Chapter 13

End of Chapter Exercises

1. *Explain why a stack is a LIFO (Last In, First Out) data structure)*

Stacks are built by (metaphorically) placing data items one on top of the other. They are implemented using a list structure, but items are only added to the head and removed from the head. Therefore, the last item added must be the first one that can be removed.

2. *In addition to the head pointer, what other pointers are needed to maintain a queue data structure?*

A queue end pointer is also needed as whilst records are removed from the head new records are added to the end (just like a real queue in a bank).

3. *What would happen if the pointer to the head of a linked list were accidentally deleted in a program?*

We would not be able to gain access to any records in the list because the entry point is the address held in the head pointer. Without the head pointer we cannot locate the first record in the list. In effect, the list would become lost.

4. *Given the added complexity involved in building a maintaining a linked list, why can't we just use arrays for storing lists of data?*

Because, typically, an array is a static data structure which means its size is fixed. If more storage is needed than the array has elements we are in trouble. Also, if only a few elements of the array are used we are needlessly using up memory space to maintain the empty elements. A list, on the other hand, can grow and shrink as necessary using only the amount of memory needed to store its current contents.

5. *What changes would be necessary to the FindSortedNode function (Solution 13.14) if our list of books were required to be sorted by descending order of book title?*

Simply change the condition from

```
WHILE (current ≠ NULL) AND (searchTitle > current→.bookTitle)
```

to

```
WHILE (current ≠ NULL) AND (searchTitle < current→.bookTitle)
```

6. *In the section on creating pointer variables we created a linked list element of type `PatientRecord` which was referenced by the variable `patientReference`. State what is wrong with the following statement and provide a correct version:*

```
patientReference.givenName ← 'Henry' ;
```

`patientReference` is a pointer/reference variable, not a record. It simply points to a record somewhere in memory. Therefore, to access the record it points to we must dereference it with the pointer operator →.

```
patientReference →.givenName ← 'Henry' ;
```

7. *Using the sub-programs given, write a new function `AmendNode` which will change the title of a specified book record in an unsorted list. The function will have three formal value parameters: `listHead`, `searchString`, and `replaceString`. `searchString` will hold the title of the book whose title we want to change. `replaceString` holds the new title of the book. The function will return a Boolean value true if the replacement was successful or false if a record matching `searchString` could not be found.*

```
FUNCTION AmendNode (→BookRecord:listHead, string:searchString,
                    string:replaceString) RETURNS Boolean
   →BookRecord:predecessor,
               theBook ;
   Boolean:success ;
   theBook ← FindNode (listHead, searchString);
   IF (theBook ≠ NULL) // Book found
      theBook→.bookTitle ← replaceString ;
      success ← True ;
   ELSE // Book not found
      success ← False ;
   ENDIF
   RETURN success ;
ENDFUNCTION
```

8. *Repeat the above exercise but this time for the sorted list*

`AmendNode` function for sorted list (uses `FindSortedNode`, Solution 13.14).

```
FUNCTION AmendNode (→BookRecord:listHead, string:searchString,
                    string:replaceString) RETURNS Boolean
   →BookRecord:predecessor,
               theBook ;
   Boolean:success ;
      success ← FindSortedNode (listHead, searchString,
                                REFERENCE:predecessor) ;
      IF (success) // Book found
      theBook ← predecessor→.next ;
      theBook→.bookTitle ← replaceString ;
   ENDIF
   RETURN success ;
```

```
ENDFUNCTION
```

9. *When comparing one string to another to test for alphabetic ordering, we would want 'e' to be judged to come before 'F'. The default would be for 'e'< 'F' to be False as 'e' comes after 'F' in the character set. The `AddSortedNode` procedure for an ordered linked list uses the `FindSortedNode` function (Solution 13.14). Amend `FindSortedNode` to ensure that 'Paul' would come before 'POET' in the list.*

This is really quite straightforward. `FindSortedNode` uses the following `WHILE` condition to find the appropriate place in the list:

```
WHILE (current ≠ NULL) AND (searchTitle > current→.bookTitle)
```

All we need to do is compare the upper case versions of the two titles. A small function `ToUpperString` (which uses the previous character function `ToUpper` from Chapter 10) to return the upper case version of a string would look thus:

```
FUNCTION ToUpperString(string:theString) RETURNS string
   integer:counter ;
   string:upper ;
   upper ← '' ;
   FOR counter GOES FROM 0 TO Length (theString)-1
      upper ← upper + ToUpper (theString [counter]) ;
   ENDFOR
   RETURN upper ;
ENDFUNCTION
```

All we then need to do is use this function in the `WHILE` loop:

```
WHILE (current ≠ NULL) AND (ToUpperString(searchTitle)>
                            ToUpperString(current→.bookTitle))
```

Projects

StockSnackz Vending Machine

No exercises for this project

Stocksfield Fire Service

Make linked list of codes & strings.

```
NEWTYPE fireCodeRecord IS
   RECORD
   character:code ;
   string:fightingMethod ;
   →fireCodeRecord:next ;
```

```
   ENDRECORD
NEWTYPE precautionCodeRecord IS
   RECORD
   character:code ;
   string:explosionRisk ;
   string:precaution ;
   string:treatmentMethod ;
    →precautionCodeRecord:next ;
   ENDRECORD

→fireCodeRecord:fcHead ;
→precautionCodeRecord:pcHead ;

// Code for building lists
…
//

FUNCTION processEAC (string:EAC), →fireCodeRecord:fcListHead,
                     →precautionCodeRecord:pcListHead)
                     RETURNS Boolean
   →fireCodeRecord:fcCurrent ;
   →precautionCodeRecord:pcCurrent ;
   Boolean:valid ;
   valid ← True ;
 // Deal with fire fighting method
   fcCurrent ← fcListHead ;
   WHILE NOT ((EAC [0]= fcCurrent→.code) OR (fcCurrent = NULL))
      fcCurrent ← fcCurrent→.next ;
   ENDWHILE
   IF fcCurrent ≠ NULL
      Display (fcCurrent→.fightingMethod) ;
   ELSE
      Display ('Invalid fire fighting code↲') ;
      valid ← false ;
   ENDIF

   // Deal with precautions
   pcCurrent ← pcListHead ;
   WHILE NOT ((EAC [1]= pcCurrent→.code) OR (pcCurrent = NULL))
      fcCurrent ← fcCurrent→.next ;
   ENDWHILE
```

```
   IF pcCurrent ≠ NULL
      IF (pcCurrent→.explosionRisk) ≠ '')
         Display (pcCurrent→.explosionRisk) ;
      ENDIF
      Display (pcCurrent→.precaution) ;
      Display (pcCurrent→.treatmentMethod) ;
   ELSE
      Display ('Invalid precaution ing code↲') ;
      valid ← false ;
   ENDIF

   // Deal with public hazard
   IF(EAC [2] = 'E')
      Display ('Public hazard↲') ;
   ELSE IF EAC [2] = ' ')
      Display ('No hazard↲') ;
   ELSE
      Display ('Invalid public hazard code↲') ;
      valid ← False ;
   ENDIF
   RETURN valid ;
ENDFUNCTION
```

Puzzle World: Roman Numerals & Chronograms

No solutions provided as they are highly dependent on how you have structured your own solutions over the previous chapters.

Pangrams: Holoalphabetic Sentences

No solutions provided as they are highly dependent on how you have structured your own solutions over the previous chapters.

Online Bookstore: ISBNs

No solutions provided as they are highly dependent on how you have structured your own solutions over the previous chapters.