Test 1: CPS 100

Owen Astrachan

October 11, 2000

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	30 pts.	
Problem 2	16 pts.	
Problem 3	12 pts.	
Problem 4	20 pts.	
TOTAL:	78 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.

Some common recurrences and their solutions.

`T(n) = T(n/2) + O(1)`	$O(\log n)$
`T(n) = T(n/2) + O(n)`	$O(n)$
`T(n) = 2T(n/2) + O(1)`	$O(n)$
`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
`T(n) = T(n-1) + O(1)`	$O(n)$
`T(n) = T(n-1) + O(n)`	$O(n^2)$

The declaration for Nodes on this test is:

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

PROBLEM 1 : (*Listing*)

Part A, *5 points*

Write a function that changes every `'t'` that occurs as the first letter of a word to a `'b'`. No other letters should change. For example,

```
("tin", "tile", "ant", "saint", "tot")
```

should be changed to

```
("bin", "bile", "ant", "saint", "bot")
```

```
void change(Node * list)
// post: all t's that occur as first letters of a word are changed to b's
{









}
```

Part B, *5 points*

The function below correctly counts the number of nodes in a list.

Write a recurrence relation for the function `count`. What is the solution (using big-Oh) to the recurrence?

```
int count(Node * list)
{
    if (list == 0) return 0;
    return 1 + count(list->next);
}
```

Part C, *8 points*

Write the function `vec2list` (see below) that creates a linked list storing the same values that are stored in vector `a` in the same order as they're stored in the vector (the first node of the linked list is `a[0]` in the function below).

```
Node * vec2list(const tvector<string>& a)
// pre: a contains a.size() entries
// post: return pointer to first node of linked list
//       containing same values as in a in same order
```

Part D, *8 points*

The function `find` correctly correctly satisfies its postcondition.

```
Node * find(Node * list, const string& s)
// post: return pointer to first node in list containing s
//       or returns 0/NULL if s not contained in list
{
    while (list != 0) {
        if (list->info == s) return list;
        list = list->next;
    }
    return 0;
}
```

A list `a` is **contained in** a list `b` if every value in `a` is in `b`. For example, `("ant", "cat", "dog", "cat")` is contained in `("dog", "wolf", "pig", "ant", "slug", "cat")` but is *not* contained in `("dog", "ant")`

Write the function `containedIn` whose header is below. You can (and should) call the function `find` in writing `containedIn`.

```
bool containedIn(Node * a, Node * b)
// pre: a and b are 0/NULL-terminated, no header nodes
// post: returns true if a contained in b, false otherwise
```

Part E, *4 points*

Express the complexity of your function `containedIn` using big-Oh notation assuming that both `a` and `b` have N nodes. Justify your answer.

PROBLEM 2 : (*The shape of things*)

The classes `Shape`, `Square`, and `Circle` are shown on the last page of this test (you may tear this page off). The output of the function `main` is shown below:

```
circle
perimeter = 6.28318

circle
perimeter = 18.8495

square
perimeter = 20
```

Part A, *4 points*

If the word `virtual` is removed from before the declaration of the function `name` in the class `Shape` how does the output of the program change?

Part B, *4 points*

What is the purpose of the `= 0` for the functions in `Shape`?

Part C, *4 points*

Suppose a new function `area` is added to the class `Shape`:

```
virtual double area() const = 0;
```

Write the function `Square::area`, the area of a square is $s \times s$ where s is the length of a side of the square.

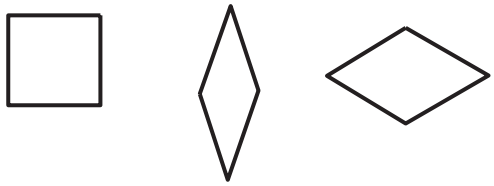
```
double Square::area() const
// post: return area
{



}
```

Part D, *4 points*

A *rhombus* is a four-sided figure in which all sides have the same length and opposite sides are parallel. Three examples of a rhombus are shown below. Each of these has the same perimeter since the length of a side is the same in all three.



A new class `Rhombus` is declared as follows:

```
class Rhombus : public Square
{
 public:
    Rhombus(double s)
      : Square(s)
    { }
    virtual string name() const
    {
        return "rhombus";
    }
};
```

If this class is used in the program whose output is shown above (and whose implementation is given at the end of the test) by adding in main the code below:

```
   shapes.push_back(new Rhombus(7));
```

the output generated for the newly added shape will be as follows (which is correct).

```
rhombus
perimeter = 28
```

However, it is **not** a good idea to have `Rhombus` inherit from `Square` even though it works in the program as shown. Why isn't it a good idea?

PROBLEM 3 : (***Breakfast of Champions***)

Part A, *4 points*

A partial implementation of a class `MSTotal` that computes the total number of entries in a MultiSet is shown below. The total number of entries in `("ant", "ant", "bat", "cat", "ant" )` is 5. You are to write the body of the member function `MSTotal::apply`.

```
class MSTotal : public MSApplicant
{
  public:
    MSTotal();
    virtual void apply(const string& word, int count);
    int getTotal() const;
  private:
    int myTotal;
};

MSTotal::MSTotal()
    : myTotal(0)
{

}

void MSTotal::apply(const string& word, int count)
// post: state updated appropriately
{

}

int MSTotal::getTotal() const
// post: total returned
{
    return myTotal;
}
```

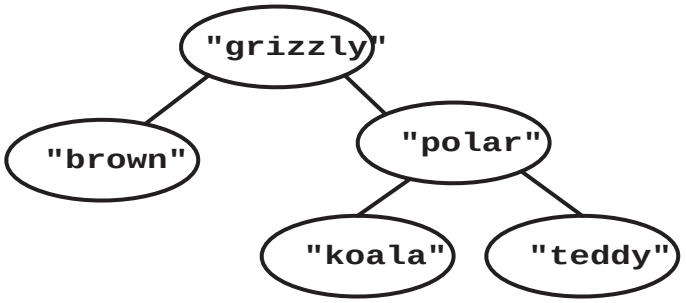
Part B, *8 points*

Write the body of the function `larger` below that returns true if `a` has more total entries than `b` and false otherwise. You may use MSTotal objects and any MSApplicant subclasses or MultiSet functions in writing `larger`.

```
bool larger(const MultiSet& a, const MultiSet& b)
// post: return true if a contains more total entries than b
//       return false otherwise
```

PROBLEM 4 : (*When a Bear attacks . . .*)

The tree below is a search tree.



Part A, *6 points*

1. What is the preorder traversal of the tree?

2. Add nodes containing "black" and "panda" so that the tree remains a search tree. Add "black" first. Draw the nodes attached to the tree diagram above.

Part B, *4 points*

Draw a search tree in which "teddy" is at the root of the tree, and the root's left child is "polar" (include all other nodes from the tree in the diagram above, these other nodes can occur in any order in the tree you draw.)

Part C, *5 points*

The code below for `insert` correctly adds a string to a search tree so that it remains a search tree.

```
void insert(Tree *& root, const string& s)
// pre: root is a search tree
// post: s is added to root, root is still a search tree
{
    if (root == 0)             root = new Node(s,0,0);
    else if (s <= root->info)  insert(root->left,s);
    else                       insert(root->right,s);
}
```

Suppose nodes from a (roughly) balanced search tree are inserted into an initially empty tree using the code below:

```
void copyFromTo(Node * root, Node* & copy)
//pre: root is a search tree,
//post: copy is a search tree containing the same values as root
{
    if (root != 0) {
        insert(copy,root->info);
        copyFromTo(root->left,copy);
        copyFromTo(root->right,copy);
    }
}
```

If there are `N` values in root and copy is initially empty, what is the complexity (using big-Oh) of the call `copyFromTo(root,copy)`? Do **not** write a recurrence relation. Reason about the code. Assume that the tree pointed to by root is balanced.

Part D, *5 points*

If the `copyFromTo` function changes from using a preorder traversal (as it does currently) to an inorder traversal (so that the call to `insert` comes between the two recursive calls) the complexity of the function changes. What is the new complexity using big-Oh? Why?