University of Toronto
csc148F—Introduction to Computing, Fall 1996

First Term Test — Horton's 1:00 section

Duration: 50 minutes

Aids allowed: None

Make sure that your examination booklet has 6 pages (including this one). Write your answers in the spaces provided. Write legibly.

Family Name: SOLUTIONS **Lecturer:** ____________________

First Name: ____________________ **Tutor:** ____________________

Student # : ____________________

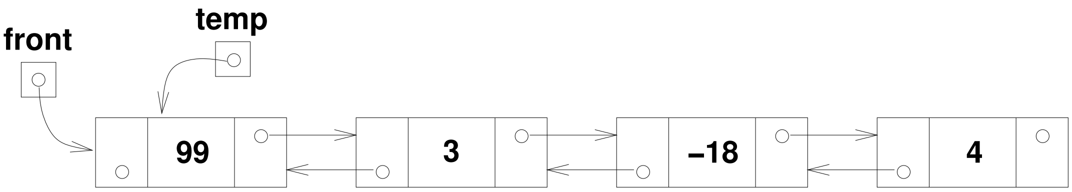
1.	________	/ 9
2.	________	/ 4
3.	________	/ 17
Total	________	/ 30

Question 1

[9 marks in total]

Assume that the type listNode below has been declared, and that some OOT code has been executed to create the doubly-linked list and temp pointer shown below. In the diagram, the `forwardLink` pointers are drawn pointing towards the right, and the `backwardLink` pointers are drawn pointing towards the left.

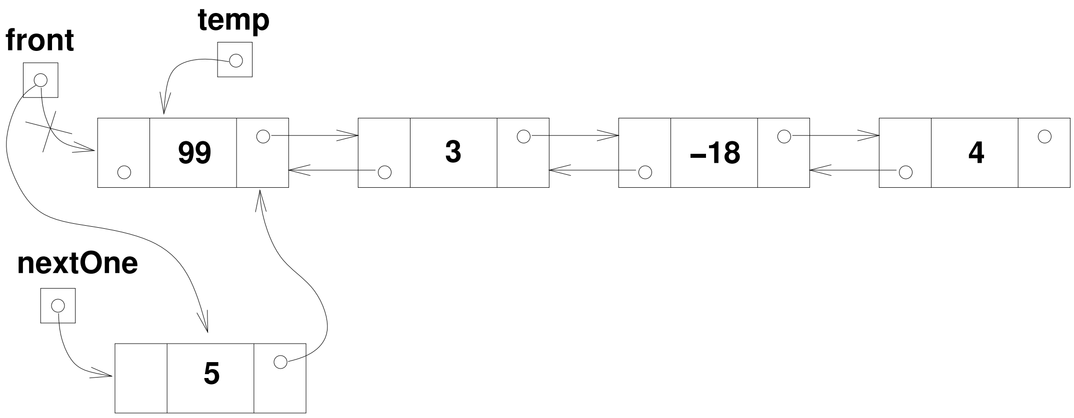
```
type listNode:
    record
      num : int
      forwardLink, backwardLink: ^listNode
    end record
```



(a) [5 marks]

Trace the following code and modify the diagram above to show the state of the doubly-linked list and the variables front, temp, and nextOne after the code has been executed.

```
var nextOne : ^listNode
new nextOne
nextOne -> num := 5
nextOne -> forwardLink := front
front := nextOne
```



(b) [2 marks]
After all of the code above has been executed, the doubly-linked list that remains is not properly formed. Write one additional OOT statement, to go at the end of the code, that will make it a properly-formed doubly-linked list.

```
Actually, two lines are needed:
     temp -> backwardLink := nextOne
     nextOne -> backwardLink := nil
Either was accepted for full marks.
```

(c) [2 marks]
If the statement `put front -> num` were added at the end of the above code, what output would result?

```
5
```

Question 2

[4 marks in total]

Consider the problem of search in an array. Assume that the array contains n elements to be searched through. One way to think about the efficiency of a search method is to consider how many elements it is capable of eliminating from consideration during a single iteration of its loop.

(a) [2 marks]
On a single iteration of the loop in a **linear** search procedure, how many elements of the array are removed from consideration? Circle the best answer.

1. roughly half of the elements originally under consideration, i.e., roughly $n/2$ elements
2. roughly half of the elements still under consideration

$\boxed{3}$ exactly 1 element

3. exactly n elements
4. exactly $n - 1$ elements

(b) [2 marks]
On a single iteration of the loop in a **binary** search procedure, how many elements of the array are removed from consideration? Circle the best answer.

1. roughly half of the elements originally under consideration, i.e., roughly $n/2$ elements

$\boxed{2}$ roughly half of the elements still under consideration

2. exactly 1 element
3. exactly n elements
4. exactly $n - 1$ elements

Question 3

[17 marks in total]

Note: The parts of this question can be answered independently of each other.

In assignment 1, you used a module called `CarQueue` to keep track of the cars waiting at the crosswalk. Below is the signature for that module.

```
% File "queue.tu"

unit
module CarQueue

    import Car in "car.tu"
    export enqueue, dequeue, isEmpty, isFull, printQueue

    type elementType : Car.carType

    % Add element newElement to the end of the queue.  Return true
    % if successful, false otherwise.  (Failure occurs if the queue
    % is already full.)
    function enqueue (newElement : elementType) : boolean

    % Remove the first element from the front of the queue and return it.
    % Precondition: isEmpty must not be true.
    function dequeue : elementType

    % Return true iff the queue is empty.
    function isEmpty : boolean

    % Return true iff the queue is full.
    function isFull : boolean

    % Initialize the queue to be empty.
    proc clear

    % Print all elements of the queue, each on a separate line, prefixed
    % by the given string.
    procedure printQueue (prefix : string)

end CarQueue
```

(a) [5 marks]

Suppose we want to define a function `inTheQueue` in the main program, and that `inTheQueue`'s purpose is to determine whether a given item is a member of the queue. The function has been started on the next page. Complete the function by filling in the two boxes.

Remember that this function is to go *outside* the module.

```
import CarQueue in "queue.tu", Car in "car.tu"

% This function returns true if the given item occurs in the queue,
% and false otherwise.  It has the unfortunate side effect of damaging
% the contents of the queue.
function inTheQueue (item : Car.carType) : boolean

    % Dequeue the elements of the queue one by one.  If any one of them
    % matches the item, then that item was in the queue, so return true
    % and stop looking.
    loop
                  +-----------------------+
        exit when |   CarQueue.isEmpty    |
                  +-----------------------+

        var c : Car.carType

        +----------------------------+
        |   c := CarQueue.dequeue    |
        +----------------------------+

        if Car.isSameCar (c, item) then
            result true
        end if

    end loop

    % No element of the queue matched the item we were searching for, so it
    % was not in the queue.  Return false.
    result false

end inTheQueue
```

(b) [4 marks]
The `CarQueue` module does not provide a way to determine the size of the queue. Two ways to do this are: export `numElements` from the module, or export a `sizeOf` function. Which would be a better design? Circle one.

export `numElements` | export a `sizeOf` function

Explain your answer:
A module should "hide" implementation details, such as variables. That is, it should not export them. It should instead export a subprogram that provides any needed access to the information stored in the variable.

If we were to export numElements, then any program that used the module might come to rely on the fact that there is an int variable within the module, called numElements, that stores the number of items in the queue. This would constrain any future changes to the module unnecessarily.

(c) [8 marks]
On the assignment, the queue was implemented using a "circular array". Another way to implement it is with a linked list. The element at the front of the queue (and thus next to leave) would be at the front of the linked list, and the element at the very back of the queue would be at the end of the linked list. An additional variable stores the number of elements currently in the queue.

Below are type and variable declarations for this data structure.

```
type node :
    record
        entry : Car.carType
        next : ^node
    end record
var front : ^node := nil
var numElements : int := 0
```

Complete the module's dequeue procedure, assuming that the linked list data structure just described has been declared appropriately within the module. Assume that the data structure has been properly maintained (*e.g.*, `numElements` correctly indicates the number of elements in the queue.) Make sure that you free any memory that is no longer needed. You may assume that the precondition is true; do not check it.

Remember that this function is *inside* the module.

```
% Remove the first element from the front of the queue and return it.
% Precondition: isEmpty must not be true.
function dequeue : elementType

     % Hold onto the front node of the queue so that it can later be freed.
     var temp : ^node := front

     % The answer to be sent back is the entry in that front node.
     var answer : Car.carType := front -> entry

     % Advance the front pointer and free the front node.  Then update
     % the numElements to reflect that we've dequeued one item.
     front := front -> next
     free temp
     numElements := numElements - 1

     % Finally, send back the dequeued item.
     result answer

end dequeue
```