PROBLEM 1:  *(Sorting Match-up: (12 pts))*

Below is the state of a set of numbers after each step of a sort. Initially the numbers are in random order (Step 0), and they progress to being sorted (Steps 1-7). For this problem, you are to match the name of a sort to its steps below. Note there is one extra sort listed that will not be matched to a set of steps.

Given these sorts that we studied in class:

Bubble  Insertion  Selection  Quick  Merge

Match them with these sorting steps:

- Which sort is this?

| Step 0:  | 441 481 950 210 944 241 843 5 |
| Step 1:  | 441 481 950 210 944 241 843 5 |
| Step 2:  | 441 481 210 950 944 241 843 5 |
| Step 3:  | 210 441 481 950 944 241 843 5 |
| Step 4:  | 210 441 481 950 241 944 843 5 |
| Step 5:  | 210 441 950 944 210 843 5 |
| Step 6:  | 210 441 481 950 5 241 843 944 |
| Step 7:  | 5 210 241 441 481 843 944 950 |

- Which sort is this?

| Step 0:  | 441 481 950 210 944 241 843 5 |
| Step 1:  | 441 481 950 210 944 241 843 5 |
| Step 2:  | 441 481 950 210 944 241 843 5 |
| Step 3:  | 210 441 481 950 944 241 843 5 |
| Step 4:  | 210 441 481 944 950 241 843 5 |
| Step 5:  | 210 241 441 481 944 950 843 5 |
| Step 6:  | 210 241 441 481 843 944 950 5 |
| Step 7:  | 5 210 241 441 481 843 944 950 |

- Which sort is this?

| Step 0:  | 441 481 950 210 944 241 843 5 |
| Step 1:  | 441 481 210 944 241 843 5 |
| Step 2:  | 441 210 481 241 843 5 |
| Step 3:  | 210 441 241 481 5 843 944 950 |
| Step 4:  | 210 241 441 5 481 843 944 950 |
| Step 5:  | 210 241 5 441 481 843 944 950 |
| Step 6:  | 210 5 241 441 481 843 944 950 |
| Step 7:  | 5 210 241 441 481 843 944 950 |
PROBLEM 2:  (Insert into a Tree: (10 pts))

Part A: Insert the values 41, 33, 37 and 39 (in this order) into the following binary search tree. Show the resulting tree after each insertion.

Part B: Insert the values 41, 33, 37, and 39 (in this order) into the following red-black tree. Use B to indicate black nodes and R to indicate red nodes. Show the resulting tree after each insertion. (The Exam 2 Handout has pseudocode for inserting into a red-black tree).

PROBLEM 3:  (Family Tree: (8 pts))

See the definition for a GenTree on the Exam 2 Handout.

Part A:

Write the function NumChildren whose header is shown below. NumChildren returns the number of children for a given node.

For example, in the figure shown below, if T is pointing to the node containing Doug, then NumChildren(T) would return 4. If T is pointing to the node containing Bill, then NumChildren(T) would return 2.

```
Step 0:  441  481  950  210  944  241  843  5
Step 1:  5   481  950  210  944  241  843  441
Step 2:  5   210  950  481  944  241  843  441
Step 3:  5   210  241  481  944  950  843  441
Step 4:  5   210  241  441  944  950  843  441
Step 5:  5   210  241  441  481  950  843  944
Step 6:  5   210  241  441  481  843  950  944
Step 7:  5   210  241  441  481  843  944  950
```
int NumChildren(GenTree * T)
  // precondition: T is not NULL
  // postcondition: returns the number of children of T
{

Part B:

Write the function BigFamily whose header appears below. BigFamily is given a pointer to a tree and a maximum value and returns a pointer to any node that has more than maximum children. Returns NULL if there are no such nodes.

In the figure in part A, if T points to the root of the tree, then BigFamily(T, 3) could return a pointer to either Paul or Doug, since both of them have more than 3 children.

GenTree * BigFamily(GenTree * T, int maximum)
  // postcondition: returns a node with at least maximum children, or
  // returns NULL if no nodes have maximum children
{

PROBLEM 4: (Max Tree: (12 pts))

A max-tree is a tree in which every node is larger than or equal to the nodes in its subtree. Below is a picture of a max-tree. The largest value in the tree, 89, is at the root of the tree. All the nodes in 89’s left and right subtrees are less than 89. Note: Children are not ordered as they are in a binary search tree.
See the definition for a MaxTree on the Exam 2 Handout.

**PART A:** Write the function `PrintAllLargest` that prints all values in the tree that are greater than a given value X.

For example, in the figure above, if T is pointing to the root of the tree, `PrintAllLargest(T, 19)` would print the numbers 89, 66, 20, 55, 54, 51, and 45 (in any order).

```cpp
void PrintAllLargest(MaxTree * T, int X)
// postcondition: prints all values in the tree larger than X
{
```

**PART B: ANALYSIS**

Assume a max-tree contains N nodes and efficient algorithms are used for the following problems.

1. For part A, assume there are M nodes greater than X, where M is substantially smaller than N. What is the worst case running time of the `PrintAllLargest` function you wrote for part A?

2. What is the worst case running time to find the maximum value in a max-tree?

3. What is the worst case running time to find the minimum value in a max-tree?

4. What is the worst case running time to print the values in a max-tree in sorted order?

**PROBLEM 5:** *(Priority Stack: 12 pts)*

Consider modifying the stack class to become a priority stack. A priority stack is the same as a stack except that each item pushed onto the stack also has a priority associated with it, 1 being highest priority and 100 being lowest priority.

Assume the stack class has been implemented using a linked list. `Pop()` and `IsEmpty()` work as before. However, `Push()` will be modified to keep items stacked based on their priority and a new member function `PopGroup()` will be added.
The stack is maintained by keeping items with highest priority on top, items with the same priority are stacked in LIFO order. When an item is pushed onto the stack it is inserted into the stack underneath all items with higher priority, but as the top item with the same priority.

For example, the left figure below shows a priority stack, the middle picture shows the stack after pushing the person Ralph with priority 7 on, and the right figure shows the stack after pushing the person Jasmine with priority 3 on.

Use the following definitions.

```cpp
struct Person
{
    string name;
    int priority;
    Person * next;
};

class Stack
{
public:
    Stack(); // construct empty stack
    ~Stack(); // destructor
    bool IsEmpty() const; // return true if empty, else false
    void Push( Person * item ); // push item onto top of stack
    void Pop( Person * & item ); // pop’s item off of stack
    void PopGroup(); // pop items with top priority

private:
    Person * myTop; // top element of stack
};
```
The constructor and one of the member functions have been implemented.

```cpp
Stack::Stack() : myTop(NULL) {
}

bool Stack::IsEmpty() const {
    return (myTop == NULL);
}
```

**Part A** Complete the function Push, which pushes one item on the stack, keeping the stack ordered by priority.

```cpp
void Stack::Push(Person * item) {
}
```

**Part B** Complete the function Pop, which pops and returns the top item off the stack.

```cpp
void Stack::Pop(Person * & item)  
// precondition: Stack is not empty
{
}
```

**Part C** Complete the function PopGroup, which pops all the elements with the same priority as the top element off of the stack. (The memory for these elements should be returned to the memory heap).

For example, using the rightmost stack in the figure above, a call to PopGroup() would pop all persons with priority 2, in this case Janet. A second call to PopGroup() would pop all persons with priority 3, in this case Jasmine, Sandra and Sarah.

```cpp
void Stack::PopGroup()  
// precondition: Stack is not empty  
// postcondition: All elements with the same priority as the top item have  
//           been removed.
{
}
```

**PART D: Design**

Explain why the next field in the Person struct of the given stack class is a poor design.
PART A: Printing in Reverse Order

Write a function `PrintReversed` that prints the contents of a binary search tree to an output stream in reverse sorted order.

For example, given the tree below, your function should print out on the output stream: z y w v u t q p g f d b

```
   p
   / \
  g   u
 /     /
d     q   z
 / \
 b   t
 / \
f   w
 /   \\
 v   y
```

```c
void PrintReversed(TreeNode * t, ostream & output)
// precondition: t is the root of a binary search tree
// postcondition: contents of t have been printed to output in
// reverse sorted order, one on each line
{
```

PART B: Tree2List

Write function `Tree2List` that converts a binary search tree into a sorted linked list. You will complete this problem in two parts: a recursive function that traverses the tree and uses a queue to keep track of the names visited; and a function that calls the recursive function and then uses the resulting queue to build a list in sorted order.

Write a recursive function `Tree2ListRec` that traverses a binary search tree in order and enqueues each name. The Exam 2 Handout has the queue class.

```c
void Tree2ListRec(TreeNode * t, Queue<string> & q)
// precondition: t is the root of a binary search tree
// postcondition: q contains the names in t in sorted order
{
```

Write a function `Tree2List` that uses `Tree2ListRec` and the resulting queue to build a sorted linked list. You may assume that `Tree2ListRec` from the previous section works correctly even if your implementation does not.

```c
Node * Tree2List(TreeNode * t)
```
// precondition: t is the root of a binary search tree
// postcondition: returns a pointer to the first node of a
//                sorted linked list containing same values as t
{

PROBLEM 7:  (EXTRA CREDIT: Stacking Them Up: (2 pts))

Describe how the function Tree2ListRec would change if you were to use a stack instead of a queue to hold each name as you traversed the tree.

PROBLEM 8:  (EXTRA EXTRA CREDIT: Building the List Directly: (3 pts))

Describe how you could write the function Tree2ListRec so that it built the list directly, without using either an extra stack or queue.

CPS 100E                     Exam 2 Handout                      Fall 1997
Rodger and Duvall

struct GenTree
{
    string info;
    GenTree * sibling; // a sibling of T
    GenTree * child;   // a child of T
};

struct MaxTree
{
    int value;
    MaxTree * left;
    MaxTree * right;
};

struct List
{
    string name;
    List * next;

    List(const string & newName, List * newNext = 0)
        : name(newName),
          next(newNext)
    {}
};
struct Tree
{
    string name;
    Tree * left;
    Tree * right;

    Tree(const string & newName, 
           Tree * newLeft = 0, Tree * newRight = 0)
    : name(newName),
      left(newLeft),
      right(newRight)
    {}
    
};

Red-Black tree Insert Operation

RB-Insert(root,x)
    Tree-Insert(root,x)    // binary search tree insert
    color x red
    while x!=root & & x->p -> color==red do
        if x->p is a left child then
            let y denote the sibling of x->p
            if y is red then
                . color x->p and y black
                . color x->p->p red
                . x = x->p->p
            else
                . if x is a right child then
                    . . x = x->p
                    . . Left-rotate(root,x)
                else
                    . . color x->p black and x->p->p red
                    . . Right-rotate(root,x->p->p)
else (case if \( x \rightarrow p \) is a right child)
. let \( y \) denote the sibling of \( x \rightarrow p \)
. if \( y \) is red then
. . . color \( x \rightarrow p \) and \( y \) black
. . . color \( x \rightarrow p \rightarrow p \) red
. . . \( x = x \rightarrow p \rightarrow p \)
. else
. . . if \( x \) is a left child then
. . . . \( x = x \rightarrow p \)
. . . . Right-rotate(root, \( x \))
. . . . color \( x \rightarrow p \) black and \( x \rightarrow p \rightarrow p \) red
. . . . Left-rotate(root, \( x \rightarrow p \rightarrow p \))
color root black
template <class Type>

class Queue
{
    public:

    // constructors/destructor

    Queue( );                     // construct empty queue
    Queue( const Queue & q );     // copy constructor
    Queue( );                     // destructor

    // assignment

    const Queue & operator = ( const Queue & rhs );

    // accessors

    const Type & front( ) const;   // return front (no dequeue)
    bool isEmpty( ) const;        // return true if empty else false
    int length( ) const;          // return number of elements in queue

    // modifiers

    void enqueue( const Type & item ); // insert item (at rear)
    void dequeue( );               // remove first element
    void dequeue( Type & item );    // combine front and dequeue
    void makeEmpty( );             // make queue empty

    private:

    int mySize;                   // # of elts currently in queue
    int myFront;                  // index of first element
    int myBack;                   // index of last element
    Vector<Type> myElements;      // internal storage for elements

    // private helper functions
    void DoubleQueue();           // double storage for myElements
    void Increment(int & val) const; // add one with wraparound
};