Test 1: CPS 103

Owen Astrachan

October 8, 1993

Name: ____

Honor code acknowledgement (signature) ______________________________________

	value	grade
Problem 1	9 pts.	
Problem 2	9 pts.	
Problem 3	10 pts.	
Problem 4	15 pts.	
Problem 5	12 pts.	
Problem 6	15 pts.	
Bonus *extra*	6 pts.	
TOTAL:	70 pts.	

PROBLEM 1 : (*Expressions:* 9 points)

What is the value of the C/C++ expression: `6.2 + 5/3`

What is the value of the postfix expression: `8 6 4 + 3 * + 5 +`

Write an equivalent postfix expression for the infix expression: `6 + 5 * 8` assuming standard precedence for order of operations.

PROBLEM 2 : (*Vocabulary:* 9 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. *I/O streams*
   (a) a thought process in which ideas are immediately transformed to programs
   (b) standard classes that are used for input and output of both built-in and user-defined types
   (c) a data aggregate of heterogeneous types.
2. *loop invariant*
   (a) a boolean expression that is true each time a loop guard is evaluated
   (b) the boolean expression whose truth determines if a loop body is executed
   (c) a term used to describe the evaluation of the expression `(a && b)` in which b is <u>not</u> evaluated if a is false.
3. *Abstract Data Type (ADT)*
   (a) the concept that C++ classes are defined in terms of abstract ideas which are subsequently refined into a working implementation
   (b) the idea that different implementations of a stack (e.g., arrays and linked lists) may have different performance characteristics although the implementations support a standard set of operations
   (c) a domain (of things) and the operations defined to manipulate the things in the domain

PROBLEM 3 : (*ListSum:* 10 points)

Linked lists of integers are implemented using the following definitions:

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

Write a *recursive* function *ListSum* that returns the sum of all the info fields in its parameter *list*. By definition if *list* contains no elements then the value zero (0) should be returned. Linked lists are implemented WITHOUT header nodes. Partial credit can be obtained for an iterative solution if you cannot write a recursive solution.

Complete *ListSum* below the following header:

```
int ListSum(Node * list)
// precondition:  list represents  a1, a2, ..., an
// postcondition: returns  a1 + a2 + ... + an
// exception: returns 0 if list is empty
```

PROBLEM 4 : (*Stacks:* 15 points)

Assume that stacks are implemented using *dstack.c*, part of the header file *stack.h* is reproduced below:

```
typedef struct stack_s *StackType;
typedef int ElementType;

StackType InitStack(void);
int Push(ElementType, StackType);     // returns 1 if Push succeeds
int Pop(StackType);                   // returns 1 if Pop succeeds
int IsEmpty(StackType);               // returns 1 if stack empty
ElementType Top(StackType);           // returns top element of stack
```

You are to write the function *DupeSome* whose header is given below. *DupeSome* duplicates the top *num* elements of its stack parameter *s*. For example, if *st* represents the stack (5,6,7,9,2,3), with the top of the stack at the left, then the call `DupeSome(st,3);` would change *st* so that it represents the stack (5,6,7,5,6,7,9,2,3). If *num* is greater than the number of items in the stack *s*, then the entire stack should be duplicated (i.e., values of *num* larger than the number of elements in the stack are treated as equal to the number of elements in the stack).

Complete *DupeSome* below the following header and on the next page. In writing *DupeSome* stacks may be manipulated only using the methods given in the header file *stack.h* and described above – you are NOT to use any knowledge of how the stacks are implemented.

Part A: (*3 pts* It is a fact that in C the value of a parameter cannot change within a function with the change communicated to the calling statement. Nevertheless, both *Pop* and *Push* can alter their stack parameter as can *DupeSome* below. Briefly explain why this is the case.

Part B: (*3 pts*) describe at a very high level the method you propose to implement in **Part C** below. This is to focus your attention on the algorithm as well as provide a plan to use in implementing the function.

Part C: (*9 pts*) Write code to implement *DupeSome*, complete the function below:

```
void DupeSome(StackType s, int num)
```

PROBLEM 5 : (*Multiset: 12 points*)

A *multiset* is a collection similar to a mathematical set, but in which duplicates are allowed, i.e., the same element may appear several times in the multiset. For example, the multiset $\{2,3,2,4,5,5,2\}$ contains 7 elements, but only four different elements (2,3,4,5).

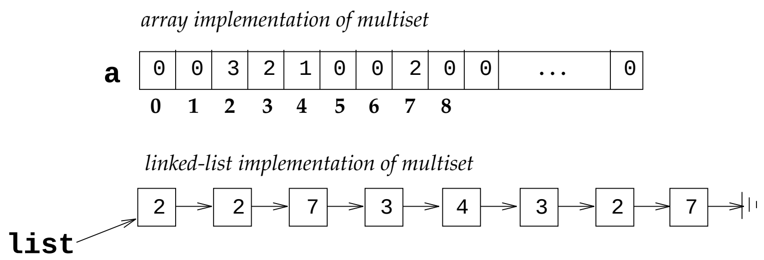
For this question, the only operations on multisets that will be considered are:

- *IsEmpty* : determine if a multiset is empty, i.e., contains any elements.
- *Union* : combine two multisets by forming their union, e.g., $\{2,3,5\} \cup \{2,2,6\} = \{2,3,5,2,2,6\}$
- *Size* : determine the total number of elements in a multiset, e.g., the size of $\{2,3,5,2,2,6\}$ is 6.
- *Count* : determine the number of occurrences of a specified element in a set, e.g., if m is a multiset object representing the multiset $\{2,4,5,2,2,4\}$ then `m.Count(2) = 3` and `m.Count(4) = 2` since 2 occurs three times and 4 occurs two times in m.

Two implementations of a C++ class *MultiSet* will be considered, they are based on:

1. An array of ints in which for an array a, the element $a[k]$ represents how many times k occurs in the multiset
2. A linked list of ints in which each element in a multiset appears once in the linked list.

For example, the multiset $\{2,2,7,3,4,3,2,7\}$ might be stored as diagrammed below as an array a or a linked-list *list*



In addition to the array or linked list, other information would be included with the C++ class or C struct that implemented a multiset. For example, the number of items stored in the multiset might be kept as a separate field (see the C++ class definition on the next page).

Part A: (*3 pts*) If the size is kept as a separate field, then the *Size* operation defined above is an $O(1)$ operation. If the size is NOT kept, but is determined by examining the array or linked list to determine the number of elements, then the *Size* operation is $O(n)$. In the latter case if arrays are used then n represents the largest element that can be a member of a multiset.

Explain why keeping the size field changes the complexity from $O(n)$ to $O(1)$. Also explain what n represents when linked-lists are used to implement multisets.

Either implementation can be based on the definition of a C++ class *MultiSet* partially shown below.

```
class MultiSet{
  private:
    int size;   // number of elements in multiset
    < declarations here depend on implementation >

  public:
    MultiSet();                 // construct an initially empty multiset
    ~MultiSet();
    int IsEmpty();              // returns 1 if multiset has no elements
    int Count(int element);     // returns # occurrences of element
    int Size();                 // returns # of elements in multiset
    // overload + so that it performs union

    friend MultiSet & operator + (const & MultiSet, const & MultiSet);
};
```

Part B: (*3 pts*) Explain why the constructor in the C++ class requires $O(n)$ time when arrays are used and $O(1)$ time when linked lists are used.

Part C: (*3 pts*) What is the complexity of the *Count* operation when arrays are used and when linked lists are used? Briefly justify your answer in both cases.

Part D: (*3pts*) Briefly describe what the method `~MultiSet` would do if a linked-list implementation of multisets is used.

PROBLEM 6 : (*Encore Multimax* 15 points)

The remaining questions assume a linked-list representation of multisets.

Part E: (*3 pts*) For the linked list implementation the following definitions are added to the private section of the class *MultiSet*.

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

Node * list;
```

Complete the method *Count* below:

```
int MultiSet::Count(int element)
{
    int total = 0;
    Node * temp = list;
    while (temp != NULL){




    }


}
```

Part F: (*12 pts*) A new method *Remove* is to be added to the implementations of multisets. The prototype is

```
void Remove(int element);
```

and this operation removes all occurrences of *element* from the multilist. For the array implementation the body of the method could consist of the single statement

```
if (0 <= element && element < setSize){
    elts[element] = 0;
}
```

(assuming that *elts* represents the array, that *setSize* represents the size of the array, and that both are defined in the private section). Why is the *if* statement guarding the assignment of 0 necessary?

With linked-lists, the *Remove* method is implemented as shown on the next page

```
void MultiSet::Remove(int element)
{
    DoRemove(list,element);
}
```

where the 'helper function' *DoRemove*, whose header is given below, actually does the work of removing the elements from the list. You are to implement the function *DoRemove*, complete it below the following header:

```
void DoRemove(Node * & list, int element)
```

Bonus Question: (*6 points*) Suppose a method *NumDistinct* is to be implemented that determines the number of distinct elements in a multiset, e.g., in $\{2, 2, 3, 2, 3, 4\}$ there are 3 distinct elements. Describe an implementation that executes in $O(n)$ time when arrays are used, where n is the largest possible element in a multiset. Describe a method and its complexity when linked lists are used.