Test 1: CPS 100E

Owen Astrachan and Dee Ramm

November 19, 1996

Name: ____

Honor code acknowledgement (signature) ____

	value	grade
Problem 1	15 pts.	
Problem 2	15 pts.	
Problem 3	8 pts.	
Problem 4	12 pts.	
Problem 5	15 pts.	
Extra	5 pts.	
TOTAL:	65 pts.	

This test has 11 pages, be sure your test has them all (page 11 is blank, use it if you need more room). Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

The following declarations are used throughout the test

```
struct Node
{
    string info;
    Node * next;
    Node(const string & s, Node * link = 0)
        : info(s),
          next(link)
    {}
};

struct Tree
{
    string info;
    Tree * left;
    Tree * right;

    Tree(const string & s,
         Tree * lchild = 0, Tree * rchild = 0)
        : info(s),
          left(lchild),
          right(rchild)
    {}
};
```

PROBLEM 1 : (*Short Cuts and Answers* 15 points)

Part A (2 points)

All the items from a stack are popped off and inserted, in the order popped, onto a queue. Then all items are dequeued and inserted, in the order dequeued, back onto the stack. How does the order of the items on the stack compare to the original order (before all items are popped)?

Part B (2 points)

Similar to part A, but items are initially in a queue, are dequeued and inserted onto a stack, then are popped from the stack and inserted back onto the queue. How does the order of the items on the queue compare to the original order?

Part C (4 points)

Is the statement below true or false, and why? (why is worth 3 points):

Merge sort for vectors is $O(n \log n)$, but is $O(n^2)$ for linked lists because the middle of a vector can be determined in $O(1)$ time, but the middle of a linked list requires $O(n)$ time to determine.

Part D (4 points)

In a doubly linked list each node points to both its successor and its predecessor (the nodes after and before, respectively). Is the following statement true or flase, and why? (why is worth 3 points).

Given pointers to the first and last nodes of a doubly linked list, the elements can be printed in reverse order (from last to first) in $O(n)$ time, but to print a *singly linked* list in reverse order (given pointers to the first and last nodes) requires $O(n^2)$ time (space is not an issue).

Part E (3 points)

Insertion sort is an $O(n^2)$ sort. Merge sort is an $O(n \log n)$ sort. A coder implements both of these sorts efficiently, but notices that insertion sort is faster than merge sort for vectors/arrays of fewer than 50 elements. Provide a reason for this observation.

PROBLEM 2 : (*Land o' Linkin* 15 points)

Part A: (6 points)

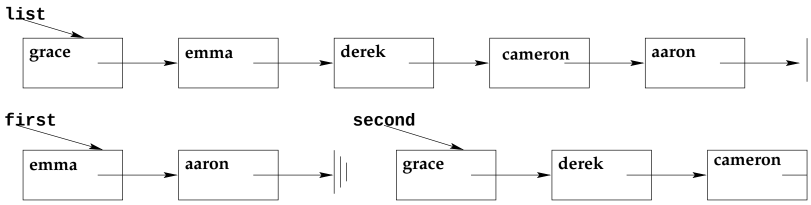
The function `VectorToList` returns a pointer to a linked list containing the same elements, in the same order, that are stored in the `Vector` parameter `v`. It works by calling a **recursive auxiliary function** `AuxVTL`. Write the *recursive* function `AuxVTL`.

```
Node * VectorToList(const Vector<string> & v, int numElts)
// precondition: numElts = # elements stored in v
// postcondition: returns pointer to the first node of a linked
//                list whose nodes contain values
//                in the order v[0], v[1], ... v[numElts-1]
{
    return AuxVTL(v,0,numElts);
}

Node * AuxVTL(const Vector<string> & v, int index, int numElts)
// precondition: 0 <= index < numElts, numElts = # elements stored in v
// postcondition: returns pointer to first node of a linked list
//                whose nodes contain values in the order
//                v[index], v[index+1], ..., v[numElts-1].
//                Return 0 if precondition not satisfied
//
// THIS FUNCTION MUST BE RECURSIVE!!
{









}
```

Part B: (9 points)

Write the function `ListSplit` whose header is given below. The function moves the nodes from `list` to two other lists — no new nodes are created, nodes are moved. The list pointed to by `first` should contain all nodes whose names begin with vowels; the list pointed to by `second` should contain all nodes whose names do NOT begin with vowels. Assume a function `IsVowel` exists (the prototype is shown below). You **do NOT** need to write `IsVowel`. Note that if `s = "elbow"`, then `IsVowel(s[0])` evaluates to true (since `s[0] == 'e'`) and `IsVowel(s[2])` evaluates to false (since `s[2] == 'b'`).

For example, the picture below shows one scenario given an initial list of five names.



The order of the nodes in `first` and `second` is not important.

```
bool IsVowel(char ch);

void ListSplit(Node * list, Node * & first, Node * & second)
// precondition: list points to a 0-terminated linked list
// postcondition:  nodes whose info field begin with a vowel are
//                 in the list pointed to by first, other nodes are
//                 in the list pointed to by second.
```

PROBLEM 3 : (*Adiorrsx* 8 points)

Part A:

A "fast" algorithm can be used to sort vectors of numbers if the range of the numbers is known to be between 0 and m. The idea is to count how many zeros, ones, twos, ..., m's there are in the original vector. These counts are then used to fill the original vector in with values. For example, if a vector contains (0,2,1,2,1,1,0,4,4,2), then there are 2 zeros, 3 ones, 3 twos, and 2 fours. These counts can be used to "construct" a sorted vector: (0,0,1,1,1,2,2,2,4,4) by filling in 2 zeros, followed by 3 ones, followed by 3 twos, followed by 2 fours — the counts are used to stores values in the vector.

The incomplete function below implements this algorithm, fill in the rest of the function.

```
void Sort(Vector<int> & v, int numElts)
// precondition: numElts = # elements in v
// postcondition: v is sorted
{
    int max = FindMax(v,numElts);  // makes max largest value in v
    Vector<int> aux(max+1,0);
    int j,k;

    for(k=0; k < numElts; k++)
    {
        aux[v[k]]++;
    }
    // use values in aux to store values back in v









}
```

Part B:

What is the complexity of the algorithm (use big-Oh notation) for an n element vector. Assume that m is much smaller than n so that your answer is only in terms of n. Briefly justify your answer.

(continued)

Part C:

What happens with this algorithm (in terms of performance) if m is much larger than n?

PROBLEM 4 : (*eeorrst (*12 points*)*)

In this problem the functions `VectorToTree` and `VectorToTreeII` convert a sorted vector into a binary search tree (different methods are used in the different functions). Assume that the function `Insert` correctly inserts a value into a binary search tree (we went over this function in class).

```
Tree * VectorToTree(const Vector<string> & a, int first, int last)
// precondition: a[first] <= a[first+1] <= ... <= a[last]
// postcondition: returns balanced search tree containing
//                a[first]...a[last]
{
    if (first <= last)
    {
        int mid = (first + last)/2;

        return new Tree(a[mid], VectorToTree(a,first,mid-1),
                        VectorToTree(a,mid+1,last));
    }
    return 0;
}

Tree * VectorToTreeII(const Vector<string> & a, int first, int last)
// precondition: a[first] <= a[first+1] <= ... <= a[last]
// postcondition: returns search tree containing a[first]...a[last]
{
    Tree * t = 0;
    int k;
    for(k=first; k <= last; k++)
    {
        Insert(t,a[k]);
    }
    return t;
}
```

Part A: (3 points)

Note that the vector `a` is sorted.

What is the complexity (using big-Oh) of the function `VectorToTree` for an n element vector? (briefly justify your answer).

Part B (3 points)

What is the complexity of the function `VectorToTreeII` assuming that `Insert` does NOT do any rebalancing? (briefly justify your answer).

Part C (3 points)

What is the complexity of the function `VectorToTreeII` in the case that `Insert` reblances the tree (as though it was an AVL tree)? (briefly justify your answer)

Part D (3 points)

Does the answer to each of the three questions above change if the parameter `a` is a linked-list instead of a vector, but the basic algorithm remains the same for the functions (which are renamed `ListToTree` and `ListToTreeII`? Why?

PROBLEM 5 : (*Trees* 15 points)

Part A: (2 points)

The code below prints a binary search tree in alphabetical order

```
void Print(Tree * t)
{
    if (t != 0)
    {
        Print(t->left);
        cout << t->info << endl;
        Print(t->right);
    }
}
```

Indicate how to modify the function to print the values in a binary search tree in reverse alphabetical order.

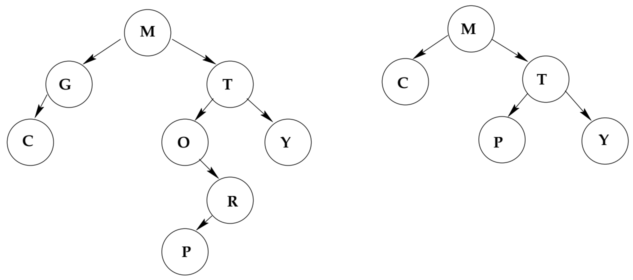
Part B: (5 points)

Assume a tree stores ints (rather than strings as shown in the declaration at the beginning of the test). Write a function that returns the sum of all the values stored in the nodes. The function should return zero for an empty tree.

```
int SumNodes(Tree * t)
// postcondition: returns sum of all values in t
{


}
```

Part C: (8 points)

Write a function `Compress` that removes all nodes with exactly one child from a tree, replacing each node with one child by the child (and doing this recursively). The tree diagrammed below on the left is transformed into the tree on the right by `Compress`.




```
void Compress(Tree * & t)
// postcondition: all nodes of t with a single child have been removed
```

PROBLEM 6 : (*Extra Credit* 5 points)
Part of the declaration of the class `Queue` is provided below. A member function `Queue::Size()` has been added that returns the number of elements in a queue.

```
template <class Etype>
class Queue
{
  public:
    Queue( );                               // construct empty queue
    ~Queue( ) {}                            // destruct (nothing now)

    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
    int  Size()    const;                  // returns # elements in queue
};
```

Write a function `Interchange` that interchanges the order of each pair of adjacent elements in a queue. Assume that the queue has an even number of elements. For example: a queue of ints (1,2,3,4,5,6) where 1 is at the front of the queue will change to (2,1,4,3,6,5) where 2 is at the front of the queue. The queue (2,1,3) should be changed to (1,2,3). Your code must use $O(1)$ storage and run in $O(n)$ time for an n-element queue (this means you won't be able to define another queue, or a vector, or a stack).

```
  template <class Type>
  void Interchange(Queue<Type> & q)
  // postcondition: adjacent pairs of elements in q have been interchanged.
```