Test 1: CPS 100

Owen Astrachan and Dee Ramm

February 21, 1997

Name: ________________________________

Honor code acknowledgment (signature) ________________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>9 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>21 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>74 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 9 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 50 minutes.

In writing code you do not need to worry about specifying the proper \#include header files. Assume that all the header files we’ve discussed are included in any code you write.

Class declarations for stacks and queues are provided on the last page of the test. You can remove this page to make it easier to have a reference to these classes when needed.
PROBLEM 1:  (Vocabulary: 10 points)

For each of the words/phrases below, circle the definition that is the best description as it pertains in the context of computer science, programming, and C/C++.

1. Stack
   (a) A first-in, first-out data structure
   (b) A last-in first-out data structure
   (c) A low-level C++ construct used in implementing vectors

2. big-Oh
   (a) A heuristic for developing large-scale, object-oriented programs efficiently and on time
   (b) A notation that expresses an asymptotic upper bound on resource requirements such as time or space
   (c) A nomenclature used solely to differentiate sorting algorithms

3. base case
   (a) A construct used for transporting a large musical instrument that is a member of the string family
   (b) The part of a recursive function that is computable without making any recursive calls (the stopping condition)
   (c) The zero-th element of a vector, used to anchor the vector in memory

4. segmentation violation
   (a) An attempt by a non-member function to access private data which is not permitted in C++
   (b) An error that can result from dereferencing a NULL or garbage pointer value
   (c) A design problem that result from including data in the public section rather than restricting data to be private

5. heap
   (a) Fragmented memory resulting from too many calls of new.
   (b) Where memory is allocated from statically, at compile time.
   (c) Where memory is allocated from dynamically, at run time

PROBLEM 2:  (In a fix 9 points)

Evaluate each of the following postfix expressions

\[ 3 5 7 + * 4 6 * - \]

\[ 12 8 + 5 * 9 14 17 + + * \]
Find the exact value of the postfix expression diagrammed below

\[ 1 2 3 \cdots n \quad + + \cdots + \]
\((n-1)\) times

**PROBLEM 3:** (Frequent Occurrences 21 points)

Linked lists of strings, with a header node, are implemented using the declaration for `Node` below.

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

**Part A:** 6 points

Write the function `Count` whose header is given below. `Count` returns the number of occurrences of `key` in `list`. For example, `Count(list,"apple")` should evaluate to 2 and `Count(list,"cherry")` should evaluate to 0 where `list` is shown below.

```
int Count(Node * list, const string & key)
// precondition: list has a header node
// postcondition: returns # occurrences of key in list
{
```
Part B: 12 points

Write the function Modal whose header is shown below. Modal returns the string that occurs most often (or at least as many times as any other string) in list. If there is a tie, any string that is a modal string can be returned. Return "" for an empty list. In writing Modal, you can call the function Count from Part A; assume that the function Count works as intended.

```cpp
string Modal(Node * list)  
// precondition: list has header node 
// postcondition: returns the modal string in list  
{
```

Part C: 3 points

For a list of $N$ nodes, what is the complexity of the solution you wrote in part B using big-Oh notation. Justify your answer briefly.

```cpp
}
```
PROBLEM 4: (Spoiled Fruit 10 points)

Part A: 6 points
Complete the function Compress below that removes duplicate nodes from a sorted linked list. In the diagram below, a list is shown before and after duplicates are removed.

```
while (list && list->next) // at least two nodes?
{
    if (list->info == list->next->info) // remove list->next
    {
        list = list->next; // not a dupe, goto next node
    }
    else
    {
    }
}
```

Part B: (4 points)
What is the complexity, using big-Oh, of the function Compress; justify your answer briefly. How would the complexity change if the list to be compressed (so that it contains no duplicates) is NOT sorted?
PROBLEM 5: (SQL: 12 points)

For this problem you can use stacks, queues (and, of course, vectors and linked lists, etc.). Declarations for the stack and queue class are at the end of the test.

Postscript is a stack based language, one of the operations in postscript is ReverseN which reverses the top \( N \) elements of a stack. For example, if the top of a stack is to the left, then \( \text{ReverseN}(s,4) \) for the stack \( s = (1, 2, 3, 4, 5, 6, 7, 8, 9) \) changes the stack to \( s = (4, 3, 2, 1, 5, 6, 7, 8, 9) \).

Write the function \( \text{ReverseN} \) whose header is shown below. Assume the precondition holds, don’t worry about stacks that contain fewer than \( N \) elements.

```c
void Reverse(Stack<int> & s, int n)
// precondition: n <= s.Size(), s contains at least n elements
// postcondition: top n elements of s are reversed
{
```


PROBLEM 6: (Building Trees 12 points)

Write a function MakeTree that converts a sorted, doubly-linked list (NO header node) into a binary search tree. The middle node of the linked list should be the root of the search tree. Assume the existence of a function MiddleNode, that returns a pointer to the middle node of a doubly-linked list, you do NOT need to write MiddleNode.

```
DNode * MiddleNode(DNode * list)
// precondition: list is NULL-terminated, NO header node
// postcondition: returns pointer to middle node of list,
// returns 0 if list is empty
```

Declarations for doubly-linked lists and binary trees are given below, DNode for doubly-linked lists, TNode for binary trees.

```
// for doubly-linked lists
struct DNode
{
    string info;
    DNode * prev;
    DNode * next;
    DNode(const string & s,
          DNode * prior = 0,
          DNode * after = 0)
        : info(s),
        prev(prior),
        next(after) { }
};

// for binary (search trees)
struct TNode
{
    string info;
    TNode * left;
    TNode * right;
    TNode(const string & s,
          TNode * lchild = 0,
          TNode * rchild = 0)
        : info(s),
        left(lchild),
        right(rchild) { }
};
```

You must use the following algorithm to build the search tree:

1. Find the middle node, build a root node (TNode) with an info field the same as the middle node.
2. Unlink the middle node from the list (so the next field of the node before it points to NULL, and the prev field of the node after it points to NULL) — this step is necessary for the next step (so that MiddleNode will work) — you may want to store pointers to the nodes before and after the middle node.
3. Make left and right subtrees recursively, using the list before the middle node for the left subtree and the list after the middle node for the right subtree.
4. Link the middle node back into the list (which is why you may have stored pointers in step 2).

(continued)
TNode * MakeTree(DNode * list)
// precondition: list is NULL-terminated, NO header node
// postcondition: returns a pointer to the root of a binary search
// tree containing same values in list
{

}
template <class Type>
class Queue
{
  public:
    Queue(); // construct empty queue
    void Enqueue(const Type & elt); // insert elt (at rear)
    void Dequeue(); // remove first element
    void Dequeue(Type & elt); // remove first element
    void MakeEmpty(); // clear queue to 0 elements
    const Type & GetFront() const; // return front (still there)
    bool IsEmpty() const; // true if empty else false
    int Size() const; // # of elements in queue
  
  private:
  // not shown
};

template <class Type>
class Stack
{
  public:
    Stack(); // construct empty stack
    void Push(const Type & elt); // push elt onto top of stack
    void Pop(); // pop top element
    void Pop(Type & elt); // pop top element and return
    void MakeEmpty(); // empty stack (no elements)
    const Type & Top() const; // return top element (NO pop)
    bool IsEmpty() const; // return true if empty, else false
    int Size() const; // returns # of elements in stack
  
  private:
  // not shown
};