Test 1: CPS 100

Owen Astrachan Robert Duvall Jeff Forbes

February 19, 2001

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	18 pts.	
Problem 2	10 pts.	
Problem 3	30 pts.	
Problem 4	12 pts.	
TOTAL:	70 pts.	

This test has 10 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.

The declaration for linked list nodes on this test is:

```
struct ListNode
{
  string info;
  ListNode * next;

  ListNode(const string& s, ListNode * ptr)
    : info(s), next(ptr)
  { }
};
```

The declaration for binary search tree nodes on this test is:

```
struct TreeNode
{
  string info;
  TreeNode * left;
  TreeNode * right;

  TreeNode(const string& s, TreeNode * lt, TreeNode * rt)
    : info(s), left(lt), right(rt)
  { }
};
```

PROBLEM 1 : (*Count Chocula (18 points/6 each)*)

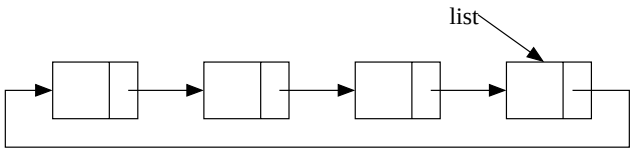
Part A

Write function `count` that returns the number of nodes in a NULL-terminated linked list that does not have a header node.

```
int count(ListNode * list)
// pre:  list is NULL/0 terminated
// post: return # nodes in list
{









}
```

Part B

Write function `circularcount` that returns the number of nodes in a circularly-linked list like the one illustrated below.




```
int circularcount(ListNode * list)
// pre:  list is circular
// post: return # nodes in list, list is unchanged
{









}
```

Part C

Write function `treecount` that returns the number of nodes in a binary tree.

```
int treecount(TreeNode * root)
// pre:  root points to a binary tree (or root == NULL)
// post: return # nodes in tree
{









}
```

PROBLEM 2 : (***Sordid Find (10 points/5 each)***)

Part A

The function `find` below satisfies its postcondition.

```
bool find(ListNode * list, const string& s)
// pre: list has a header node
// post: returns true if s in list, returns false otherwise
{
    list = list->next;    // skip header
    while (list != 0) {
        if (list->info == s) return true;
        list = list->next;
    }
    return false;
}
```

If the list passed to `find` is sorted, the performance of the function can be improved by stopping the search when a value larger than `s` is found (in this case the function can return false without searching the rest of the list). Rewrite the function so that it has this early stopping condition.

```
bool find(ListNode * list, const string& s)
// pre: list has a header node, list is sorted from lowest to highest value
// post: returns true if s in list, returns false otherwise
{









}
```

Part B

The original function `find` has a worst-case complexity $O(n)$ for an n-node list. What is the complexity using big-Oh of the new version you wrote for Part A. Justify your answer briefly.

PROBLEM 3 : (*Intersecting interests*)

An `MSApplicant` sub-class named `MSIntersection` computes the intersection of two `Multiset` objects using the code below.

```
void doIntersection (Multiset * lhs, Multiset * rhs, Multiset * result)
// post: *result is intersection of *lhs and *rhs
{
    MSIntersection intersection(lhs, result);
    rhs->applyAll(intersection);
}
```

The intersection of two multisets contains one occurrence of every string that occurs at least once in both multisets.

For example, the intersection of the multisets

```
(["apple", 2], ["kiwi", 6], ["orange", 4], ["cherry", 23])
```

and

```
(["apple", 10], ["banana", 3], ["orange", 12])
```

is the multiset containing `(["apple", 1], ["orange",1])`, i.e., one occurrence of each value in common to both multisets.

The code for `MSIntersection` is on the last page of the test. You'll be asked to reason about its complexity. The basic algorithm for calculating the intersection as shown in the code above for `doIntersection` when used with `MSIntersection:applyOne` is:

- For each word in multiset rhs, see if it occurs in lhs
- If the word does occur in lhs, add it to the intersection

Part A (4 points)

The call `doIntersect(a,b,result)` will correctly store the intersection of `a` and `b` in `result` regardless of whether `a`, `b`, and `result` are implemented using `LinkedMultisets`, `TableMultisets`, or `TreeMultisets`. In one or two sentences explain why.

Part B (6 points)

What is the big-Oh complexity of calculating the intersection of two `LinkedMultiset` objects `a` and `b` where `a` has `N` elements and `b` has `N` elements, and `result` is an initially empty `LinkedMultiset` object using the following call.

```
doIntersection(a,b,result);
```

Express your answer in terms of `N` and justify your answer briefly. Recall that `count` is an $O(n)$ operation for an n-element `LinkedMultiset` and that `add` is an $O(1)$ or constant-time operation.

Part C (6 points)

What is the complexity using big-Oh of the `doIntersection` call

```
doIntersection(a,b,result);
```

if all multisets are implemented as `TreeMultiset` objects, `a` and `b` both contain `N` elements, and `result` is initially empty. Recall that the `TreeMultiset` class uses a binary search tree so that `count` and `add` are both $O(logn)$ operations for an n-element `TreeMultiset`. Justify your answer briefly.

Part D (8 points)

Suppose multisets `a` and `b` have N and M elements, respectively. Consider the following calculation for storing the intersection of `a` and `b` in `result`.

```
if (a->size() < b->size()) {
    doIntersection(a,b,result)
}
else {
    doIntersection(b,a,result)
}
```

Explain why this method makes an enormous difference when `TreeMultiset` objects are used but very little difference when `LinkedMultiset` objects are used. For full credit you should analyze the complexity of the code when the sizes of `a` and `b` differ by many orders of magnitude, e.g., the sizes are 2^{10} and 2^{20}.

Part E (6 points)

For this part, assume the definition of intersection changes so that the result set contains `min(x,y)` occurrences of strings that occur x times in one set and y times in another and `min` is a function that returns the minimum of its two integer arguments.

For example, the intersection of the multisets

```
(["apple", 2], ["kiwi", 6], ["orange", 4], ["cherry", 23])
```

and

```
(["apple", 10], ["banana", 3], ["orange", 12])
```

is the multiset containing `(["apple", 2], ["orange",4])`,

Rewrite `MSIntersection::applyOne` so that it works with this definition (see the code page for `MSIntersection`).

```
void MSIntersection::applyOne(const string& s, int count)
{









}
```

PROBLEM 4 : (*Least Common Denominator (12 points)*)

This problem involves creating a new sorted linked list from two sorted linked lists. The new linked list contains those elements in common to both sorted linked lists. For example, for the two lists `("bat", "cat", "hat", "sat", "vat")` and `("cat", "fat", "hat", "pat", "sat")` the list returned should be `("cat", "hat", "sat")`. Assume that each list being processed has no duplicate elements, i.e., a list like `("mat", "rat", "rat")` won't be processed since it has a duplicate/repeated element.

Write the function `common` *whose header is shown below so that it runs in* $O(n)$ *time* when both lists `a` and `b` have n nodes.

The idea is to traverse both lists adding nodes to the end of `result` when appropriate. The function is started for you, you should maintain the invariant noted in the comment. *All lists have header nodes.*

```
ListNode * common(ListNode * a, ListNode * b)
// pre: a and b are sorted, both have header nodes,
// post: return sorted list with header node containing
//       values in common to a and b
{
    a = a->next;       // skip header nodes in both a and b
    b = b->next;

    ListNode * result = new ListNode("",0);   // return this list
    ListNode * tail = result;

    // invariant: result list is sorted, tail points to its last node









    return result;
}
```

Multisets have the following interface:

```
class Multiset
{
 public:

  virtual ~Multiset() {}      // multisets are deleted properly

  virtual bool add(const string& word)         = 0; // add word to multiset
  virtual int  count(const string& word) const = 0; // # times word occurs
  virtual int  size()                    const = 0; // # elements in multiset
  virtual void applyAll(MSApplicant& applicant) const = 0;
};
```

MSApplicants have the following interface:

```
class MSApplicant
{
  public:

    virtual ~Multiset() {}
    virtual void applyOne(const string& word, int count) = 0;
};
```

The declaration/definition of `MSIntersection` follow.

```
class MSIntersection : public MSApplicant
{
  public:

    MSIntersection (Multiset * lhs, Multiset * result)
      : myLHS(lhs), myResult(result)
    {}

    virtual void applyOne (const string& word, int count)
    {
        if (myLHS->count(word) != 0) {
            myResult->add(word);
        }
    }

  private:

    Multiset * myLHS;
    Multiset * myResult;
};
```