Test 1: CPS 100

Owen Astrachan

October 8, 1997

Name: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	9 pts.	
Problem 2	12 pts.	
Problem 3	14 pts.	
Problem 4	10 pts.	
Problem 5	10 pts.	
Problem 6	10 pts.	
TOTAL:	65 pts.	

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, you might want to tear this page off.

PROBLEM 1 : (*Short- and post-stuff* (9 points))

part A *6 points*

Evaluate the following postfix expressions, you can show work for partial credit.

```
3 5 6 * + 8 +
```

```
7 8 2 - * 1 * 6 /
```

part B *3 points*

In a sentence or two, what's the advantage of using the function `Vector::append` in client programs rather than resizing the vector, e.g., why use

```
Vector<string> v;
string s;
while (cin >> s)
{
    v.append(s);
}
```

instead of

```
Vector<string> v;
int count;
string s;
while (cin >> s)
{
    if (count >= v.length()) v.resize(v.length()*2);
    v[count] = s;
    count++;
}
```

PROBLEM 2 : (*Reversal of Fortune* (12 points))

Write the function *reverse* whose header is given below. The function *reverse* reverses the elements of the queue *q*. For example, if *q* is represented by (a,b,c,d), with a the first element and d the last element of the queue, then after the call `reverse(q)` *q* is represented by (d,c,b,a). In part A you may define variables that are stacks, queues, and ints, but not vectors. In part B you may only define variables that are queues or ints.

part A *6 points* You may define stack, queue, and int variables (no vectors).

```
void reverse(Queue<int> & q)
// pre: q represented by (a1, a2, ..., an)
// post: q represented by (an, ..., a2, a1), i.e., q is reversed
{









}
```

part B *6 points* You may define queue and int variables, but no stack variables (hint: think recursively).

```
void reverse(Queue<int> & q)
// pre: q represented by (a1, a2, ..., an)
// post: q represented by (an, ..., a2, a1), i.e., q is reversed
{









}
```

PROBLEM 3 : (*Big-OO-Oh* (14 points))

part A *6 points*

Consider the function *blurb* below. What is the **exact** value of the function calls `blurb(1)` and `blurb(1024)`?

Using big-Oh notation, determine the running time and the value of `blurb(n)`? The running time is how long the function takes to execute, as a function of n; the value is what *blurb(n)* returns, again as a function of n.

```
int blurb(int n)
{
    int count=0;
    while (n > 0)
    {
        count++;
        n = n / 2;
    }
    return count;
}
```

part B *4 points* Consider the code fragment below

```
int j,k;
int sum = 0;
for(j=0; j < n; j++)
{
    for(k=j; k < n; k++)
    {
        sum++;
    }
}
```

Using big-Oh, what is the running time of the two loops as a function of n? Briefly justify.

If `k++` is replaced with `k *= 2`, what is the running time of the two loops (using big-Oh, as a function of n)? Briefly justify.

extra credit: If the inner loop is changed to

```
for(k=0; k < j; k *= 2)
```

What is the (big-Oh) running time of the loops? (Justify your answer)

part C *4 points*

Consider the function *search* below that performs a binary search *correctly* on a doubly linked list given pointers to the first and last nodes of the list. Write a recurrence relation for the function in terms of $T(n)$ where n is the number of elements in the list between (and including) first and last. (note: only one recursive call is made for each invocation of *search*.)

Using the substitution/plug-in method, solve the recurrence. You may find it helpful to note that

```
    1/2 + 1/4 + 1/8 + ... = 1
```

```
Node * search(Node * first, Node * last, const string & key)
// pre: first->info <= ... <= last->info, i.e., list is sorted
// post: returns pointer to node containing key, returns NULL/0
//       if key not found in list
{
    if (first == 0 || last == 0) return 0;

    Node * mid = first;
    Node * ptr = first;

    while (ptr != last)
    {
        ptr = ptr->next;
        if (ptr != 0)
        {
            mid = mid->next;
            ptr = ptr->next;
        }
    }
    if (mid->info == key)     return mid;
    else if (key < mid->info) return search(first,mid->prev);
    else                      return search(mid->next,last);
}
```

PROBLEM 4 : (*Moving experiences* (10 points))

For these problems assume the following declaration for Node

```
struct Node
{
   string info;
   Node * next;
};
```

Write the function *find* whose header is given below. *find* returns a pointer to the first node in *list* containing *key* and returns NULL/0 if *key* does not appear in list. Note that *list* has a header node.

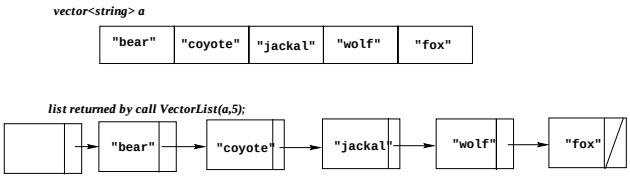
```
Node * find(Node * list, const string & key)
// pre:  list has a header node
// post: return pointer to first node in list containing key
//       return NULL/0 if key is not in list
{









}
```

PROBLEM 5 : (*Conversion* (10 points))

The declaration for node below is used in this problem.

```
template <class Type>
struct Node
{
    Type info;
    Node * next;
    Node(const Type& t, Node * link=0)
        : info(t), next(link)
    { }
};
```

Write the function *VectorToList* that creates a linked list, with a header node, whose nodes contain values from the vector in the same order as the values in the vector. For example




```
Node<string> * VectorToList(const Vector<string> & a, int numElts)
// pre: a contains numElts entries
// post: returns a pointer to a linked list, with a header node,
//       with nodes containing values in the order v[0], v[1], ... v[numElts-1]
//
{









}
```

PROBLEM 6 : (*Garbage Removal* (10 points))

For this problem assume the following declaration for Node:

```
struct Node
{
   string info;
   Node * prev;  // points to previous node
   Node * next;  // point to next node
};
```

Write a function that removes all nodes containing *key* from a doubly-linked, circular list, leaving the list circular. In a one-node circular list the node's `prev` and `next` fields both point to the node itself.

```
void Remove(Node * & list, const string & key)
// pre: list is circular
// post: all nodes with info value == key are removed, list is circular
{









}
```