**PROBLEM 2 :** (*Trees (22 points)*)

For the purposes of this problem, a *full, complete* binary tree with n levels has 2^{n-1} leaf nodes and, more generally, 2^{k-1} nodes at level k where the root is at level 1, the root's two children are at level 2, and so on. The diagram below shows two such trees, the tree on the left is a level-3 full, complete tree and the tree on the right is a level-2 full, complete tree.
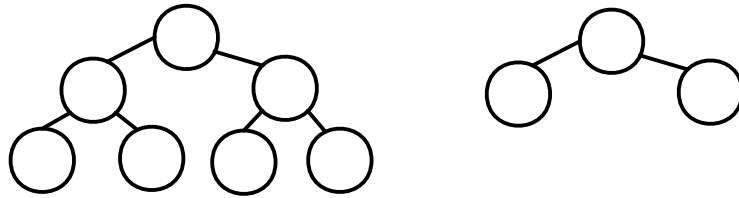
In this problem tree nodes have parent pointers. The declaration for such tree nodes follows.

```
public static class TreeNode {
    String info;
    TreeNode left, right, parent;
    TreeNode(String s, TreeNode lptr, TreeNode rptr, TreeNode pptr) {
        info = s;
        left = lptr; right = rptr;
        parent = pptr;
    }
}
```

Part A (6 points)

Write the method *makeComplete* that returns a full-complete binary tree with the specified number of levels. The call `makeComplete(3,null)` should return a tree such as the one above on the left; `makeComplete(1,null)` should return a single-node tree. The root of the tree has a null parent; all other tree nodes should have correct parent pointers. Use the empty string "" for the `info` value when creating nodes.

```
/**
 * Return root of a full complete binary tree with # levels specified,
 * returning null when level == 0.
 * @param level is the level of the full/complete tree
 * @param parent is the parent of the root being created and returned
 */
TreeNode makeComplete(int level, TreeNode parent) {

    if (level == 0) return null;
    return new TreeNode("",makeComplete(level-1),
                           makeComplete(level-1),parent);

}
```

8

Part B (4 points) What is the recurrence relation, and big-Oh solution for the code you wrote for part A for an n-level tree? Justify your answer

Part C (6 points)

For this problem you'll treat the full complete tree like a *tournament tree*. In a tournament tree, leaf-value store names. Each internal node stores the winner of the values stored in its two children (since the tree is complete, all non-leaf/internal nodes have two children).

For example, the tree below shows a hypothetical tournament tree with the leaf value storing the names of schools competing in a computer programming contest tournament.
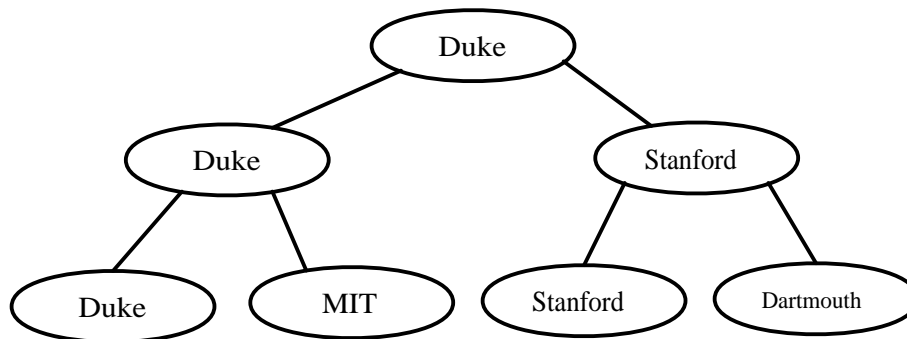
Assume you have a full, complete binary tree, e.g., as would be returned by the function `makeComplete` from Part A. Write the function `assign2leaves` that assigns values in a stack passed to the function to the leaves. For example, suppose the stack is created by the code below:

```
Stack<String> names = new Stack<String>();
names.push("Dartmouth");
names.push("Stanford");
names.push("MIT");
names.push("Duke");
```

then the call `assign2leaves(root, names)` where root is the root of a level-3 full, complete tree should assign values as shown above to the leaves. Note that `"Duke"` is the value at the top of the stack and is stored as the left-most leaf. Your code should do this – this means also that the right-most leaf gets the first value pushed onto the stack.

(continued)

Complete the function below.

```
/**
 * Assign values from stack to leaves of a tournament tree, the
 * stack is emptied in the process.
 * @param stack has at least as many values as there are leaves
 * @param root is root of full, complete tree
 */
public void assign2leaves(TreeNode root, Stack<String> names) {




















}
```

Part D (6 points)

In this problem you'll assign winners to the internal nodes of a tree. Assume all leaf nodes have been assigned values, e.g., as in the tournament tree diagrammed previously. You can also assume that a method `determineWinner` exists that will determine which of two teams has won (or will win) a match. For example, here's code to determine the winner of a match between `"Duke"` and `"MIT"`.

```
String winner = determineWinner("Duke","MIT");
```

Here's the javadoc/method header.

```
/**
 * Determine (or predict) winner of a contest/match between two teams
 * and return the winner. Return null if no winner can be determined.
 * @param teamA is one of the teams in the match
 * @param teamB is the other team in the match
 * @return one of teamA or teamB depending on who wins, returns null
 * if winner cannot be determined.
 */
public String determineWinner(String teamA String teamB)
```

Write the method `assignwinners` whose header is given below. The function is passed the root of a tournament tree like the one diagrammed above. Assume all the leaf values have been filled in. The method should assign values to internal nodes so that each internal node stores the winner of the match played between the internal node's children. The winner is determined by calling method `determineWinner`, assume the method always returns a non-null String.

```
public void assignwinners(TreeNode root)
{




















}
```

PROBLEM 3 : (*E-voting (18 points)*)

You've been contracted to write a small piece of software to determine the winner of an election using online voting software. Specifically, each person voting has three votes – they can vote for three different people or the same person three times. Candidates are ranked and stored in an array: the first candidate (index zero) receives 3 points, the second candidate (index one) receives 2 points, and the third candidate receives 1 point.

Votes are stored in a map as shown below, this code shows how the map could be constructed, it's not the code that actually records the votes – but it illustrates in code how the map is structured.

```java
Map<String,String[]> votes = new HashMap<String,String[]>();

votes.put("Jim", new String[]{"Bob", "John", "Mary"});
votes.put("Sam", new String[]{"John", "Mary", "Mary"});
votes.put("Chris", new String[]{"Mary","Bob","Alice"});
```

These votes show that Jim gave three points to Bob, two to John, and one point to Mary. The final election results in Bob with 5 points, John with 5 points, Mary with 7 points, and Alice with 1 point; Mary would be the winner.

Part A (10 points)

Write the method `voteCalc` that returns a map providing vote totals for every candidate voted for as described above. The method returns a map in which keys are candidate names and in which corresponding values are the vote total for the candidate. This means that for the map returned for the data shown above the value of `map.get("Mary")` should be 7 and the value of `map.get("Alice")` should be 1.

```java
/**
 * Return a map of candidates->vote totals based on votes recorded.
 */
public Map<String,Integer> voteCalc(Map<String,String[]> votes){




}
```

Part B (8 points)

One of your colleagues has written code to determine the names of the candidates who would be in a runoff. Any candidate receiving at least 25% of the votes cast participates in a runoff. The code below was written by your colleague before being hired away by an Australian voting firm. The code below adds candidate names to a priority queue so that the first element removed from the queue is the candidate with the most votes, i.e., the priority is determined by votes with high vote-getters having a higher priority.

You should add code to return an ArrayList of those candidates who participate in a runoff. The candidate with the most votes should be stored at index zero of the ArrayList and in general the ArrayList should be ordered from highest to lowest vote-getter of the runoff candidates (ties don't matter). The idea is to repeatedly remove candidate names from the priority queue and add them to an ArrayList if the vote total for the candidate has at least the 25% threshold of total votes. As soon as a candidate doesn't have 25% of the total votes you can stop since the priority queue will remove candidate names from highest vote-getter to lowest. Assume method `voteCalc` works as specified. For example, for the sample data on the previous page the list returned should contain "Mary" first and then "Bob" and "John" in some order since the total number of votes is 18, 25% of that is 4.5, and Bob and John are tied with five votes while Mary has 7.

```
public ArrayList<String> runOff(Map<String,String[]> votes) {

    final Map<String,Integer> tally  = voteCalc(votes);

    PriorityQueue<String> pq = new PriorityQueue<String>(10,
            new Comparator<String>(){
                public int compare(String o1, String o2) {
                    return tally.get(o2) - tally.get(o1);
                }
    });
    pq.addAll(tally.keySet());          // all candidates in pq

    double total = 0.0;
    for(int count : tally.values()){   // find total # votes cast
        total += count;
    }
    // add code below
```

```
}
```