CS140 Final Exam - Spring, 2012 - James S. Plank

In all of the example pieces of code, you may assume that all of the proper "#include" and "using" statements are at the top of the program. I am skipping them to save on space. If I ask you to give output, show me everything, including spaces and newlines. If a program is in the file **xxx.cpp**, then you can assume that it has been compiled into the output program **xxx**. When you write a program, you may skip the "#include" and "using" statements.

Put your answers on the given answer sheets. You can work on other sheets of paper, but I do not want to see your work.

Question 1

```
class Sudoku {
  public:
    Sudoku();
    void Print_Screen();
    void Print_Convert();
    int Solve();
  protected:
    int Is_Row_Valid(int r);
    int Is_Col_Valid(int c);
    int Is_Panel_Valid(int r, int c);
    int Recursive_Solve(int r, int c);
    vector <string> Grid;
    vector <int> checker;
};
```

Recall the header file for the Sudoku solver that we wrote in class, which is listed here to the right.

Internally, the puzzle is held as a vector of strings in the variable **Grid**, where the characters '1' through '9' represent filled cells, and spaces represent empty cells. The methods **Is_Row_Valid()**, **Is_Col_Valid()** and **Is_Panel_Valid()** return booleans specifying whether the given row, column and panel are valid (**r** and **c** in **Is_Panel_Valid()** can specify any row or column in the panel -- the method will figure out which panel it is -- this differs slightly from the lecture notes, but it will make your life easier.).

The method **Recursive_Solve()** assumes that all cells in rows less than **r**, and all cells in row **r** and columns less than **c** are filled in legally, and it either returns zero, saying that the puzzle is unsolvable, or one with the puzzle solved.

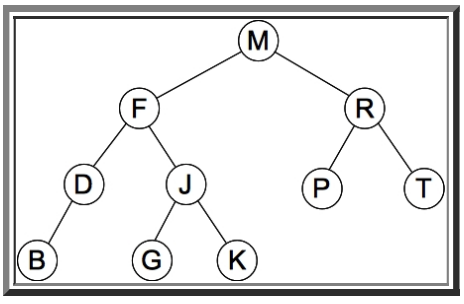
You may assume that all methods are written, with the exception of **Recursive_Solve()**.

Write **Recursive_Solve()**.

Question 2: True/False

- **Statement A**: If a node in a binary search tree has two non-NULL children, you cannot delete it from the tree.
- **Statement B**: Destructors are only important if a program calls **delete**.
- **Statement C**: In C++, the statement **x = y** can only copy megabytes worth of data from **y** to **x** if **x** and **y** are not pointers.
- **Statement D**: If $f(n) = O(g(n))$, then $f(n) = \Omega(g(n))$.
- **Statement E**: You may model recursion with a stack.
- **Statement F**: If $f(n) = O(g(n))$, then $2f(n) = O(g(n))$.
- **Statement G**: With B-Trees, searching for keys within a node is much less expensive than accessing a new node.
- **Statement H**: If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.
- **Statement I**: If an AVL tree has a height of h, then the tree rooted at the left child of the tree must have a height of $h-1$.
- **Statement J**: You can use a preorder traversal on binary search tree to print its elements in sorted order.
- **Statement K**: If a B-tree has a height of h, then the tree rooted at the leftmost child the tree must have a height of $h-1$.
- **Statement L**: If $f(n) = O(g(n))$, then $f(n) + \log_2(f(n)) = O(g(n))$.
- **Statement M**: B-Trees are binary.
- **Statement N**: In terms of big-O, deletion from an AVL tree is more expensive than finding an element.
- **Statement O**: Finding an element in a binary search tree with n elements can have running time complexity as high as $O(n)$.

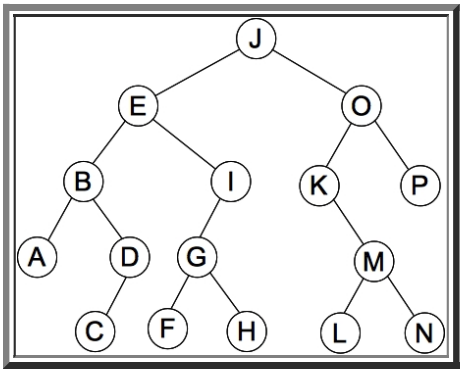
Question 3



Draw the tree that results when you perform the following insertions into the AVL tree to the right. Do not show cumulative insertions. Just show what happens when you insert the given node into the tree on the right. I know I'm repeating myself here, but in **Part B**, you are inserting **A** into the tree on the right, not a tree that has a **Q** in it.

- **Part A**: Insert **Q** into the tree to the right.
- **Part B**: Insert **A** into the tree to the right.
- **Part C**: Insert **H** into the tree to the right.

Question 4

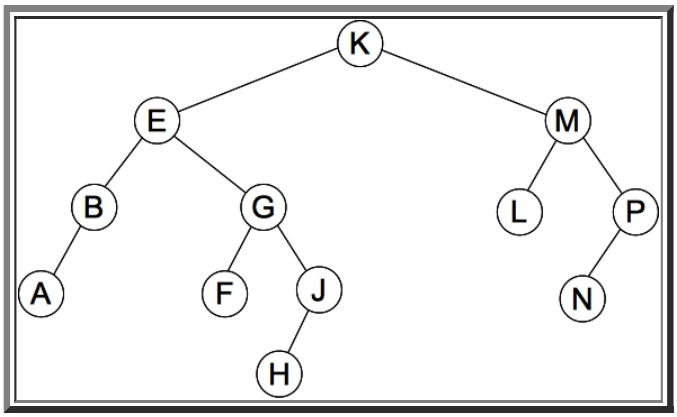


The tree to the right is a regular tree, not an AVL tree.

Part A: Why is it not an AVL tree (be specific)?

Part B Draw the tree that results when you delete the root. Again, this is a regular binary search tree, not an AVL tree.

Question 5



The tree to the right is an AVL tree. For each deletion below, tell me what rotations are required to rebalance the tree. Specify rotations as "Rotation about node *X*," etc. As with Question 3, deletions are not cumulative -- each deletion is to the tree to the right, not to the previously deleted tree:

- **Part A**: Delete **F** from the tree to the right.
- **Part B**: Delete **B** from the tree to the right.
- **Part C**: Delete **L** from the tree to the right.

Question 6

For this question, we define a team to consist of seven *players*. Each player has a name, which consists only of lowercase letters. Your job is to write the program **nu_teams.cpp**. This program will read teams from standard input, one after another. It must then print the number of unique teams. Two teams are the same if they have the same seven players, specified in any order. Thus:

```
UNIX> cat list1.txt
doc happy sneezy grumpy dopey bashful sleepy
greg peter jan cindy marsha bobby alice
monica rachel phoebe chandler joey ross gunther
UNIX> nu_teams < list1.txt
3
UNIX> cat list2.txt
dopey bashful doc happy sneezy sleepy grumpy
grumpy sneezy dopey bashful sleepy doc happy
dopey grumpy doc sneezy sleepy snookie bashful
sleepy grumpy doc violent happy dopey sneezy
violent sneezy dopey doc sleepy grumpy happy
violent angry petulant moronic peevish evil snookie
UNIX> nu_teams < list2.txt
4    (Lines 1 & 2 are the same team, as are lines 4 & 5)
UNIX> cat list3.txt
x x x x x o o
x o x o x o x
x o o x x x x
o o x x x o x
o x x x x o x
UNIX> nu_teams < list3.txt
2    (Lines 1, 3 & 5 are the same team, as are lines 2 & 4)
UNIX>
```

CONSTRAINTS

- Each line will contain exactly seven names.
- Each name will contain up to 50 characters, all of which are lowercase letters.
- There will be less than 1,000,000 lines on standard input.

Hint: I did this with a vector, a set, two strings and an integer. You only need one string, but using two made it clearer.

As a reminder, if you want to sort vector **v**, you do: **sort(v.begin(), v.end())**.

Also, remember, you don't need to specify **#include** or **using** lines.