CPS 100 **Exam 2** **Fall 1996**

Dr. Rodger

Declarations - Structs

```
struct Node                   // "binary tree declaration"
{
    string info;
    Node * left;
    Node * right;
    Node (string in, Node * lf = 0, Node * rt = 0)      // constructor
    {  info = in;   left = lf;   right = rt; }
};

struct GenNode                   // "general tree declaration"
{
    string name;
    GenNode * sibling;
    GenNode * child;
};

struct Node23         // "2-3 tree declaration"
{
    bool leaf;        // true if a leaf, false if a nonleaf
    Node23 * left;    // pointer to left subtree
    Node23 * middle;  // pointer to middle subtree
    Node23 * right;   // pointer to right subtree
    int val1;         // if a leaf, then the leaf value
                      // if a nonleaf, largest value in left subtree
    int val2;         // if a leaf, no value
                      // if a nonleaf, largest value in middle subtree
};
```

QueueAr.h

```
template <class Etype>
class Queue
{
  public:
    Queue( );                                 // construct empty queue
    ~Queue( ) {}                              // destruct (nothing now)
    const Queue & operator=( const Queue & Rhs );  // assign
    void Enqueue( const Etype & X );          // insert X (at rear)
    void Dequeue( );                          // remove first element
    void MakeEmpty( );                        // clear queue to 0 elements
    const Etype & GetFront( ) const;          // return front (still there)
    bool IsEmpty( ) const;                    // true if empty else false
    bool IsFull( ) const;                     // true if full else false
    void Print() const;                       // queue printed to cout
    int NumEnqueues() const;                  // return # calls of Enqueue

  private:
    // not shown
};
```

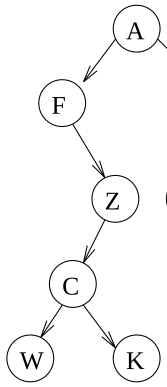
PROBLEM 1 : (*Binary Trees:* 24 pts)

Part A (*5 points*)

Draw the binary *search* tree that results from inserting the letters B, W, M, G, Y, E, P (in that order) into an initially empty tree.

Part B (*6 points*)

Give the preorder and inorder traversals of the tree shown below.



preorder:

inorder:

Part C (*4 points*)

The postorder traversal of the above tree is: WKCZFYHMEA, thus the **reverse** postorder traversal is: AEMHYFZCKW.

(continue on the next page)

Part C *continued*

Write the **recursive** function **ReversePostorder** to list the elements in reverse postorder. Calculating the postorder traversal first, and then listing the elements in reverse will receive **no credit!**

Complete *ReversePostorder* below the following header (you do not need this much space).

```
void ReversePostorder(Node * T)
// postcondition: prints values in T in reverse postorder
{
```

Part D (*8 points*)

Write the function *LevelOrder* that lists the values in a tree in level order. Level order traversal lists the nodes in a tree level by level starting with the root, on each level nodes are listed left to right.

For example, *LevelOrder(T)* on the tree T in Part B outputs the values: AFEZYMCHWK.

Hint: You will need to use a Queue. See page 2 for QueueAr.h.

```
void LevelOrder(Node * T)
// postcondition: prints values in T in level order
{
```

Part E (*1 point*)

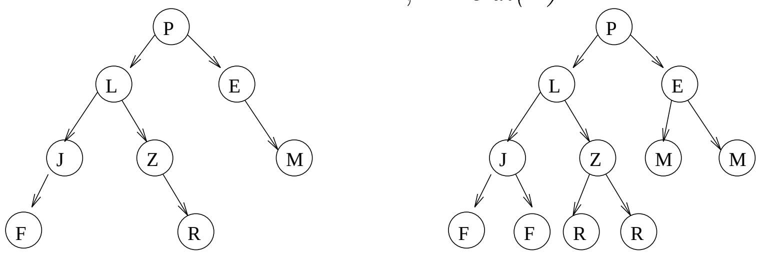
Assume Tree T contains N nodes. What is the worst case (big-Oh) running time of the function *LevelOrder* you wrote in Part D?

PROBLEM 2 : (*Expanding and Shrinking:* 14 pts)

Part A (*7 pts*)

Write the **recursive** function *FillOut* whose header is given below. *FillOut* is given a binary tree, and modifies the tree so that every node that has only one child is added a second child with the same value.

For example, in the tree T shown below on the left, *FillOut(T)* results in the tree on the right.

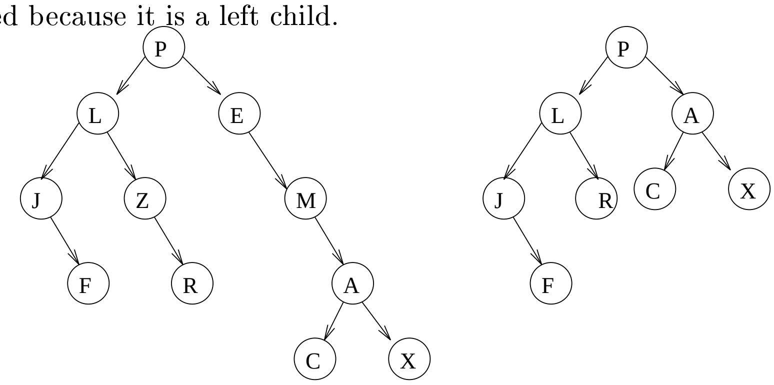



```
void FillOut(Node * T)
// postcondition: Every node that previously had only one child now has
//                a second child with the same value as the other child
{
```

Part B (*7 pts*)

Write the **recursive** function *Shrink* whose header is given below. *Shrink* is given a binary tree, and modifies the tree so that every right child that has a right child but no left child has been removed.

For example, in the tree T shown below on the left, *Shrink(T)* results in the tree on the right. Note that E, M, and Z (all right children) were removed since they had a right child and no left child. J is not removed because it is a left child.



Complete *Shrink* below the following header.

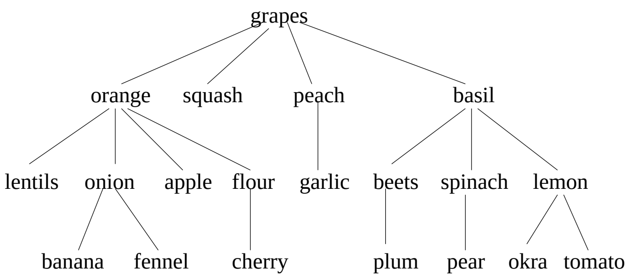
```
void Shrink(Node * T)
// postcondition: Every right child that had a right child but no left
//                child has been removed.
```

PROBLEM 3 : (*Are you my Mother?:* (11 pts))

Part A (*6 pts*)

Write the function *Find* whose header is given below. *Find* is given a general tree and a name, and returns a pointer to the node in the tree whose value is name, or returns NULL if name is not in the tree.

For example, in the general tree T shown below, *Find(T,"spinach")* returns a pointer to the GenNode containing "spinach", and *Find(T,"avocado")* returns NULL, since "avocado" does not appear in T.



Remember that general trees are implemented as binary trees with a pointer to a sibling and child. See the GenNode definition on page 2.

```
GenNode * Find (GenNode * T, string name)
// precondition: values in T are unique
// postcondition: returns a pointer to the GenNode with value 'name',
//                returns NULL if 'name' is not in T
{
```

Part B (*5 pts*)

Write the function *IsDescendant* whose header is given below. *IsDescendant* is given a general tree and two names, and returns true if the second name is a descendant of the first name; otherwise, it returns false.

For example, in the general tree T shown in Part A, *IsDescendant(T,"orange", "fennel")* returns true, and *IsDescendant(T,"peach","okra")* returns false.

You may call the function *Find* that you wrote in Part A. Assume *Find* works correctly, regardless of what you wrote in Part A.

Complete *IsDescendant* below the following header.

```
bool IsDescendant(GenNode * T, string name1, string name2)
// precondition: T contains unique values, name1 is not equal to name2
```

```
// postcondition: returns true if name2 is a descendant of name1,
//                otherwise returns false.
{
```

PROBLEM 4 : (*2-3 Trees:* (6 pts))

Assume a 2-3 tree T has N nodes and you are given a pointer to the root of the tree. What is the worst case running time (big-Oh) of each of the following operations on 2-3 tree T?

1. Calculate the number of leaves in T:
2. Find the minimum value in T:
3. Is value X in the 2-3 tree?:

(Space below here not needed)

PROBLEM 5 : (*Sorting sorted is ginorst:* (15 pts))

Part A (*3 pts*)
Give an example of a six element array of integers that would require Quicksort to take $O(n^2)$ worst case time to sort.

Part B (*3 pts*)
Shellsort calls Insertion sort, which is worst case $O(N^2)$, as part of its algorithm. Selection sort is also worst case $O(N^2)$. Could Shellsort call Selection sort instead of Insertion sort and achieve the same worst case analysis? Explain.

Part C (*5 pts*)
Two arrays are *equivalent* if each element in the first array is also in the second array, and vice versa; the arrays contain the same elements. It does not matter if arrays contain duplicate copies of elements.

In the example shown below, arrays A and B are equivalent (each element in A appears in B, and each element in B appears in A). Arrays A and C are not equivalent, and arrays B and C are not equivalent.

	0	1	2	3	4
A:	12	6	8	12	6
B:	8	12	8	8	6
C:	8	6	10	8	12

Briefly describe an $O(N \log N)$ algorithm to determine if two *unordered* arrays of size N are equivalent. Give the running time of each step in your algorithm.

Part D (*4 pts*)
Briefly describe an $O(N + M)$ algorithm to determine if two *unordered* arrays of size N, each containing elements in the range 1 to M, are equivalent. Give the running time of each step in your algorithm. (YOU DO NOT NEED THIS MUCH SPACE)

PROBLEM 6 : (*Recursion:* (6 pts))

Consider the recursive function *MazeSolve* to solve a maze. An initial position is passed in (xpos,ypos) and moves can be made one at a time either left, right, up or down on empty (or blank) spaces. The object is to find the gold (indicated by 'G'). The 'X' represents a wall. The struct Position is defined on the next page.

```
Position MazeSolve(Matrix <char> & maze, int rows, int cols, int xpos, int ypos)
// precondition: maze has 'rows' rows and 'cols' columns, 'X' appears
//               all around the border, one 'G' appears in the maze,
//               (xpos,ypos) is the current position.
// postcondition: returns true if 'G' is found and the (x,y) position of G
//                otherwise returns false and (x,y) is invalid
{
     Position Pos;
     Pos.found = false;   // not found yet
     if (maze[xpos][ypos] == 'G')  // found gold
     {
        Pos.found = true;
        Pos.x = xpos;
        Pos.y = ypos;
        return Pos;
     }
     else if (maze[xpos][ypos] == ' ')
     {
         maze[xpos][ypos] = 'X';  // mark been here

         Pos = MazeSolve(maze,rows,cols,xpos+1,ypos);  // check down
         if (Pos.found == true)
         {
            return Pos;
         }
         Pos = MazeSolve(maze,rows,cols,xpos-1,ypos);  // check up
         if (Pos.found == true)
         {
            return Pos;
         }
         Pos = MazeSolve(maze,rows,cols,xpos,ypos+1);  // check right
         if (Pos.found == true)
         {
            return Pos;
         }
         Pos = MazeSolve(maze,rows,cols,xpos,ypos-1);  // check left
         if (Pos.found == true)
         {
            return Pos;
         }
     }
     return Pos;   // not found
}

struct Position
```

```
{
   bool found;   // true if found
   int x;        // xposition of gold if found
   int y;        // yposition of gold if found
};
```

Consider the following maze represented by Matrix M.

	0	1	2	3	4	5
0	X	X	X	X	X	X
1	X	X				X
2	X	X	X	X		X
3	X	*		X		X
4	X			G		X
5	X	X	X	X	X	X

Show the tree of recursive calls for the call MazeSolve(M,row,col,3,1). The position (3,1) is indicated above by the $*$ just to show you where this position is, but note that this position really contains a blank (the $*$ is NOT in the actual Matrix). You can represent the call MazeSolve(M,row,col,3,1) by the shorthand notation MS(3,1) since we are only interested in the xpos and ypos positions.

Complete the tree of recursive calls below. The first call is shown for you.

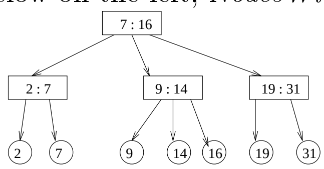
```
MS(3,1)
```

EXTRA CREDIT *(6 pts)*

This problem is optional.

Write the **recursive** function *NodesWith3* whose header is given below. *NodesWith3* is given a 2-3 tree, and returns the number of nodes in the tree that have exactly three children.

For example, in the tree T shown below on the left, *NodesWith3(T)* returns 2.



See the definition of Node23 on page 2.

Complete *NodesWith3* below the following header.

```
int NodesWith3(Node23 * T)
// postcondition: returns the number of nodes that have three children
{
```