Quiz 0

Engi 4892. Data Structures. 2004.

June 3, 2004

Total marks: 37
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 [**8**] Recall that the abstract fields of the list ADT contains the following abstract values, (where T is some type).

- len
  - A nonnegative integer representing the length of the list
- seq
  - A finite sequence of T's of length len representing the current contents of the list.

Use a precondition and a postcondition to define a mutator that exchanges (swaps) two items in the list. The precondition should specify that the two input parameters are valid indices.

- void swap(int p, int q) ;
- Precondition: $0 \leq p < len$ and $0 \leq q < len$
  (Also acceptable as an additional conjunct is $p \neq q$)
- Postcondition:
  - $\text{seq}'(p) = \text{seq}(q)$ and
  - $\text{seq}'(q) = \text{seq}(p)$ and
  - $len' = len$ and
  - for all $i \in \{0, ..., len - 1\}$, if $i \neq p$ and $i \neq q$ then $\text{seq}'(i) = \text{seq}(i)$

Marking notes: The concept of pre- and postconditions is fundamental and is used throughout this course and elsewhere in computing. The particular notation we are using is not the only one, but it is a good one, and as always it is important to understand and be able to use the notations of the organization you are with, in this case the notations used in this course.

Q1 [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. $\langle 1, 1, 2, 2, 2, 3, 6, 8, 9, 9\rangle$ is sorted whereas $\langle 1, 2, 3, 3, 2, 1\rangle$ is not.

Design a subroutine that takes as input parameters two sorted, singly-linked lists and produces as a result a new linked list that is sorted and which contains one copy of each node from either input list. Set a boolean output parameter, ok, to false if the task can not be completed (due to heap exhaustion), and to true otherwise. Note either or both of the input lists may be empty.

Singly-linked lists are represented by pointers to nodes, with a null pointer (0) marking the end of the list. Nodes are defined as:

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

Define any auxiliary subroutines you use. *Hint*: Writing a sort routine is entirely unnecessary; instead take advantage of the fact that the input lists are sorted.

A solution using a pointer to a link

```
void merge(Node * leftHead, Node *rightHead, Node *&newHead, bool &ok)
{
    Node **lastLink = &newHead ;
    while( rightHead != 0 || leftHead != 0 ) {
        *lastLink = new(nothrow) Node ;
        if( *lastLink == 0 ) { ok = false ; return ; }
        if( rightHead==0 || leftHead != 0 && leftHead->data < rightHead->data ) {
            (*lastLink)->data = leftHead->data ;
            leftHead = leftHead->next ; }
        else {
            assert(leftHead == 0 || rightHead !=0 && rightHead->data <= leftHead->data);
            (*lastLink)->data = rightHead->data ;
            rightHead = rightHead->next ; }
        lastLink = & (*lastLink)->next ; }
    *lastLink = 0 ;
    ok = true ;
}
```

A solution that doesn't use a pointer to a link

```
void merge(Node * leftHead, Node *rightHead, Node *&newHead, bool &ok)
{
    if( leftHead == 0 && rightHead == 0 ) {
        newHead = 0 ;
        ok = true ; }
    else {
        newHead = new(nothrow) Node ;
        if( newHead == 0 ) { ok = false ; return ; }
        if( rightHead == 0 || leftHead != 0 && leftHead->data < rightHead->data ) {
            newHead->data = leftHead->data ;
            leftHead = leftHead->next ; }
        else {
            newHead->data = rightHead->data ;
            rightHead = rightHead->next ; }

        Node *lastNode = newHead ;
        while( rightHead != 0 || leftHead != 0 ) {
            lastNode->next = new(nothrow) Node ;
            if( lastNode->next == 0 ) { ok = false ; return ; }
            if( rightHead==0 || leftHead != 0 && leftHead->data < rightHead->data ) {
                lastNode->next->data = leftHead->data ;
                leftHead = leftHead->next ; }
            else {
                lastNode->next->data = rightHead->data ;
                rightHead = rightHead->next ; }
            lastNode = lastNode->next ; }
        *lastNode->next = 0 ;
        ok = true ; }
}
```

There are lots of other perfectly good solutions, the two above are only illustrative. Marking notes:

- *Abbreviations used in marking*
  - *NN – code is not needed*
  - *VNI – variable not initialized. For example using newHead (an output parameter) as if it has a value).*
  - *PND – potential null dereference. For using leftHead->data when leftHead could be null.*

- *Mostly what I was looking for is that you could successfully write code to traverse the input lists and successfully build the output list.*
- *Typical errors:*
  - *Using as a loop guard* `rightHead !=0 && leftHead != 0` *but then after the loop not dealing with the fact that one list may not be empty, or dealing with this fact incorrectly.*
  - *Using as a loop guard* `rightHead !=0 || leftHead != 0` *but not checking to see whether the pointers are actually null in the loop body.*
  - *Failing to assign to newHead. Sometimes this seemed to stem from a failure to understand the nature of assignment, for example writing*

    ```
    Node *p = newHead ; p = new(nothrow) Node ;
    ```

    This will not assign to newHead.
  - *Discarding allocated nodes. Often nodes were allocated, but the last pointer to the node would then be thrown away. For example*

    ```
    Node *p = new(nothrow) Node; p->data = v ; p = p->next ;
    ```

    The last assignment overwrites the only pointer to the allocated Node.

Q2 [6]

We have looked at both an array-based and a linked-list based implementation of the List ADT.

(a) What are the main advantages of the linked-list based implementation?

- *The amount of space used is proportional to the amount needed.*
- *The length of the list is limited only by the available heap.*
- *Certain operations are quicker. Specifically: Construction and destruction may be quicker when the data type has a nontrivial constructor or destructor. Insertion and deletion require only pointer jumps and not moves of the data values.*

(b) What is the main advantage of the array-based implementation?

- *The retrieve operation is fast as it uses direct addressing.*

Marking Notes:

- *The use of the singular "main advantage" in the second part is deliberate. I was looking for the most important advantage. A number of less important advantages are also valid, but far less important.*

- *Numerous answers talked about the case where the array is sorted. For example "retrieve is faster if the array is sorted"; whether the array is sorted or not doesn't matter. Others wrote to the effect that "finding whether a given data value is in the list or not is faster, provided the array is sorted"; this is true (if one uses the a binary search algorithm), but it is a consequence of the fact that retrieve is faster.*

Q3 [8]

Below is a subroutine intended to delete the last item of a nonempty, singly-linked linked list.

```
void delLast( Node *& head ) {
      Node *p = head ;
      while( p->next->next != 0 ) {
           p = p->next ; }
      p->next = 0 ;
}
```

There are at least two logic errors in the routine. List all errors you can spot:

- *The algorithm will dereference a null pointer in the case where the list is of length 1.*
- *The algorithm does not delete the node from the heap. Thus it has a space leak.*

Marking note. Generally this question was well done.
A few answers seemed to suggest that cure to the first problem would be

```
void delLast( Node *& head ) {
      Node *p = head ;
      while( p->next != 0 ) {
           p = p->next ; }
      p = 0 ;
}
```

While this does not dereference any null pointers it also fails to alter the list in any way.