a method to compute the average length of words in a tree *without enumerating the words themselves* (which would be inefficient). In the example figure (without insertions from (b)), the average is $21.0/8.0 = 2.625$.

```
double averageWordLength() {
```