Test 1: CPS 100

Owen Astrachan        Susan Rodger

October 5, 1995

Name: _______________________________________

Honor code acknowledgement (signature) ____________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Extra</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>56 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 12 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.
PROBLEM 1:  (Stacks/Queues: 6 points)

• What is the value of the postfix expression: 7 5 3 + * 5 2 - 8 9 * +

• Write a postfix expression equivalent to 13 + 4 * (10 - 3). Note that this expression has the value 41.

• Suppose that a Stack class is implemented and will be used to implement a Queue class. The only data members in the private section of the Queue class are an integer representing the number of elements in the queue and a stack for storing queue elements.

    private:
        Stack<Etype> myStack;
        int mySize;

The member function for Enqueue is shown below (recall that both stack and queue classes are templated).

    Queue<Etype>::Enqueue( const Etype & X )
    // postcondition: X added to rear of queue
    {
        myStack.Push(X);
        mySize++;
    }

Describe briefly how to implement the function Dequeue that removes the first element from the queue. You do NOT need to write code (although you can), but do need to describe how to dequeue an element when elements are stored and enqueued as described above.
PROBLEM 2:  (Yin/Yang 10 points)

A circularly-linked list is maintained by keeping a pointer to the last node; the first node is then the node “after” the last node (since the list is circular). Write two functions: \texttt{Prepend} that adds a new node to the front of a circularly-linked list and \texttt{RemoveLast} that removes the last node from a circularly-linked list.

```c
struct Node
{
    int info;
    Node * next;
    Node(int value, Node * follow = 0)
    {
        info = value;
        next = follow;
    }
};
```

![Diagram of list operations before and after Prepend and RemoveLast](image)

Part A: (3 points)

```c
void Prepend(Node * list, int value)
// precondition: list points to last node of NON-empty circularly-linked list
//         (first node is list->next)
// postcondition: new first node with info == value added to
//            the circularly-linked list whose last node is list
{
```
Part B: (7 points)

```c
void RemoveLast(Node * & list)
// precondition: list points to last node of NON-empty circularly-linked list
// (first node is list->next)
// postcondition: last node is removed (returned to heap) and list
// points to the new last node
{
```
PROBLEM 3: (Sequences : 20 points)

A sequence of integers can be considered abstractly as a set of integers in which order does not matter so that the sequence (1,8,2,5) is the same as the sequence (1,2,8,5) since they both contain the set of integers {1,2,5,8}. In this problem you will be asked to implement several functions that manipulate sets of integers where the sets are implemented using the class DigitSeq. Variables of type DigitSeq (with some new member functions) will represent sets. You will be asked to implement operations described below.

<table>
<thead>
<tr>
<th>description</th>
<th>function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>determine if k is in s</td>
<td>s.Contains(k)</td>
</tr>
<tr>
<td>add k to set s if not already present</td>
<td>s.Insert(k)</td>
</tr>
<tr>
<td>is s a subset of t? (are all elements of s in t)</td>
<td>s &lt; t</td>
</tr>
<tr>
<td>are s and t equal? (do s and t contain the same elements)</td>
<td>s == t</td>
</tr>
</tbody>
</table>

For reference, the class declarations for DigitSeq and DigitSeqIterator are reproduced on a separate sheet.

Part A: (6 points)
Write a member function Contains that returns true if a set/sequence contains a number and false otherwise. This function should work regardless of how the class DigitSeq is implemented, i.e., whether Vectors or linked lists are used.

```cpp
bool DigitSeq::Contains(int num) const
// postcondition: returns true if num is in sequence (*this)
// returns false otherwise
```

Part B: (4 points)
Write the member function \texttt{Insert} that inserts a number into a set if the number is NOT already in the set.
In writing \texttt{Insert} you may call function \texttt{Contains}; assume that \texttt{Contains} works as specified regardless of what you wrote in part A.

```cpp
void DigitSeq::Insert(int num)
// postcondition: if num is NOT contained in sequence/set then
//                    num is inserted into the sequence/set
{
```
Part C: (6 points)
The operator $<$ will be overloaded so that $s < t$ if the set $s$ is a subset of the set $t$, that is, if every element of $s$ is also an element of $t$. Conceptually, $(1,4,3) < (3,4,5,2,1)$ is true, but $(1,4,3) < (3,4,5,2,7)$ is false. Write the body of the operator so that it works as intended. You may call function Contains; assume that Contains works as specified regardless of what you write in part A.

```cpp
bool operator < (const DigitSeq & s, const DigitSeq & t)
// postcondition: returns true if s is a subset of t, false otherwise
{
}
```

Part D (4 points)
Two sets $s$ and $t$ are equal if and only if $s$ is a subset of $t$ and $t$ is a subset of $s$. Write a function that overloads $==$ so that it can be used to determine if two sets are equal. Assume that operator $<$ for determining if one set/sequence is a subset of another is implemented correctly regardless of what you wrote in part C.
PROBLEM 4:  (Exiled on East 20 points)

For this problem there is a linked list of dormitories, each represented by a DormNode; each DormNode contains a linked list of students in its dorm, each student is represented by a StudentNode. DormNode and StudentNode are defined as follows (these are reproduced on the attached sheet).

```c
struct DormNode
{
    string name;       // name of a dorm
    DormNode * next;   // next dormitory in list
    StudentNode * students; // linked list of students
};

struct StudentNode
{
    string name;       // first name of a student
    StudentNode * next; // next student in list
};
```

In the example below, duke points to a linked list of DormNodes for Southgate, Randolph, Aycock, Trent and Taylor; each DormNode points to a linked list of students in the dorm.
Part A: (4 points)

Complete the function NumStudents whose header is given below. NumStudents returns the number of students in a dormitory. In the picture above, NumStudents(duke->next->next) should return 4, the number of students in Aycock dorm.

```c
int NumStudents(DormNode * dorm)
{ // precondition: dorm != 0
  // postcondition: returns the number of students in dorm
  
  return 4; // Example return value
}
```
Part B: (4 points)
Complete the function NumDormStudents whose header is shown below. NumDormStudents returns the number of students in the dormitory specified by dormName if that dormitory is in the list pointed to by parameter list. In the picture on the previous page, NumDormStudents(duke,"Randolph") returns 5, the number of students in Randolph dorm; NumDormStudents(duke,"Roundtable") returns 0.
In writing NumDormStudents you may call function NumStudents from part A. Assume that NumStudents works as specified regardless of what you wrote in part A.

```cpp
int NumDormStudents(DormNode * list, string dormName)
// postcondition: returns the number of students in the dormitory in list with name
//                 dormName (and 0 if there is no such dormitory in list)
{
```
Part C: (8 points)
Complete the function PopularName whose header is given below. PopularName returns the name used most frequently by students in a dormitory. If there is a tie, any of most frequently used names may be returned. For example, in the figure at the beginning of this problem the most frequent name in Southgate is Bill, so PopularName(duke) should return "Bill". Since all names in Trent are used equally often, PopularName(duke->next->next->next) can return any of the names in the dormitory. If there are no students in a dorm then the empty string "" should be returned.
In writing PopularName you may call the function SameName whose specification is given below. You do NOT need to write SameName.

```c
int SameName(StudentNode * list)
// postcondition: returns 0 if list is NULL; otherwise returns the number of nodes
// in list whose name field is the same as the first node of list
{
    // assume implemented correctly
}

string PopularName(DormNode * dorm)
// precondition: dorm != 0
// postcondition: returns the name that appears the most frequently
// in the dorm; if there is a tie, return any of the
// most frequently occuring names; if no students in dorm return ""
{
    // implementation
}
```
**Part D:** (4 points)
Assume that there are N dorms, and each dorm has at most M students. The function `SameName` is $O(M)$. What is the running time of each of the functions you wrote above? Briefly justify your answers.

- `NumStudentNodes`

- `NumDormStudent`

- `PopularName`

**PROBLEM 5 : (EXTRA CREDIT (6 points))**

Complete the function `Shuffle` whose header is given below. `Shuffle` moves nodes in a list so that all odd nodes occur before all even nodes and that otherwise the relative order of nodes is unchanged. Nodes are odd or even depending on their position in the list, e.g., even nodes are the 2nd, 4th, 6th, etc., and odd nodes are the 1st, 3rd, 5th, etc. For example, if the original list is (1, 13, 7, 9, 3, 10), then after shuffling the list it should be (1, 7, 3, 13, 9, 10).

(Node is defined earlier in Problem 2.)

```c
void Shuffle(Node * & list)
// precondition: list does not have a header node,
// number of nodes in list is even.
// list = a1, a2, a3, ... ,aN
// postcondition: list = a1, a3, a5, ... a(N-1), a2, a4, ... aN
//
{
```
struct DormNode
{
    string name;    // name of a dorm
    DormNode * next; // next dormitory in list
    StudentNode * students; // linked list of students
};

struct StudentNode
{
    string name;    // first name of a student
    StudentNode * next; // next student in list
};

//---

#include "vector.h"    // access vector class (private)
class DigitSeqIterator; // so friend declaration is ok

class DigitSeq
{
    public:
        DigitSeq();    // constructor (default)
        ~DigitSeq();   // destructor
        DigitSeq(const DigitSeq & ds);    // copy constructor
        DigitSeq & operator = (const DigitSeq & ds);    // assignment

        void Append(int digit);    // add new digit to end
        void Prepend(int digit);   // add new digit to front
        void Clear();              // clear all digits

    friend class DigitSeqIterator;

    private:
        // stuff here doesn't matter
};

class DigitSeqIterator
{
    public:
        DigitSeqIterator(DigitSeq & ds);
        DigitSeqIterator(const DigitSeq & ds);

        void First();    // must call before accessing digits
        bool IsDone() const; // true implies no more digits to access
        void Next();    // advance to next digit in sequence
        int Current() const; // return current element
        int & Current(); // return modifiable current element

    private:
        DigitSeq & mySequence; // bound to this sequence
        int myCurrent;    // internal index of current item
};