Part 4 Further Problem-Solving and Programming Skills

Chapter 23
Computational Thinking and Problem-Solving

Learning objectives

By the end of this chapter you should be able to:

- show understanding of how to model a complex system by only including essential details
- write a binary search algorithm to solve a particular problem
- show understanding of the conditions necessary for the use of a binary search
- show understanding of how the performance of a binary search varies according to the number of data items
- write algorithms to:
  - implement an insertion sort
  - implement a bubble sort
  - find an item in each of the following: linked list, binary tree, hash table
  - insert an item into each of the following: stack, queue, linked list, binary tree, hash table
  - delete an item from each of the following: stack, queue, linked list
- show understanding that performance of a sort routine may depend on the initial order of the data and the number of data items
- show understanding that different algorithms which perform the same task can be compared by using criteria such as time taken to complete the task and memory used
- show understanding that an abstract data type (ADT) is a collection of data and a set of operations on those data
- show understanding that data structures not available as built-in types in a particular programming language need to be constructed from those data structures which are built-in within the language
- show how it is possible for ADTs to be implemented from another ADT
- describe the following ADTs and demonstrate how they can be implemented from appropriate built-in types or other ADTs: stack, queue, linked list, dictionary, binary tree.

23.01 What is computational thinking?

We have already been thinking computationally in Chapters 11 to 15. Here is the formal definition:

Computational thinking is a problem-solving process where a number of steps are taken in order to reach a solution, rather than relying on rote learning to draw conclusions without considering these conclusions.

Computational thinking involves abstraction, decomposition, data modelling, pattern recognition and algorithm design.

Abstraction

Abstraction involves filtering out information that is not necessary to solving the problem. There are many examples in everyday life where abstraction is used. In Chapter 11 (Section 11.01), we saw part of the underground map of London, UK. The purpose of this map is to help people plan their journey within London. The map does not show a geographical representation of the tracks of the underground train network.nor does it show the streets above ground. It shows the stations and which train lines connect the stations. In other words, the information that is not necessary when planning how to get from Kings Cross St. Pancras to Westminster is filtered out. The essential information we need to be able to plan our route is clearly represented.

Abstraction gives us the power to deal with complexity. An algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs. An abstract data type defines an abstract set of values and operations for manipulating those values.

Decomposition

Decomposition means breaking tasks down into smaller parts in order to explain a process more clearly. Decomposition is another word for step-wise refinement (covered in Chapter 12, Section 12.01). This led us to structured programming using procedures and functions with parameters, covered in Chapter 14 (Section 14.03 to 14.05).

Data modelling

Data modelling involves analysing and organising data. In Chapter 13 we met simple data types such as integer, character and Boolean. The string data type is a composite type: a sequence of characters. When we have groups of data items we used one-dimensional (1D) arrays to represent linear lists and two-dimensional (2D) arrays to represent tables or matrices. We stored data in text files. In Chapter 10, we used data modelling to design database tables.

We can set up abstract data types to model real-world concepts, such as records, queues or stacks. When a programming language does not have such data types built-in, we can define our own by building them from existing data types (see Section 23.03). There are more ways to build data models. In Chapter 27 we cover object-oriented programming where we build data models by defining classes. In Chapter 29 we model data using facts and rules. In Chapter 26 we cover random files.

Pattern recognition

Pattern recognition means looking for patterns or common solutions to common problems and exploiting these to complete tasks in a more efficient and effective way. There are many standard algorithms to solve standard problems, such as insertion sort or binary search (see Section 23.02).

Algorithm design

Algorithm design involves developing step-by-step instructions to solve a problem (see Chapter 11).

23.02 Standard algorithms

Bubble sort

In Chapter 11, we developed the algorithm for a bubble sort (Worked Example 11.12).

Discussion Point:

What were the essential features of a bubble sort?

TASK 23.01

Write program code for the most efficient bubble sort algorithm. Assume that the items to be sorted are stored in a 1D array with n elements.

Insertion sort

Imagine you have a number of cards with a different value printed on each card. How would you sort these cards into order of increasing value?

You can consider the pile of cards as consisting of a sorted part and an unsorted part. Place the unsorted cards in a pile on the table. Hold the sorted cards as a pack in your hand. To start with only the first (top) card is sorted. The card on the top of the pile on the table is the next card to be inserted. The last (bottom) card in your hand is your current card.

Figure 23.01 shows the sorted cards in your hand as blue and the pile of unsorted cards as white. The next card to be inserted is shown in red. Each column shows the state of the pile as the cards are sorted.

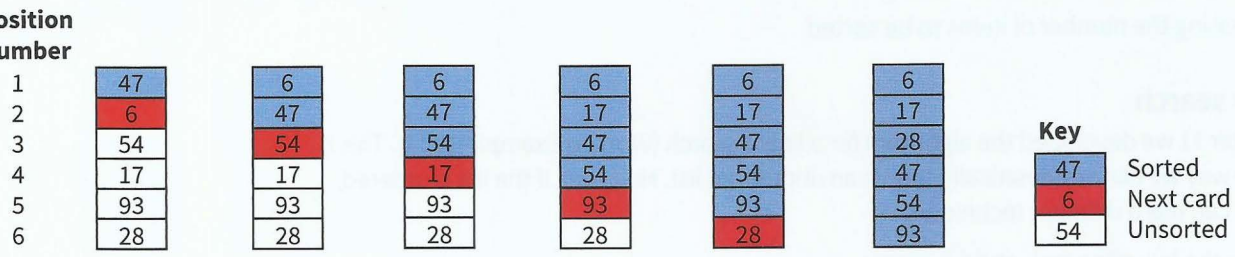


Figure 23.01 Sorting cards

Repeat the following steps until all cards in the unsorted pile have been inserted into the correct position:

1 Repeat until the card to be inserted has been placed in its correct position.

1.1 Compare the current card with the card to be inserted.

1.2 If the card to be inserted is greater than the current card, insert it below the current card.

1.3 Otherwise, if there is a card above the current card in your hand, make this your new current card.

1.4 If there is no new current card, place the card to be inserted at the top of the sorted pile.

What happens when you work through the sorted cards to find the correct position for the card to be inserted? In effect, as you consider the cards in your hand, you move the current card down a position. If the value of the card to be inserted is smaller than the last card you considered, then the card is inserted at the top of the pile (position 1).

This method is known as an insertion sort. It is a standard sort method.

We can write this algorithm using pseudocode. Assume the values to be sorted are stored in a 1D array, `List:`

```
FOR Pointer ← 2 TO NumberOfitems
   ItemToBeInserted ← List[Pointer]
   CurrentItem ← Pointer - 1 // pointer to last item in sorted part of list
   WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem > 0)
      List[CurrentItem + 1]← List[CurrentItem]// move current item down
      CurrentItem ← CurrentItem - 1 // look at the item above
   ENDWHILE
   List[CurrentItem + 1]← ItemToBeInserted // insert item
ENDFOR
```


TASK 23.02

1 Dry-run the insertion sort algorithm using a trace table. Assume the list consists of the following six items in the order given: 53, 21, 60, 18, 42, 19.

2 Write program code for the insertion sort algorithm. Assume that the items to be sorted are stored in a 1D array with n elements.

Extension question 23.01

Investigate the performances of the insertion sort and the bubble sort by:

- varying the initial order of the items
- increasing the number of items to be sorted.

Binary search

In Chapter 11 we developed the algorithm for a linear search (Worked Example 11.11). This is the only way we can systematically search an unordered list. However, if the list is ordered, then we can use a different technique.

Consider the following real-world example.

If you want to look up a word in a dictionary, you are unlikely to start searching for the word from the beginning of the dictionary. Suppose you are looking for the word 'quicksort'. You look at the middle entry of the dictionary (approximately) and find the word 'magnetic'. 'quicksort' comes after 'magnetic', so you look in the second half of the dictionary. Again you look at the entry in the middle of this second half of the dictionary (approximately) and find

the word 'report'. 'quicksort' comes before 'report', so you look in the third quarter. You can keep looking at the middle entry of the part which must contain your word, until you find the word. If the word does not exist in the dictionary, you will have no entries in the dictionary left to find the middle of.

This method is known as a **binary search**. It is a standard method.

KEY TERMS

Binary search: repeated checking of the middle item in an ordered search list and discarding the half of the list which does not contain the search item

We can write this algorithm using pseudocode. Assume the values are sorted in ascending order and stored in a 1D array, `List` of size `MaxItems`.

```
Found ← FALSE
SearchFailed ← FALSE
First ← 1
Last ← MaxItems // set boundaries of search area
WHILE NOT Found AND NOT SearchFailed
   Middle ← (First + Last) DIV 2 // find middle of current search area
   IF List[Middle] = SearchItem
      THEN
         Found ← TRUE
      ELSE
         IF First >= Last // no search area left
            THEN
               SearchFailed ← TRUE
            ELSE
               IF List[Middle] > SearchItem
                  THEN  // must be in first half
                     Last ← Middle - 1  // move upper boundary
                  ELSE  // must be in second half
                     First ← Middle + 1  // move lower boundary
               ENDIF
         ENDIF
   ENDIF
ENDWHILE
IF Found = TRUE
   THEN
      OUTPUT Middle // output position where item was found
   ELSE
      OUTPUT "Item not present in array"
ENDIF
```

TASK 23.03

Dry-run the binary search algorithm using a trace table. Assume the list consists of the following 20 items in the order given: 7, 12, 19, 23, 27, 33, 37, 41, 45, 56, 59, 60, 62, 71, 75, 80, 84, 88, 92, 99.

Search for the value 60. How many times did you have to execute the `While` loop?

Dry-run the algorithm again, this time searching for the value 34. How many times did you have to execute the `While` loop?

Discussion Point:

Compare the binary-search algorithm with the linear-search algorithm. If the array contains n items, how many times on average do you need to test a value when using a binary search and how many times on average do you need to test a value when using a linear search? Can you describe how the search time varies with increasing n?

23.03 Abstract data types (ADTs)

An **abstract data type** is a collection of data. When we want to use an abstract data type, we need a set of basic operations:

- create a new instance of the data structure
- find an element in the data structure
- insert a new element into the data structure
- delete an element from the data structure
- access all elements stored in the data structure in a systematic manner.

KEY TERMS

Abstract data type: a collection of data with associated operations

The remainder of this chapter describes the following ADTs: stack, queue, linked list, binary tree, hash table and dictionary. It also demonstrates how they can be implemented from appropriate built-in types or other ADTs.

23.04 Stacks

What are the features of a stack in the real world? To make a stack, we pile items on top of each other. The item that is accessible is the one on top of the stack. If we try to find an item in the stack and take it out, we are likely to cause the pile of items to collapse.

Figure 23.02 shows how we can represent a stack when we have added four items in this order: A, B, C, D. Note that the slots are shown numbered from the bottom as this is more intuitive.

The `BaseOfStackPointer` will always point to the first slot in the stack. The `TopOfStackPointer` will point to the last element pushed onto the stack. When an element is removed from the stack, the `TopOfStackPointer` will decrease to point to the element now at the top of the stack.

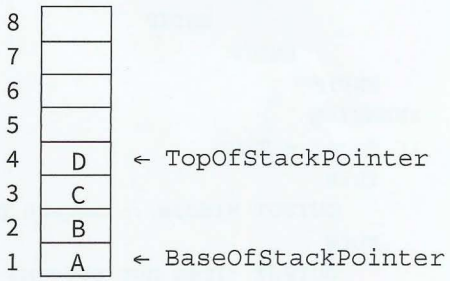


Figure 23.02 A stack

23.05 Queues

What are the features of a queue in the real world? When people form a queue, they join the queue at the end. People leave the queue from the front of the queue. If it is an orderly queue, no-one pushes in between and people don't leave the queue from any other position.

Figure 23.03 shows how we can represent a queue when five items have joined the queue in this order: A, B, C, D, E.

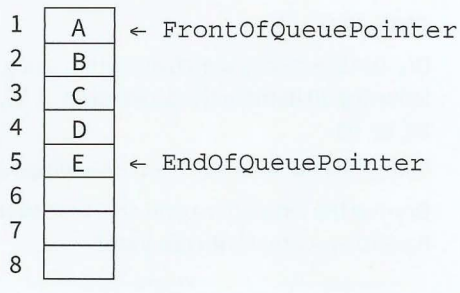


Figure 23.03 A queue

When the item at the front of the queue leaves, we need to move all the other items one slot forward. This would involve a lot of moving of data. A more efficient way to make use of the slots is the concept of a 'circular' queue. Pointers show where the front and end of the queue are. Eventually the queue will 'wrap around' to the beginning. Figure 23.04 shows a circular queue after 11 items have joined and five items have left the queue.

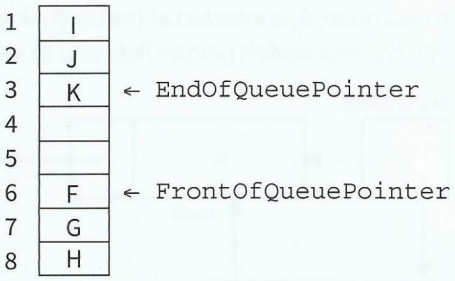


Figure 23.04 A circular queue

23.06 Linked lists

In Chapter 11 we used an array as a linear list. In a linear list, the list items are stored in consecutive locations. This is not always appropriate. Another method is to store an individual list item in whatever location is available and link the individual item into an ordered sequence using pointers.

An element of a list is called a **node**. A node can consist of several data items and a **pointer**, which is a variable that stores the address of the node it points to.

A pointer that does not point at anything is called a **null pointer.** It is usually represented by ∅. A variable that stores the address of the first element is called a **start pointer.**

KEY TERMS

Node: an element of a list
Pointer: a variable that stores the address of the node it points to
Null pointer: a pointer that does not point at anything
Start pointer: a variable that stores the address of the first element of a linked list

In Figure 23.05, the data value in the node box represents the key field of that node. There are likely to be many data items associated with each node. The arrows represent the pointers. It does not show at which address a node is stored, so the diagram does not give the value of the pointer, only where it conceptually links to.

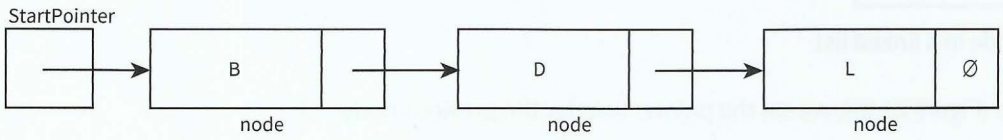


Figure 23.05 Conceptual diagram of a linked list

A new node, A, is inserted at the beginning of the list. The content of `StartPointer` is copied into the new node's pointer field and `StartPointer` is set to point to the new node, A.

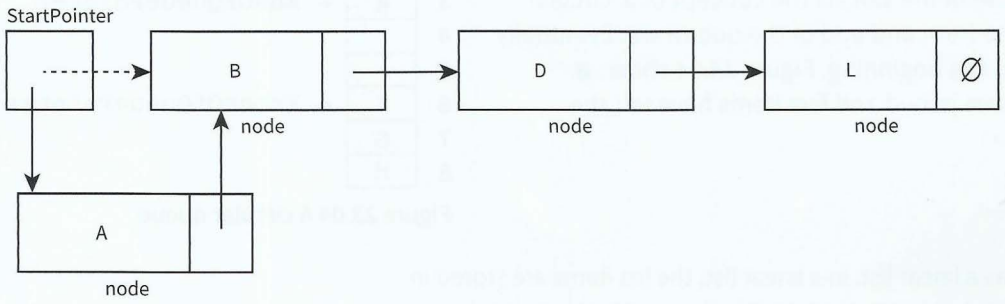


Figure 23.06 Conceptual diagram of adding a new node to the beginning of a linked list

In Figure 23.07, a new node, P, is inserted at the end of the list. The pointer field of node L points to the new node, P. The pointer field of the new node, P, contains the null pointer.

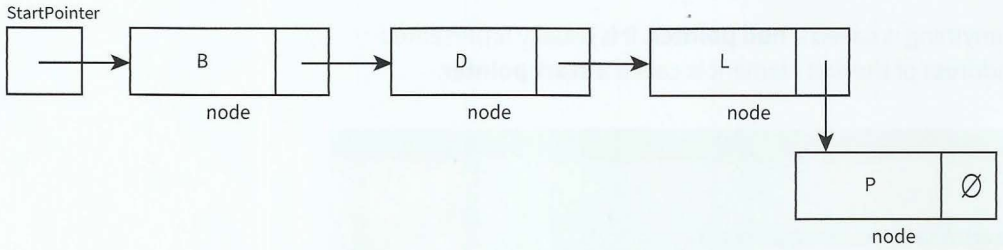


Figure 23.07 Conceptual diagram of adding a new node to the end of a linked list

To delete the first node in the list (Figure 23.08), we copy the pointer field of the node to be deleted into `StartPointer`.

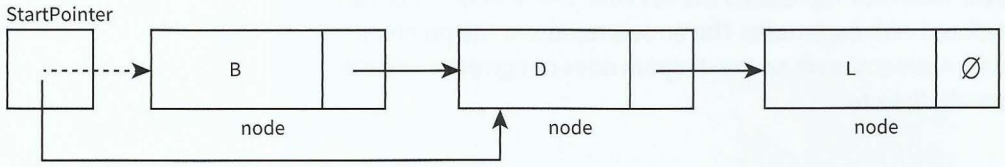


Figure 23.08 Deleting the first node in a linked list

To delete the last node in the list (Figure 23.09), we set the pointer field for the previous node to the null pointer.

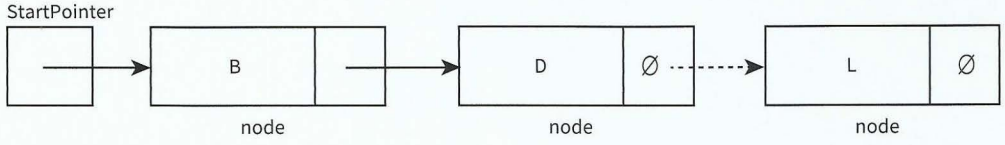


Figure 23.09 Conceptual diagram of deleting the last node of a linked list

Sometimes the nodes are linked together in order of key field value to produce an ordered linked list. This means a new node may need to be inserted or deleted from between two existing nodes.

To insert a new node, C, between existing nodes, B and D (Figure 23.10), we copy the pointer field of node B into the pointer field of the new node, C. We change the pointer field of node B to point to the new node, C.

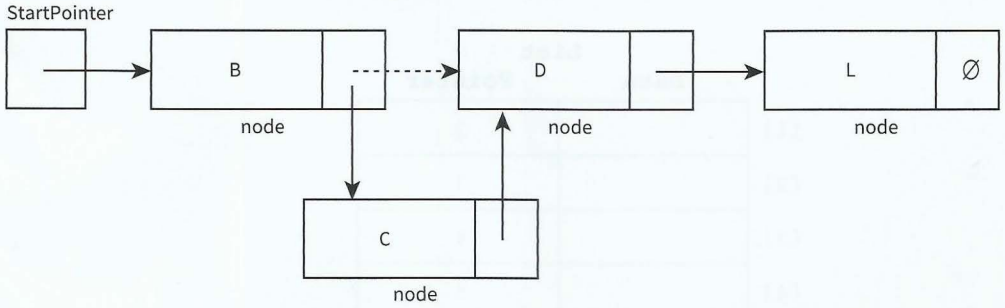


Figure 23.10 Conceptual diagram of adding a new node into a linked list

To delete a node, D, within the list (Figure 23.11), we copy the pointer field of the node to be deleted, D, into the pointer field of node B.

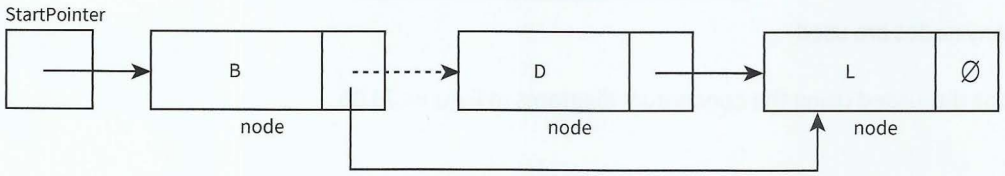


Figure 23.11 Conceptual diagram of deleting a node within a linked list

Remember that, in real applications, the data would consist of much more than a key field and one data item. This is why linked lists are preferable to linear lists. When list elements need reordering, only pointers need changing in a linked list. In a linear list, all data items would need to be moved.

Using linked lists saves time, however we need more storage space for the pointer fields.

In Chapter 16 we looked at composite data types, in particular the user-defined record type. We grouped together related data items into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

We can store the linked list in an array of records. One record represents a node and consists of the data and a pointer. When a node is inserted or deleted, only the pointers need to change. A pointer value is the array index of the node pointed to.

Unused nodes need to be easy to find. A suitable technique is to link the unused nodes to form another linked list: the free list. Figure 23.12 shows our linked list and its free list.

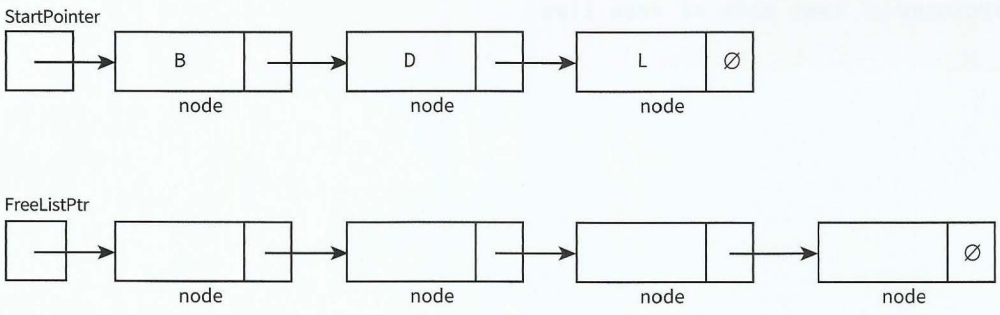


Figure 23.12 Conceptual diagram of a linked list and a free list

When an array of nodes is first initialised to work as a linked list, the linked list will be empty. So the start pointer will be the null pointer. All nodes need to be linked to form the free list. Figure 23.13 shows an example of an implementation of a linked list before any data is inserted into it.

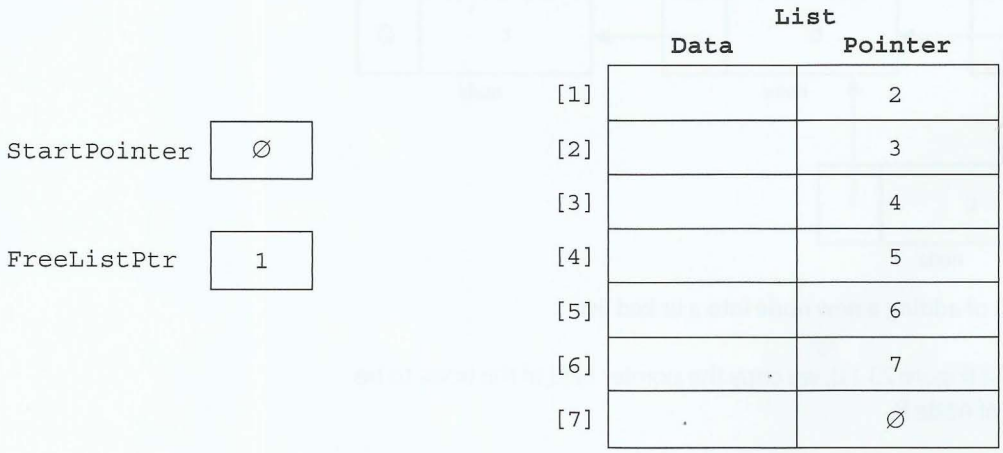


Figure 23.13 A linked list before any nodes are used

We now code the basic operations discussed using the conceptual diagrams in Figures 23.05 to 23.12.

Create a new linked list

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = 0
// Declare record type to store data and pointer
TYPE ListNode
   DECLARE Data    : STRING
   DECLARE Pointer : INTEGER
ENDTYPE
DECLARE StartPointer : INTEGER
DECLARE FreeListPtr : INTEGER
DECLARE List[1 : 7] OF ListNode

PROCEDURE InitialiseList
   StartPointer ← NullPointer     // set start pointer
   FreeListPtr ← 1                // set starting position of free list
   FOR Index ← 1 TO 6             // link all nodes to make free list
      List[Index].Pointer ← Index + 1
   ENDFOR
   List[7].Pointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Insert a new node into an ordered linked list

```
PROCEDURE InsertNode(NewItem)
   IF FreeListPtr <> NullPointer
      THEN  // there is space in the array
         // take node from free list and store data item
         NewNodePtr ← FreeListPtr
         List[NewNodePtr].Data ← NewItem
         FreeListPtr ← List[FreeListPtr].Pointer
         // find insertion point
         ThisNodePtr ← StartPointer           // start at beginning of list
         WHILE ThisNodePtr <> NullPointer          // while not end of list
               AND List[ThisNodePtr].Data < NewItem
            PreviousNodePtr ← ThisNodePtr          // remember this node
                                    // follow the pointer to the next node
            ThisNodePtr ← List[ThisNodePtr].Pointer
         ENDWHILE
         IF PreviousNodePtr = StartPointer
            THEN  // insert new node at start of list
               List[NewNodePtr].Pointer ← StartPointer
               StartPointer ← NewNodePtr
            ELSE  // insert new node between previous node and this node
               List[NewNodePtr].Pointer ← List[PreviousNodePtr].Pointer
               List[PreviousNodePtr].Pointer ← NewNodePtr
         ENDIF
   ENDIF
ENDPROCEDURE
```

After three data items have been added to the linked list, the array contents are as shown in Figure 23.14.

StartPointer: 1

FreeListPtr: 4

	List	
	Data	Pointer
[1]	B	2
[2]	D	3
[3]	L	Ø
[4]		5
[5]		6
[6]		7
[7]		Ø

Figure 23.14 Linked list of three nodes and free list of four nodes

Find an element in an ordered linked list

```
FUNCTION FindNode(DataItem) RETURNS INTEGER  // returns pointer to node
   CurrentNodePtr ← StartPointer  // start at beginning of list
   WHILE CurrentNodePtr <> NullPointer  // not end of list
         AND List[CurrentNodePtr].Data <> DataItem  // item not found
      // follow the pointer to the next node
      CurrentNodePtr ← List[CurrentNodePtr].Pointer
   ENDWHILE
   RETURN CurrentNodePtr  // returns NullPointer if item not found
ENDFUNCTION
```

Delete a node from an ordered linked list

```
PROCEDURE DeleteNode(DataItem)
   ThisNodePtr ← StartPointer                    // start at beginning of list
   WHILE ThisNodePtr <> NullPointer              // while not end of list
         AND List[ThisNodePtr].Data <> DataItem  // and item not found
      PreviousNodePtr ← ThisNodePtr  // remember this node
                                     // follow the pointer to the next node
      ThisNodePtr ← List[ThisNodePtr].Pointer
   ENDWHILE
   IF ThisNodePtr <> NullPointer  // node exists in list
      THEN
         IF ThisNodePtr = StartPointer  // first node to be deleted
            THEN
               StartPointer ← List[StartPointer].Pointer
            ELSE
               List[PreviousNodePtr] ← List[ThisNodePtr].Pointer
         ENDIF
         List[ThisNodePtr].Pointer ← FreeListPtr
         FreeListPtr ← ThisNodePtr
   ENDIF
ENDPROCEDURE
```

Access all nodes stored in the linked list

```
PROCEDURE OutputAllNodes
   CurrentNodePtr ← StartPointer // start at beginning of list
   WHILE CurrentNodePtr <> NullPointer  // while not end of list
      OUTPUT List[CurrentNodePtr].Data
      // follow the pointer to the next node
      CurrentNodePtr ← List[CurrentNodePtr].Pointer
   ENDWHILE
ENDPROCEDURE
```


TASK 23.04

Convert the pseudocode for the linked-list handling subroutines to program code. Incorporate the subroutines into a program and test them.

Note that a stack ADT and a queue ADT can be treated as special cases of linked lists. The linked list stack only needs to add and remove nodes from the front of the linked list. The linked list queue only needs to add nodes to the end of the linked list and remove nodes from the front of the linked list.

TASK 23.05

Write program code to implement a stack as a linked list. Note that the adding and removing of nodes is much simpler than for an ordered linked list.

TASK 23.06

Write program code to implement a queue as a linked list. You may find it helpful to introduce another pointer that always points to the end of the queue. You will need to update it when you add a new node to the queue.

23.07 Binary trees

In the real world, we draw tree structures to represent hierarchies. For example, we can draw a family tree showing ancestors and their children. A binary tree is different to a family tree because each node can have at most two 'children'.

In computer science binary trees are used for different purposes. In Chapter 20 (Section 20.05), you saw the use of a binary tree as a syntax tree. In this chapter, you will use an ordered binary tree ADT (such as the one shown in Figure 23.15) as a binary search tree.

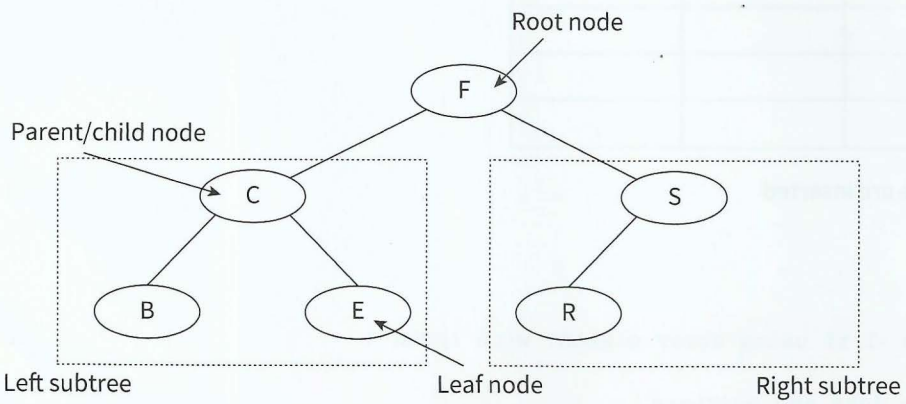


Figure 23.15 Conceptual diagram of an ordered binary tree

Nodes are added to an ordered binary tree in a specific way:

Start at the root node as the current node.
Repeat
If the data value is greater than the current node's data value, follow the right branch.
If the data value is smaller than the current node's data value, follow the left branch.
Until the current node has no branch to follow.
Add the new node in this position.

For example, if we want to add a new node with data value D to the binary tree in Figure 23.15, we execute the following steps:

1. Start at the root node.
2. D is smaller than F, so turn left.
3. D is greater than C, so turn right.
4. D is smaller than E, so turn left.
5. There is no branch going left from E, so we add D as a left child from E (see Figure 23.16).

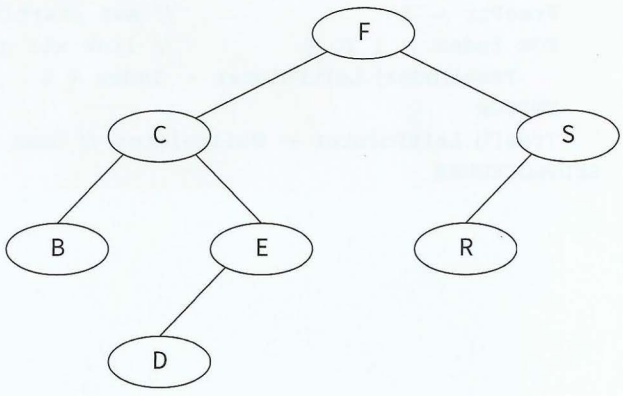


Figure 23.16 Conceptual diagram of adding a node to an ordered binary tree

This type of tree has a special use as a search tree. Just like the binary search applied to an ordered linear list, the binary

search tree gives the benefit of a faster search than a linear search or searching a linked list. The ordered binary tree also has a benefit when adding a new node: other nodes do not need to be moved, only a left or right pointer needs to be added to link the new node into the existing tree.

We can store the binary tree in an array of records (see Figure 23.17). One record represents a node and consists of the data and a left pointer and a right pointer. Unused nodes are linked together to form a free list.

RootPointer | Ø |

FreePtr | 1 |

Tree

	LeftPointer	Data	RightPointer
[1]	2		
[2]	3		
[3]	4		
[4]	5		
[5]	6		
[6]	7		
[7]	Ø		

Figure 23.17 Binary tree before any nodes are inserted

Create a new binary tree

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = 0
// Declare record type to store data and pointers
TYPE TreeNode
   DECLARE Data : STRING
   DECLARE LeftPointer : INTEGER
   DECLARE RightPointer : INTEGER
ENDTYPE
DECLARE RootPointer : INTEGER
DECLARE FreePtr : INTEGER
DECLARE Tree[1 : 7] OF TreeNode
PROCEDURE InitialiseTree
   RootPointer ← NullPointer  // set start pointer
   FreePtr ← 1                // set starting position of free list
   FOR Index ← 1 TO 6         // link all nodes to make free list
      Tree[Index].LeftPointer ← Index + 1
   ENDFOR
   Tree[7].LeftPointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Insert a new node into a binary tree

```
PROCEDURE InsertNode(NewItem)
   IF FreePtr <> NullPointer
      THEN  // there is space in the array
         // take node from free list, store data item and set null pointers
         NewNodePtr ← FreePtr
         FreePtr ← Tree[FreePtr].LeftPointer
         Tree[NewNodePtr].Data ← NewItem
         Tree[NewNodePtr].LeftPointer ← NullPointer
         Tree[NewNodePtr].RightPointer ← NullPointer
         // check if empty tree
         IF RootPointer = NullPointer
            THEN  // insert new node at root
               RootPointer ← NewNodePtr
            ELSE  // find insertion point
               ThisNodePtr ← RootPointer   // start at the root of the tree
               WHILE ThisNodePtr <> NullPointer   // while not a leaf node
                  PreviousNodePtr ← ThisNodePtr  // remember this node
                  IF Tree[ThisNodePtr].Data > NewItem
                     THEN  // follow left pointer
                        TurnedLeft ← TRUE
                        ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                     ELSE // follow right pointer
                        TurnedLeft ← FALSE
                        ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                  ENDIF
               ENDWHILE
               IF TurnedLeft = TRUE
                  THEN
                     Tree[PreviousNodePtr].LeftPointer ← NewNodePtr
                  ELSE
                     Tree[PreviousNodePtr].RightPointer ← NewNodePtr
               ENDIF
         ENDIF
   ENDIF
ENDPROCEDURE
```

Find a node in a binary tree

```
FUNCTION FindNode(SearchItem) RETURNS INTEGER  // returns pointer to node
   ThisNodePtr ← RootPointer     // start at the root of the tree
   WHILE ThisNodePtr <> NullPointer   // while a pointer to follow
         AND Tree[ThisNodePtr].Data <> SearchItem // and search item not found
      IF Tree[ThisNodePtr].Data > SearchItem
         THEN  // follow left pointer
            ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
         ELSE // follow right pointer
            ThisNodePtr ← Tree[ThisNodePtr].RightPointer
      ENDIF
   ENDWHILE
   RETURN ThisNodePtr // will return null pointer if search item not found
ENDFUNCTION
```

TASK 23.07

Write program code to implement a binary search tree.

23.08 Hash tables

If we want to store records in an array and have direct access to records, we can use the concept of a hash table.

The idea behind a hash table is that we calculate an address (the array index) from the key value of the record and store the record at this address. When we search for a record, we calculate the address from the key and go to the calculated address to find the record. Calculating an address from a key is called 'hashing'.

Finding a hashing function that will give a unique address from a unique key value is very difficult. If two different key values hash to the same address this is called a 'collision'. There are different ways to handle collisions:

- chaining: create a linked list for collisions with start pointer at the hashed address
- using overflow areas: all collisions are stored in a separate overflow area, known as 'closed hashing'
- using neighbouring slots: perform a linear search from the hashed address to find an empty slot, known as 'open hashing'.

WORKED EXAMPLE 23.01

Calculating addresses in a hash table

Assume we want to store customer records in a 1D array `HashTable[0 : n]`. Each customer has a unique customer ID, an integer in the range 10001 to 99999.

We need to design a suitable hashing function. The result of the hashing function should be such that every index of the array can be addressed directly. The simplest hashing function gives us addresses between 0 and n:

```
FUNCTION Hash(Key) RETURNS INTEGER
   Address ← Key MOD(n + 1)
   RETURN Address
ENDFUNCTION
```

For illustrative purposes, we choose n to be 9. Our hashing function is:

```
Index ← CustomerID MOD 10
```

We want to store records with customer IDs: 45876, 32390, 95312, 64636, 23467. We can store the first three records in their correct slots, as shown in Figure 23.18.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876			

Figure 23.18 A hash table without collisions

The fourth record key (64636) also hashes to index 6. This slot is already taken; we have a collision. If we store our record here, we lose the previous record. To resolve the collision, we can choose to store our record in the next available space, as shown in Figure 23.19.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636		

Figure 23.19 A hash table with a collision resolved by open hashing

The fifth record key (23467) hashes to index 7. This slot has been taken up by the previous record, so again we need to use the next available space (Figure 23.20).

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636	23467	

Figure 23.20 A hash table with two collisions resolved by open hashing

When searching for a record, we need to allow for these out-of-place records. We know if the record we are searching for does not exist in the hash table when we come across an unoccupied slot.

We will now develop algorithms to insert a record into a hash table and to search for a record in the hash table using its record key.

- The hash table is a 1D array `HashTable[0 : Max] OF Record`.
- The records stored in the hash table have a unique key stored in field `Key`.

Insert a record into a hash table

```
PROCEDURE Insert(NewRecord)
   Index ← Hash(NewRecord.Key)
   WHILE HashTable[Index] NOT empty
      Index ← Index + 1 // go to next slot
      IF Index > Max // beyond table boundary?
         THEN // wrap around to beginning of table
            Index ← 1
      ENDIF
   ENDWHILE
   HashTable[Index] ← NewRecord
ENDPROCEDURE
```

Find a record in a hash table

```
FUNCTION FindRecord(SearchKey) RETURNS Record
   Index ← Hash(SearchKey)
   WHILE (HashTable[Index].Key <> SearchKey) AND (HashTable[Index] NOT empty)
      Index ← Index + 1 // go to next slot
      IF Index > Max // beyond table boundary?
         THEN // wrap around to beginning of table
            Index ← 0
      ENDIF
   ENDWHILE
   IF HashTable[Index] NOT empty // if record found
      THEN
         RETURN HashTable[Index] // return the record
   ENDIF
ENDFUNCTION
```

23.09 Dictionaries

A real-world dictionary is a collection of key–value pairs. The key is the term you use to look up the required value. For example, if you use an English–French dictionary to look up the English word 'book', you will find the French equivalent word 'livre'. A real-world dictionary is organised in alphabetical order of keys.

An ADT dictionary in computer science is implemented using a hash table, so that a value can be looked up using a direct-access method.

Python has a built-in ADT dictionary. The hashing function is determined by Python. For VB and Pascal, we need to implement our own.

Here are some examples of Python dictionaries:

```
EnglishFrench = {} # empty dictionary
EnglishFrench["book"] = "livre" # add a key-value pair to the dictionary
EnglishFrench["pen"] = "stylo"

print(EnglishFrench["book"]) # access a value in the dictionary

# alternative method of setting up a dictionary
ComputingTerms = {"Boolean" : "can be TRUE or FALSE", "Bit" : "0 or 1"}

print(ComputingTerms["Bit"])
```

There are many built-in functions for Python dictionaries. These are beyond the scope of this book. However, we need to understand how dictionaries are implemented. The following pseudocode shows how to create a new dictionary.


```
TYPE DictionaryEntry
   DECLARE Key   : STRING
   DECLARE Value : STRING
ENDTYPE
DECLARE EnglishFrench[0 : 999] OF DictionaryEntry  // empty dictionary
```

TASK 23.08

Write pseudocode to:

- insert a key–value pair into a dictionary
- look up a value in a dictionary.

Use the hashing function from Worked Example 23.01.

Summary

- Computational thinking is a problem-solving process.
- Standard algorithms include bubble sort, insertion sort, linear search and binary search.
- Abstract data types (ADTs) include records, stacks, queues, linked lists, binary trees, hash tables and dictionaries.
- Basic operations required for an ADT include creating an ADT and inserting, finding or deleting an element of an ADT.

Exam-style Questions

1 **a** Complete the algorithm for a binary search function `FindName`.

The data being searched is stored in the array `Names[0 : 50]`.
The name to be searched for is passed as a parameter.

```
FUNCTION FindName(s : STRING) RETURNS INTEGER
   Index ← -1
   First ← 0
   Last ← 50
   WHILE (Last >= First) AND .....................................
      Middle ← (First + Last) DIV 2
      IF Names[Middle] = s
           THEN
              Index ← Middle
           ELSE
              IF ..................................................
                 THEN
                    Last  ← Middle + 1
                 ELSE
                    ..............................................
              ENDIF
      ENDIF
   ENDWHILE
ENDFUNCTION
```

[3]

b The binary search does not work if the data in the array being searched is..... [1]

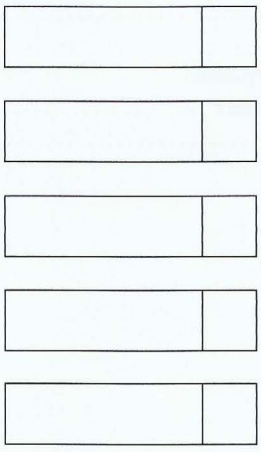
c What does the function `FindName` return when:

i the name searched for exists in the array

ii the name searched for does not exist in the array? [2]

2 A queue Abstract Data Type (ADT) is to be implemented as a linked list of nodes. Each node is a record, consisting of a data field and a pointer field. The queue ADT also has a `FrontOfQueue` pointer and an `EndOfQueue` pointer associated with it. The possible queue operations are: `JoinQueue` and `LeaveQueue`.

a **i** Add labels to the diagram to show the state of the queue after three data items have been added to the queue in the given order: Apple, Pear, Banana.



[5]

ii Add labels to the diagram to show how the unused nodes are linked to form a list of free nodes. This list has a `StartOfFreeList` pointer associated with it. [2]

b **i** Using program code, declare the record type `Node`. [3]

ii Write program code to create an array `Queue` with 50 records of type `Node`. Your solution should link all nodes and initialise the pointers `FrontOfQueue`, `EndOfQueue` and `StartOfFreeList`. [7]

c The pseudocode algorithm for the queue operation `JoinQueue` is written as a procedure with the header:

```
PROCEDURE JoinQueue(NewItem)
```

where `NewItem` is the new value to be added to the queue. The procedure uses the variables shown in the following identifier table:

Identifier	Data type	Description
`NullPointer`	`INTEGER`	Constant set to –1
		Array to store queue data
	`STRING`	Value to be added
		Pointer to next free node in array
		Pointer to first node in queue
		Pointer to last node in queue
		Pointer to node to be added

i Complete the identifier table. [7]

ii Complete the pseudocode using the identifiers from the table in part (i). [6]

```
PROCEDURE JoinQueue(NewItem : STRING)
   // Report error if no free nodes remaining
   IF StartOfFreeList = ..........................
      THEN
         Report Error
      ELSE
         // new data item placed in node at start of free list
         NewNodePointer ← StartOfFreeList
         Queue[NewNodePointer].Data ← NewItem
         // adjust free list pointer
         StartOfFreeList ← Queue[NewNodePointer].Pointer
         Queue[NewNodePointer].Pointer ← NullPointer
         // if first item in queue then adjust front of queue pointer
         IF FrontOfQueue = NullPointer
            THEN
               ..................... ← ..........................
         ENDIF
         // new node is new end of queue
         Queue[.....................].Pointer ← .....................
         EndOfQueue ← ..........................................
   ENDIF
ENDPROCEDURE
```