Quiz 0

Data Structures, 1999.

June 9, 1999

Total marks: 30

Name:

Student #:

Answer each question in the space provided or on the back of a page with an indication of where to find the answer.

Q0 [6]

I have implemented the Function abstract data type with a doubly-linked list data-structure.

(a) Explain why the programmer using my implementation does not need to know that I have used doubly-linked list to implement the ADT.

The Function ADT describes the use of the class without reference to any particular implementation. Therefore the use of my class does not depend on which implementation techniques I have chosen.

(b) Explain why the data members of the class used to represent Functions is best declared as 'private'.

(i) This prevents client code from depending on the method of implementation. Only the operations documented in the ADT should be public.

(ii) Any error in the manipulation or interpretation of these data members is isolated to the member functions of the class. This makes error correction much easier, as the location of an error that corrupts or misinterprets the data members is narrowed down to the class. Client code is likely not to be to blame.

(c) Explain why, assuming templates are used, I do not need to know what the programmer using my implementation will use it for.

Using templates removes the need for the Function class to depend on the type of the data it is storing. Another programmer can use my class to store data of any type that supports certain operations (in this case assignment and comparison of the domain elements).

Q1 [5]

What are the relative advantages of an array based implementation of the Set ADT over a linked list implementation of the same ADT? What are the disadvantages?

There are three important advantages to the linked list implementation.

(i) Arrays, being fixed in size, likely waste space.

(ii) Arrays, being fixed in size, may run out of space, when there is lots of room on the heap.

(iii) Construction, destruction, copying, and assignment of array based implementations may waste time.

Some less important matters: Arrays run out of space predictably. Arrays may actually save space, since there is no pointer overhead, but you must correctly anticipate the required maximum size. If the array is kept sorted, then finding if an element is in the set is faster than with a linked list.

Q2 [4]

Consider the following specification.

- void foo() ;
- May change: i
- Precondition: $i > 0$
- Postcondition: $i' = -i$

For each of the following function definitions, **state whether or not it meets this specification.**

A subroutine meets its specification if whenever its precondition is true initially, it ensures that its postcondition is true and it does not change any variables it should not.

Note that this means it is the callers responsibility to ensure the precondition is true and the implementer's responsibility to ensure the postcondition is true.

In this case, a subroutine will meet the specification if, whenever the initial value of i is greater than 0, the final value of i is the negation of the initial value of i, and only i may change.

- 1. void foo(void){ i = -i ; } — *Yes. When i is initially positive, it is finally the negation of the initial value. (Note that even it i is negative or 0, the postcondition is still true, but this is irrelevant in determining if the specification is met.)*
  2. void foo(void){ j= -i; i=j ; } — *No. j is changed.*
  3. void foo(void){ assert(i>0) ; i = -i ; } — *Yes. In this case the precondition is checked. This is good practice to ensure the subroutine is not misused, but irrelevant to whether the specification is met.*
  4. void foo(void){ if(i > 0){ i = -i ; } else { i = 0 ; } } — *Again what happens when i is not initial positive is irrelevant. In this case setting i to 0 is arguably poor practice.*

Q3 [15]

A list is called *sorted* if data that comes nearer the beginning of the list is never larger than data that comes later in the list. E.g. 1,1,2,2,2,3,6,8,9,9 is sorted whereas 1,2,3,3,2,1 is not. **Write a function that eliminates all duplicates from a sorted list.** For example 1,1,2,2,2,6,8,9,9 becomes 1,2,6,8,9.

Lists are represented by pointers to nodes, with the NULL pointer (0) marking the end of the list. *Any nodes that are removed from the list should be deleted from the heap.* Nodes are defined as:

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

Complete this function:

```
void removeDuplicates(Node * &head)
{
      Node* current = head ;
      while( current != 0 ) {
           if( current->next != 0 && current->data == current->next->data ) {
                Node *zombie = current->next ;
                current->next = current->next->next ;
                delete zombie ; }
           else {
                current = current->next ; }
}
// Or
void removeDuplicates(Node * &head)
{
      Node* current = head ;
      while( current != 0 ) {
           while( current->next != 0 && current->data == current->next->data
          ) {
                Node *zombie = current->next ;
                current->next = current->next->next ;
                delete zombie ; }
           current = current->next ; }
}
```

Marking notes:

- *PNPD means Potential Null Pointer Dereference.*
- *DNPD means Definite Null Pointer Dereference*
- *NN means Not Needed*
- *PNI means potentially not initialized.*