



Syllabus Content:

19.1 Abstraction



Show understanding of and use Abstract Data Types (ADT)

Notes and guidance

- Write algorithms to find an item in each of the following:
 - linked list
 - binary tree
- Write algorithms to insert an item into each of the following:
 - Stack
 - Queue
 - linked list
 - binary tree
- Write algorithms to delete an item from each of the following:
 - Stack
 - Queue
 - linked list
- Show understanding that a graph is an example of an ADT.
- Describe the key features of a graph and justify its use for a given situation
Candidates will not be required to write code for this structure



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 builtin types or other ADTs:
 - stack, queue, linked list, dictionary, binary tree



Show understanding that different algorithms which perform the same task can be compared by using criteria (e.g. time taken to complete the task and memory used)

Notes and guidance

- Including use of Big O notation to specify time and space complexity

Abstraction:

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. Abstraction involves filtering out information that is not necessary to solving the problem.

Consider your mobile phone, you just need to know what buttons are to be pressed to send a message or make a call, What happens when you press a button, how your messages are sent, how your calls are connected is all abstracted away from the user.

Abstraction is a powerful methodology to manage complex systems. Abstraction is managed by well-defined objects and their hierarchical classification.

For example a car itself is a well-defined object, which is composed of several other smaller objects like a gearing system, steering mechanism, engine, which are again



have their own subsystems. But for humans car is a one single object, which can be managed by the help of its subsystems, even if their inner details are unknown.

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.






In structured programming, algorithmic **decomposition** breaks a process down into well-defined steps.

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.

ADTs (Abstract Data Type):

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

Abstract Data Types

Definition

An abstract data type is a type with associated operations, but whose representation is hidden.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

It is called “abstract” because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using **integer**, **float**, **char** data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it.

We can think of ADT as a black box which hides the inner structure and design of the data type.

Now we'll define the **ADTs** namely **Stack** ADT, **Queue** ADT, **Linked List** ADT, **Binary Tree** ADT.

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:

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.

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.

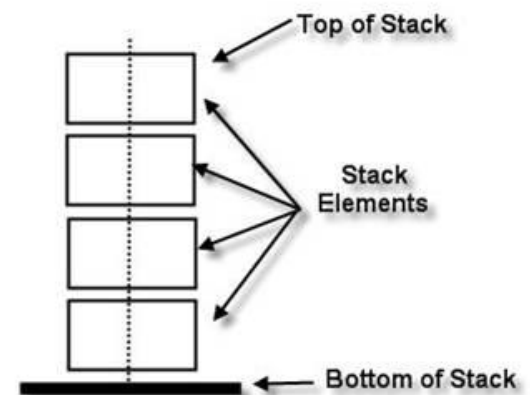
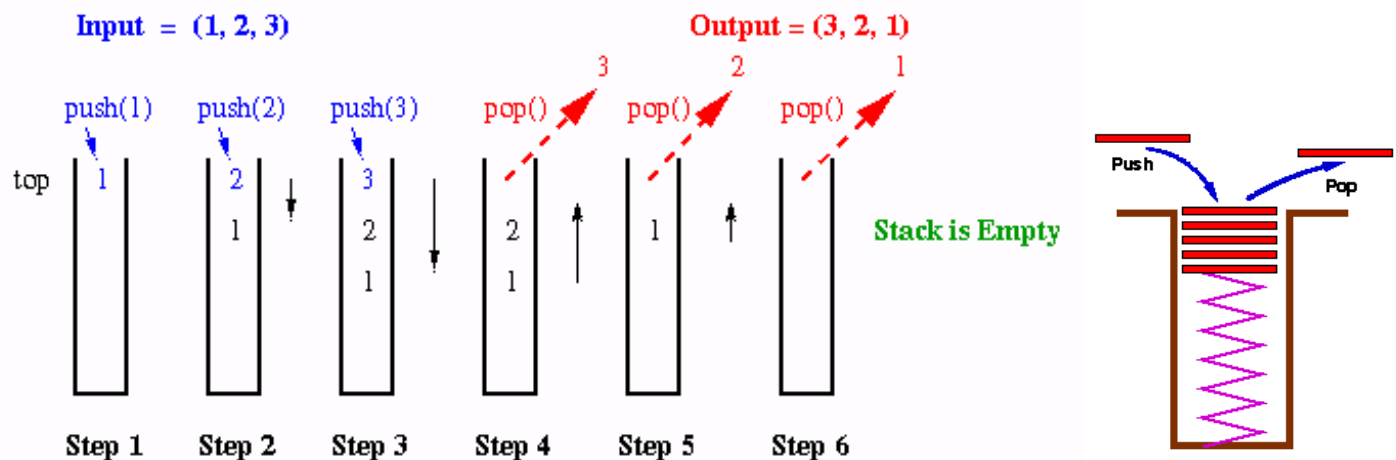


Figure below shows how we can represent a stack when we have added three items in this order: 1, 2, 3 push() adds the item in stack and pop() picks the item from stack.



The '**STACK**' is a **Last-In First-Out (LIFO)** List. Only the last item in the stack can be accessed directly.

push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.



To set up a stack

```
DECLARE stack ARRAY[1:10] OF INTEGER
DECLARE topPointer : INTEGER
DECLARE basePointer : INTEGER
DECLARE stackful : INTEGER
basePointer ← 1
topPointer ← 0
stackful ← 10
```

To push an item, stored in `item`, onto a stack

```
IF topPointer < stackful
  THEN
    topPointer ← topPointer + 1
    stack[topPointer] ← item
  ELSE
    OUTPUT "Stack is full, cannot push"
  ENDIF
```

To pop an item, stored in `item`, from the stack

```
IF topPointer = basePointer - 1
  THEN
    OUTPUT "Stack is empty, cannot pop"
  ELSE
    Item ← stack[topPointer]
    topPointer ← topPointer - 1
  ENDIF
```



Stacks in VB

```
Public Dim stack() As Integer = {Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}
Public Dim basePointer As Integer = 0
Public Dim topPointer As Integer = -1
Public Const stackFull As Integer = 10
Public Dim item As Integer
```

Stack Pop Operation

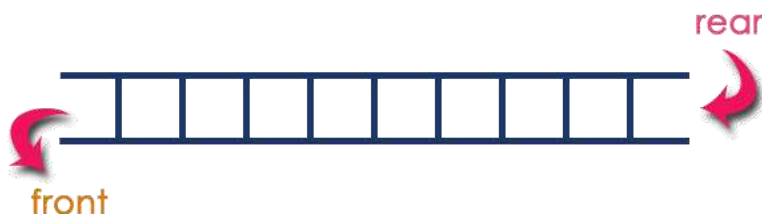
topPointer points to the top of stack

```
Sub pop()
    If topPointer = basePointer - 1 Then
        Console.WriteLine("Stack is empty, cannot pop")
    Else
        item = stack(topPointer)
        topPointer = topPointer - 1
    End If
End Sub
```

Stack Push Operation

```
Sub push(ByVal item)
    If topPointer < stackFull - 1 Then
        topPointer = topPointer + 1
        stack(topPointer) = item
    Else
        Console.WriteLine("Stack is full, cannot push")
    End if
End Sub
```

Queue ADT



Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions.

The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.

In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.

A Queue contains elements of same type arranged in sequential order. Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:

enqueue() – Insert an element at the end of the queue.

dequeue() – Remove and return the first element of queue, if the queue is not empty.

peek() – Return the element of the queue without removing it, if the queue is not empty.

size() – Return the number of elements in the queue.

isEmpty() – Return true if the queue is empty, otherwise return false.

isFull() – Return true if the queue is full, otherwise return false.

Queue after inserting 25, 30, 51, 60 and 85.

After Inserting five elements...



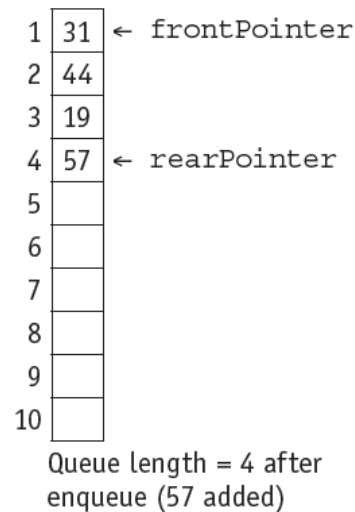
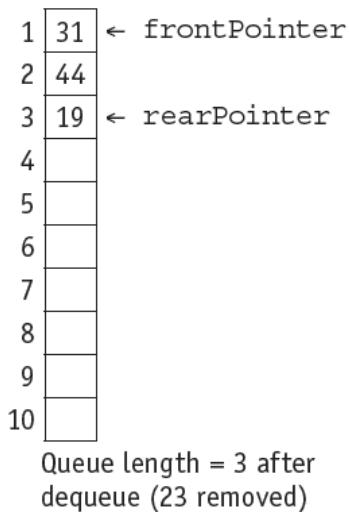
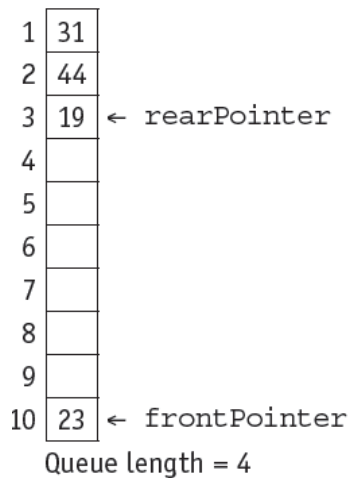
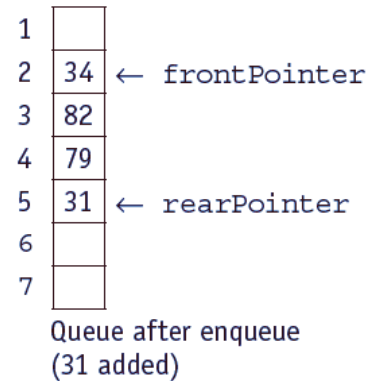
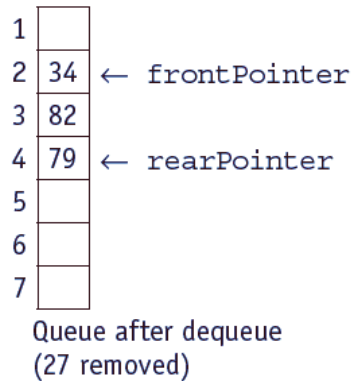
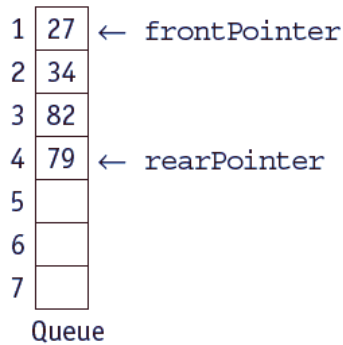
From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.



P4 19.1) Abstraction, Algorithms and ADT's

CS 9618 with Majid Tahir at
www.majidtahir.com

The value of the `frontPointer` changes after dequeue but the value of the `rearPointer` changes after enqueue:



To set up a queue

```
DECLARE queue ARRAY[1:10] OF INTEGER
DECLARE rearPointer : INTEGER
DECLARE frontPointer : INTEGER
DECLARE queueful : INTEGER
DECLARE queueLength : INTEGER
frontPointer ← 1
endPointer ← 0
upperBound ← 10
queueful ← 10
queueLength ← 0
```




To add an item, stored in *item*, onto a queue

```
IF queueLength < queueful
  THEN
    IF rearPointer < upperBound
      THEN
        rearPointer ← rearPointer + 1
      ELSE
        rearPointer ← 1
      ENDIF
    queueLength ← queueLength + 1
    queue[rearPointer] ← item
  ELSE
    OUTPUT "Queue is full, cannot enqueue"
  ENDIF
```

To remove an item from the queue and store in *item*

```
IF queueLength = 0
  THEN
    OUTPUT "Queue is empty, cannot dequeue"
  ELSE
    Item ← queue[frontPointer]
    IF frontPointer = upperBound
      THEN
        frontPointer ← 1
      ELSE
        frontPointer ← frontPointer + 1
      ENDIF
    queueLength ← queueLength - 1
  ENDIF
```




Queue Operations in VB:

Empty Queue with no items and variables, set to public for subroutine access.

```
Public Dim queue() As Integer = {Nothing, Nothing, Nothing, Nothing,  
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}  
Public Dim frontPointer As Integer = 0  
Public Dim rearPointer As Integer = -1  
Public Const queueFull As Integer = 10  
Public Dim queueLength As Integer = 0  
Public Dim item As Integer
```

Queue Enqueue (adding an item to queue)

```
Sub enqueue(ByVal item)  
    If queueLength < queueFull Then  
        If rearPointer < queue.length - 1 Then  
            rearPointer = rearPointer + 1  
        Else  
            rearPointer = 0  
        End If  
        queueLength = queueLength + 1  
        queue(rearPointer) = item  
    Else  
        Console.WriteLine("Queue is full, cannot enqueue")  
    End If  
End Sub
```

Queue Dequeue (adding an item to queue)

```
Sub dequeue()  
    If queueLength = 0 Then  
        Console.WriteLine("Queue is empty, cannot dequeue")  
    Else  
        item = queue(frontPointer)  
        If frontPointer = queue.length - 1 Then  
            frontPointer = 0  
        Else  
            frontPointer = frontPointer + 1  
        End if  
        queueLength = queueLength - 1  
    End If  
End Sub
```

Linked lists

Earlier we used an **array** as a linear list. In an **Array** (Linear list), the list items are stored in consecutive locations. This is not always appropriate.

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 below, 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.

Suppose **StartPointer** points to **B**, **B** points to **D** and **D** points to **L**, **L** Points to **NULL**

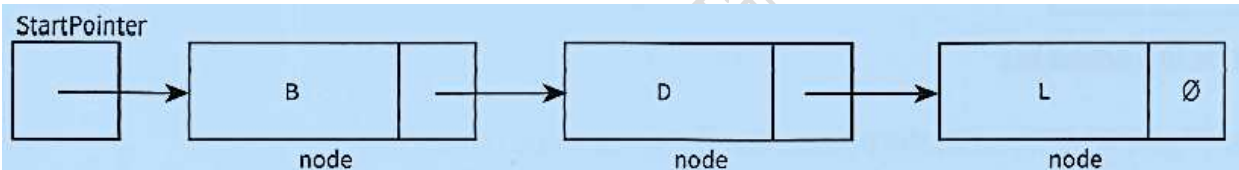


Figure 23.05 Conceptual diagram of a linked list

Add a node at the front: (A 4 steps process)

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**.

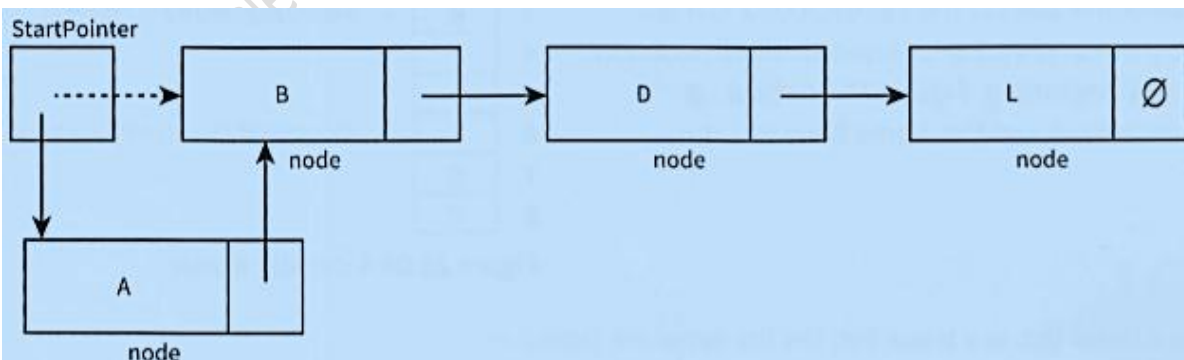


Figure 23.06 Conceptual diagram of adding a new node to the beginning of a linked list

Add a node after a given node:

We are given pointer to a node, and the new node is inserted after the given node.

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**.

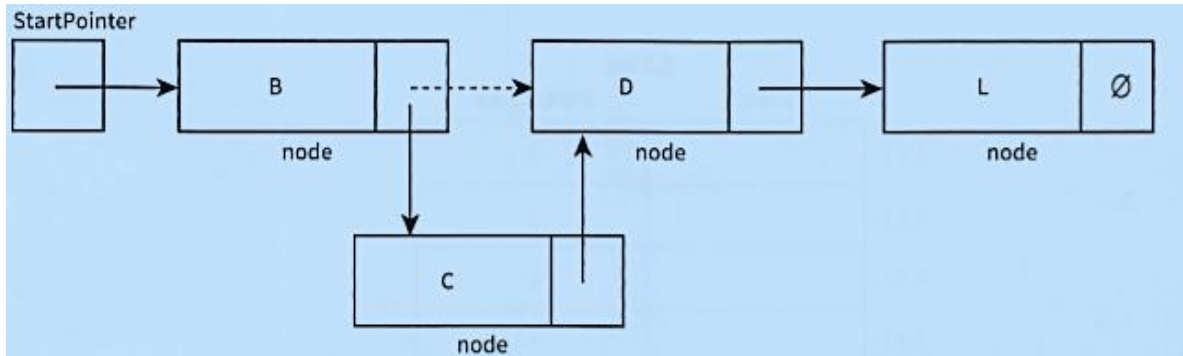


Figure 23.10 Conceptual diagram of adding a new node into a linked list

Add a node at the end:

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.

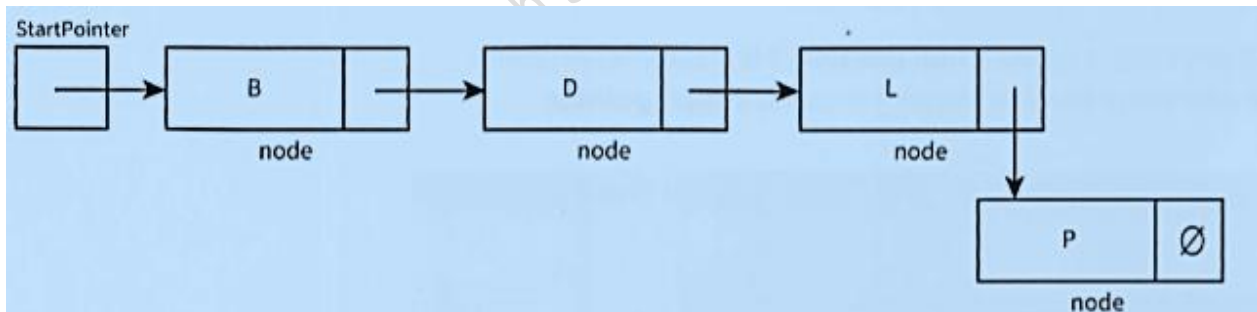
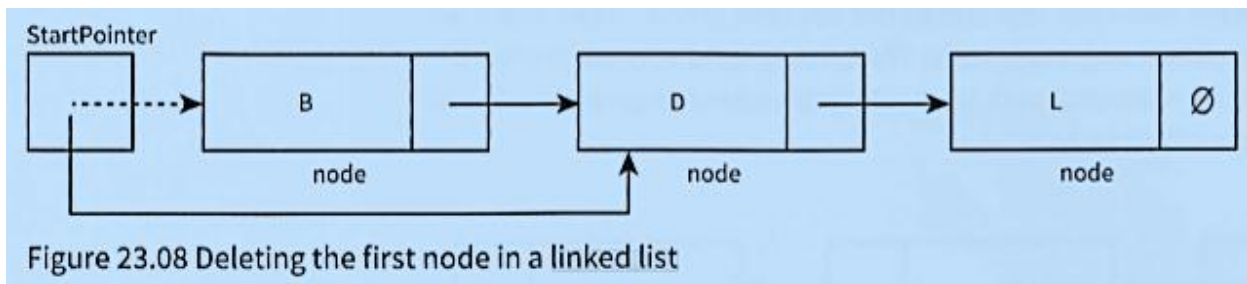


Figure 23.07 Conceptual diagram of adding a new node to the end of a linked list

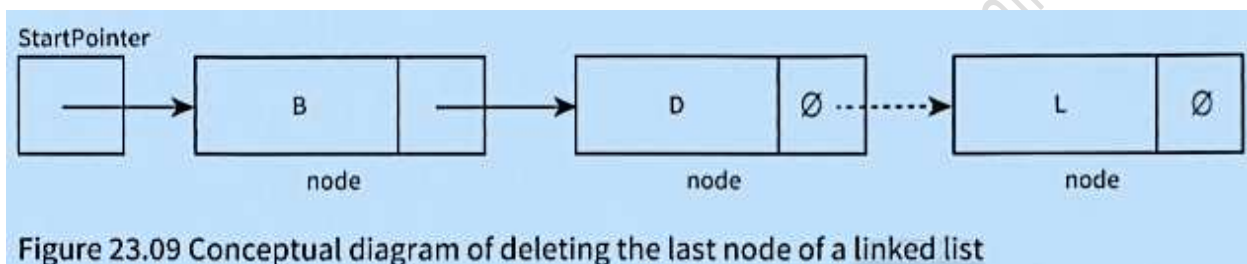
Deleting the First node in the list:

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



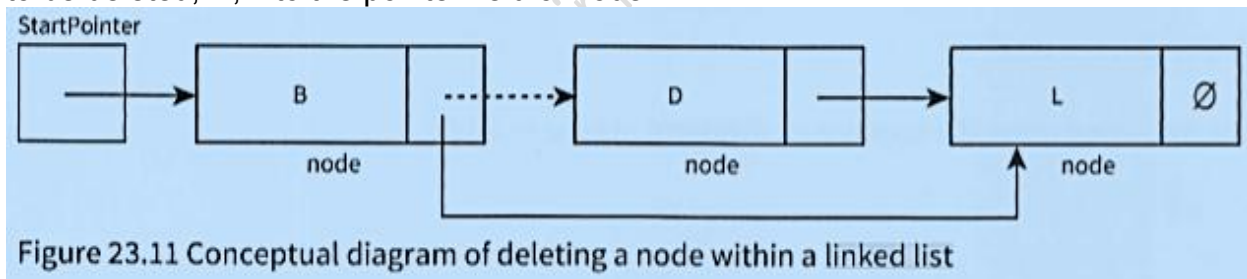
Deleting the Last node in the list:

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



Deleting a node within the 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.



Remember that, in real applications, the data would consist of much more than a **key field** and one **data item**.



When list elements need reordering, only pointers need changing in a linked list. In an **Array (linear list)**, all data items would need to be moved.



This is why linked lists are preferable to **Arrays** (linear lists).



Linked lists saves time, however we need more storage space for the **pointer fields**.

Using Linked Lists:



We can store the linked list in an array of records. One **record** represents a **node** and consists of the **data and a pointer**.



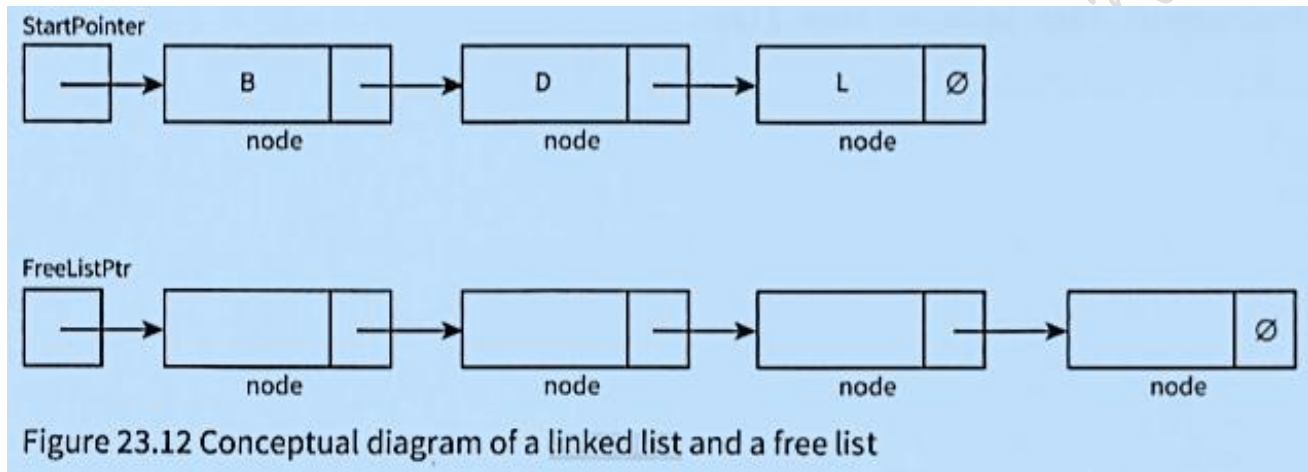
When a node is **inserted** or **deleted**, only the **pointers need to change**. A pointer value is the **array index** of the node pointed to.



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**.



When an array of nodes is first **initialised** to work as a linked list, the **linked list will be empty**.



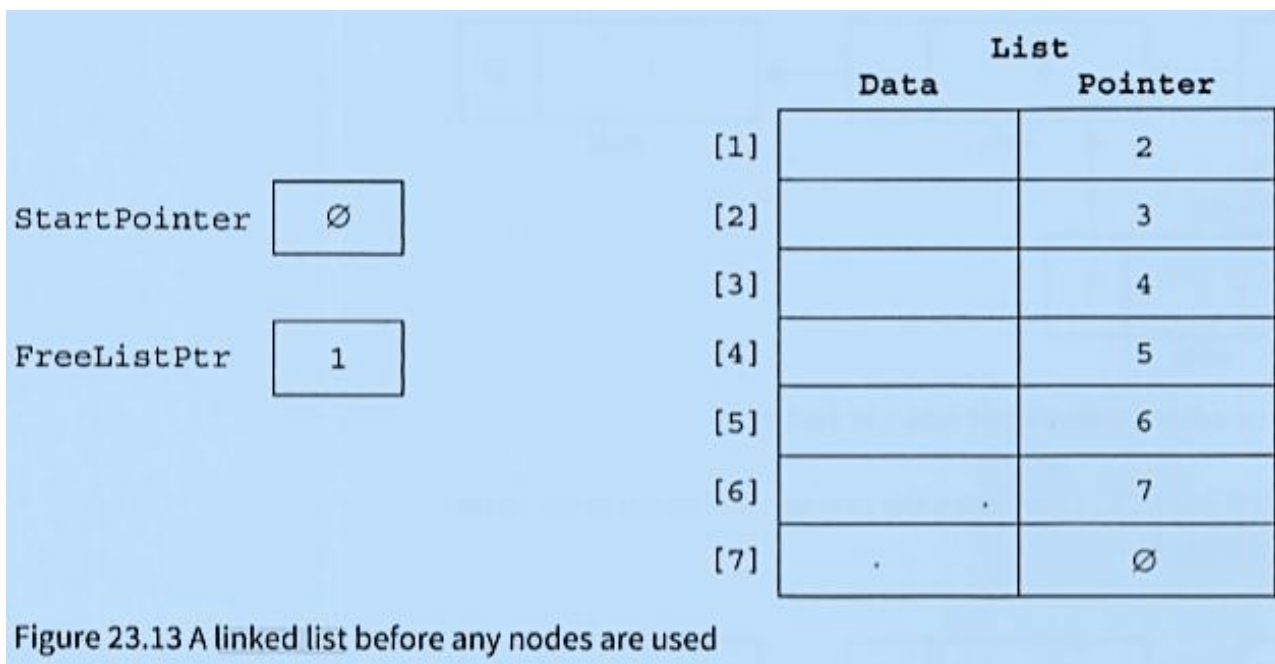
So the **start pointer** will be the **null pointer**.



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.



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

Create a new linked list

```

CONSTANT NullPointer=0 //NullPointer should be set to -1 if using array element with index 0
TYPE ListNode           // Declare record type to store data and pointer
DECLARE Data STRING
DECLARE Pointer INTEGER
ENDTYPE

DECLARE StartPointer : INTEGER // Declare start pointer to point to first item in list
DECLARE FreeListPtr : INTEGER // Declare free pointer to add data in free memory slot.
DECLARE List[1:7] OF ListNode

PROCEDURE InitialiseList
    StartPointer  $\leftarrow$  NullPointer           // set start pointer, start of list
    FreeListPtr  $\leftarrow$  1                     // set starting position of free list
    FOR Index  $\leftarrow$  1 TO 6                 // link all nodes to make free list
        List[Index].Pointer  $\leftarrow$  Index + 1
    NEXT
    List[7].Pointer  $\leftarrow$  Null Pointer       //last node of free list
END PROCEDURE
  
```

Create a new linked list in Visual Studio

Module Module1

```

' NullPointer should be set to -1 if using array element with index 0
Const NULLPTR = -1 ' Declare record type to store data and pointer

Structure ListNode
    Dim Data As String
    Dim Pointer As Integer
End Structure
  
```

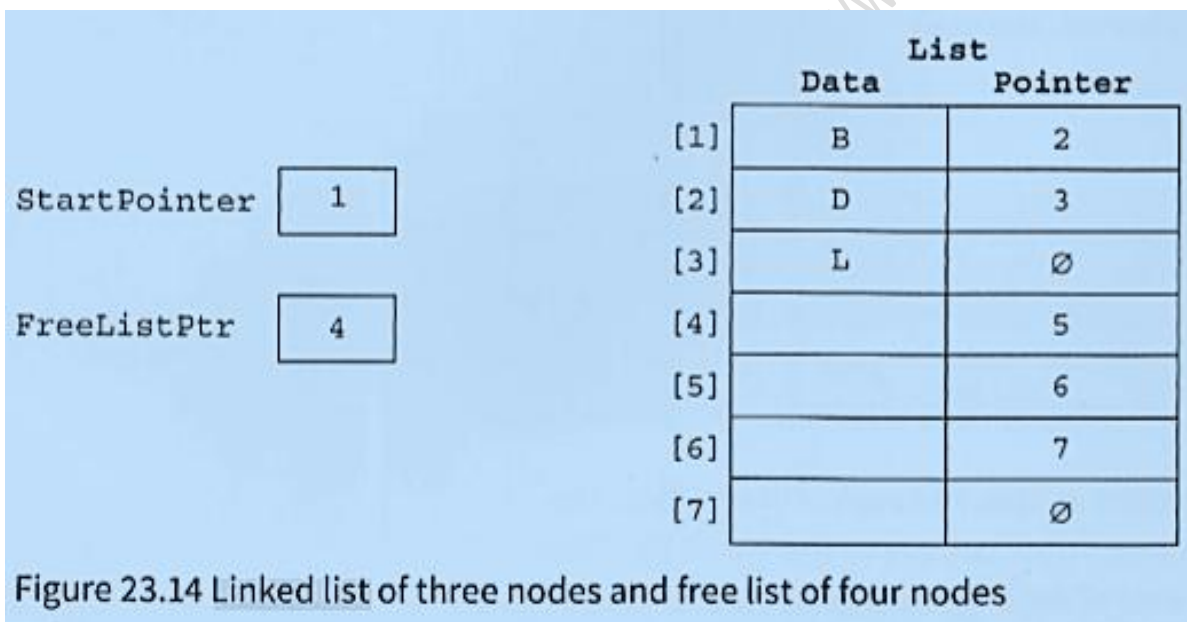
```

Dim List(7) As ListNode
Dim StartPointer As Integer
Dim FreeListPtr As Integer

Sub InitialiseList()
    StartPointer = NULLPTR ' set start pointer
    FreeListPtr = 0        ' set starting position of free list
    For Index = 0 To 7    'link all nodes to make free list
        List(Index).Pointer = Index + 1
    Next
    List(7).Pointer = NULLPTR 'last node of free list
End Sub

```

Insert a new node into an ordered linked list



Here is the identifier table.

Identifier	Description
startPointer	Start of the linked list
heapStartPointer	Start of the heap
nullPointer	Null pointer set to -1
itemAdd	Item to add to the list
tempPointer	Temporary pointer



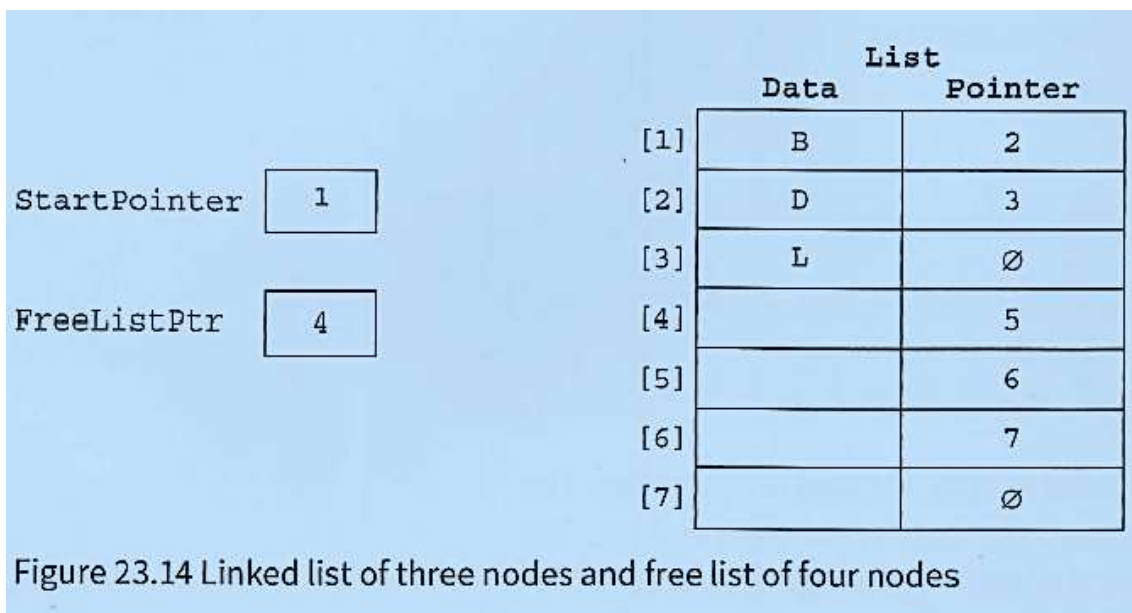
Insert a new node into an ordered linked list

```
DECLARE startpointer : INTEGER
DECLARE heapStartPointer : INTEGER
DECLARE itemAdd : INTEGER
DECLARE tempPointer : INTEGER
CONSTANT nullPointer = -1
PROCEDURE
PROCEDURE InsertNode(Newitem)
  IF FreeListPtr <> NullPointer
  THEN // there is space in the array
    NewNodePtr ← FreeListPtr //take node from free list and store data item
    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
END PROCEDURE
```

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



Insert a new node into an ordered linked list in Visual Studio:

```

Sub InsertNode(ByVal NewItem)

Dim TempPtr, NewNodePtr, PreviousNodePtr As Integer ' Temporarily Pointer, NextNode
Pointer and PreviousPointer to Swap values of pointers
If FreeListPtr <> NULLPTR 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
    PreviousNodePtr = NULLPTR
    TempPtr = StartPointer ' start at beginning of list
    Try
        Do While (TempPtr <> NULLPTR) And (List(TempPtr).Data < NewItem) '
while not end of list
            PreviousNodePtr = TempPtr ' remember this node follow the pointer to
the next node
            TempPtr = List(TempPtr).Pointer
        Loop
    Catch ex As Exception
    End Try
    If PreviousNodePtr = NULLPTR Then ' insert new node at start of list

        List(NewNodePtr).Pointer = StartPointer
        StartPointer = NewNodePtr

    Else : List(NewNodePtr).Pointer = List(PreviousNodePtr).Pointer ' insert new
node between previous node and this node
        List(PreviousNodePtr).Pointer = NewNodePtr
    End If
    Else : Console.WriteLine("no space for more data")
    End If
End Sub

```



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
END FUNCTION
```

Finding an element Visual Studio Code:

```
Function FindNode(ByVal DataItem) As Integer ' returns pointer to node
Dim CurrentNodePtr As Integer
CurrentNodePtr = StartPointer ' start at beginning of list

Try
Do While CurrentNodePtr <> NULLPTR And List(CurrentNodePtr).Data <>
DataItem ' not end of list,item(Not found)
    ' follow the pointer to the next node
CurrentNodePtr = List(CurrentNodePtr).Pointer
Loop
Catch ex As Exception
Console.WriteLine("data not found")
End Try
Return (CurrentNodePtr) ' returns NullPointer if item not found
End Function
```

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
    ENDIF
    List[ThisNodePtr].Pointer ← FreeListPtr
    FreeListPtr ← ThisNodePtr
END PROCEDURE
```



VB Code

```
Sub DeleteNode(ByVal DataItem)

    Dim ThisNodePtr, PreviousNodePtr As Integer
    ThisNodePtr = StartPointer
    Try
        ' start at beginning of list
        Do While ThisNodePtr <> NULLPTR And List(ThisNodePtr).Data <> DataItem
            ' while not end of list and item not found
            PreviousNodePtr = ThisNodePtr ' remember this node
            ' follow the pointer to the next node
            ThisNodePtr = List(ThisNodePtr).Pointer
        Loop
    Catch ex As Exception
        Console.WriteLine("data does not exist in list")
    End Try

    If ThisNodePtr <> NULLPTR Then ' node exists in list
        If ThisNodePtr = StartPointer Then ' first node to be deleted
            StartPointer = List(StartPointer).Pointer
        Else : List(PreviousNodePtr).Pointer = List(ThisNodePtr).Pointer
        End If
        List(ThisNodePtr).Pointer = FreeListPtr
        FreeListPtr = ThisNodePtr
    End If
End Sub
```

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
```

VB Code

```
Sub OutputAllNodes()
    Dim CurrentNodePtr As Integer
    CurrentNodePtr = StartPointer ' start at beginning of list
    If StartPointer = NULLPTR Then
        Console.WriteLine("No data in list")
    End If
    Do While CurrentNodePtr <> NULLPTR ' while not end of list

        Console.WriteLine(CurrentNodePtr & " " & List(CurrentNodePtr).Data)
        ' follow the pointer to the next node
        CurrentNodePtr = List(CurrentNodePtr).Pointer
    Loop
End Sub
```



VB Program for Linked Lists

```
Module Module1
    ' NullPointer should be set to -1 if using array element with index 0
    Const NULLPTR = -1 ' Declare record type to store data and pointer
    Structure ListNode
        Dim Data As String
        Dim Pointer As Integer
    End Structure

    Dim List(7) As ListNode
    Dim StartPtr As Integer
    Dim FreeListPtr As Integer

    Sub Initialiselist()
        StartPtr = NULLPTR ' set start pointer
        FreeListPtr = 0 ' set starting position of free list
        For Index = 0 To 7 'link all nodes to make free list
            List(Index).Pointer = Index + 1
        Next
        List(7).Pointer = NULLPTR 'last node of free list
    End Sub

    Function FindNode(ByVal DataItem) As Integer ' returns pointer to node
        Dim CurrentNodePtr As Integer
        CurrentNodePtr = StartPtr ' start at beginning of list
        Try
            Do While CurrentNodePtr <> NULLPTR And List(CurrentNodePtr).Data <>
DataItem ' not end of list,item(Not found)
                ' follow the pointer to the next node
                CurrentNodePtr = List(CurrentNodePtr).Pointer
            Loop
        Catch ex As Exception
            Console.WriteLine("data not found")
        End Try
        Return (CurrentNodePtr) ' returns NullPointer if item not found
    End Function

    Sub DeleteNode(ByVal DataItem)
        Dim ThisNodePtr, PreviousNodePtr As Integer
        ThisNodePtr = StartPtr
        Try
            ' start at beginning of list
            Do While ThisNodePtr <> NULLPTR And List(ThisNodePtr).Data <> DataItem
' while not end of list and item not found
                PreviousNodePtr = ThisNodePtr ' remember this node
                ' follow the pointer to the next node
                ThisNodePtr = List(ThisNodePtr).Pointer
            Loop
        Catch ex As Exception
            Console.WriteLine("data does not exist in list")
        End Try
        If ThisNodePtr <> NULLPTR Then ' node exists in list
            If ThisNodePtr = StartPtr Then ' first node to be deleted
```



```
        StartPointer = List(StartPointer).Pointer
    Else : List(PreviousNodePtr).Pointer = List(ThisNodePtr).Pointer
    End If
    List(ThisNodePtr).Pointer = FreeListPtr
    FreeListPtr = ThisNodePtr
End If
End Sub

Sub InsertNode(ByVal NewItem)

    Dim ThisNodePtr, NewNodePtr, PreviousNodePtr As Integer
    If FreeListPtr <> NULLPTR 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
        PreviousNodePtr = NULLPTR
        ThisNodePtr = StartPointer ' start at beginning of list
        Try
            Do While (ThisNodePtr <> NULLPTR) And (List(ThisNodePtr).Data <
NewItem)
                ' while not end of list
                PreviousNodePtr = ThisNodePtr ' remember this node
                ' follow the pointer to the next node
                ThisNodePtr = List(ThisNodePtr).Pointer
            Loop
        Catch ex As Exception
        End Try
        If PreviousNodePtr = NULLPTR Then ' insert new node at start of list

            List(NewNodePtr).Pointer = StartPointer
            StartPointer = NewNodePtr

        Else : List(NewNodePtr).Pointer = List(PreviousNodePtr).Pointer
            ' insert new node between previous node and this node
            List(PreviousNodePtr).Pointer = NewNodePtr
        End If
    Else : Console.WriteLine("no space for more data")
    End If
End Sub

Sub OutputAllNodes()
    Dim CurrentNodePtr As Integer
    CurrentNodePtr = StartPointer ' start at beginning of list
    If StartPointer = NULLPTR Then
        Console.WriteLine("No data in list")
    End If
    Do While CurrentNodePtr <> NULLPTR ' while not end of list

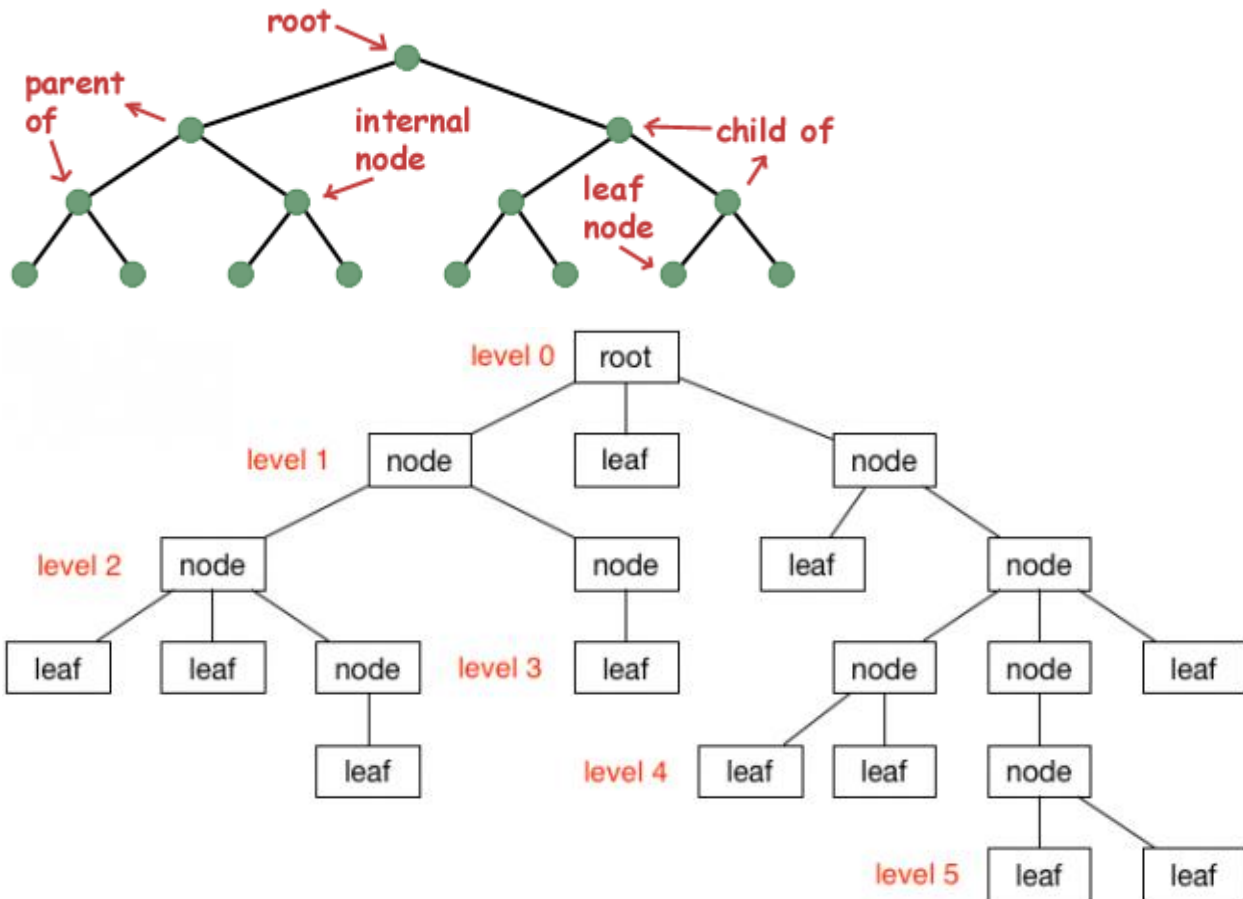
        Console.WriteLine(CurrentNodePtr & " " & List(CurrentNodePtr).Data)
        ' follow the pointer to the next node
        CurrentNodePtr = List(CurrentNodePtr).Pointer
    Loop
End Sub
```



```
Function GetOption()  
    Dim Choice As Char  
    Console.WriteLine("1: insert a value")  
    Console.WriteLine("2: delete a value")  
    Console.WriteLine("3: find a value")  
    Console.WriteLine("4: output list")  
    Console.WriteLine("5: end program")  
    Console.Write("Enter your choice: ")  
    Choice = Console.ReadLine()  
    Return (Choice)  
End Function  
  
Sub Main()  
    Dim Choice As Char  
    Dim Data As String  
    Dim CurrentNodePtr As Integer  
  
    Initialiselist()  
    Choice = GetOption()  
    Do While Choice <> "5"  
        Select Case Choice  
            Case "1"  
                Console.Write("Enter the value: ")  
                Data = Console.ReadLine()  
                InsertNode(Data)  
                OutputAllNodes()  
            Case "2"  
                Console.Write("Enter the value: ")  
                Data = Console.ReadLine()  
                DeleteNode(Data)  
                OutputAllNodes()  
            Case "3"  
                Console.Write("Enter the value: ")  
                Data = Console.ReadLine()  
                CurrentNodePtr = FindNode(Data)  
            Case "4"  
                OutputAllNodes()  
                Console.WriteLine(StartPointer & " " & FreeListPtr)  
                For i = 0 To 7  
                    Console.WriteLine(i & " " & List(i).Data & " " &  
List(i).Pointer)  
                Next  
            End Select  
            Choice = GetOption()  
        Loop  
    End Sub  
End Module
```


Trees Data Structure:

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'.



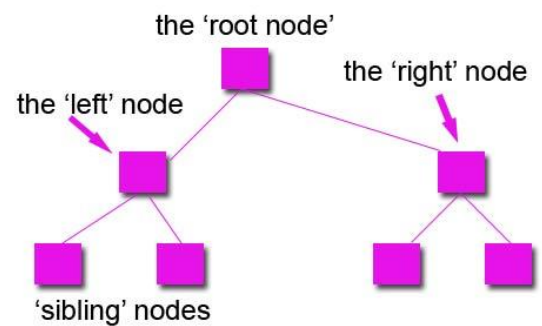
In computer science binary trees are used for different purposes. In this chapter, you will use an ordered binary tree ADT as a binary search tree.

Tree Vocabulary:

The **TREE** is a general data structure that describes the relationship between data items or 'nodes'.

The parent of a binary tree has only two child nodes.




- Each data item within a **tree** is called a **node**
- The highest data item in **tree** is called **root** or **root node**
- Below the **root** lie a number of **other nodes**. The **root** is the **parent** of **nodes** immediately linked to it and these are **children** of **parent node**.
- If **node** share **common parent**, they are **sibling nodes** just like a family

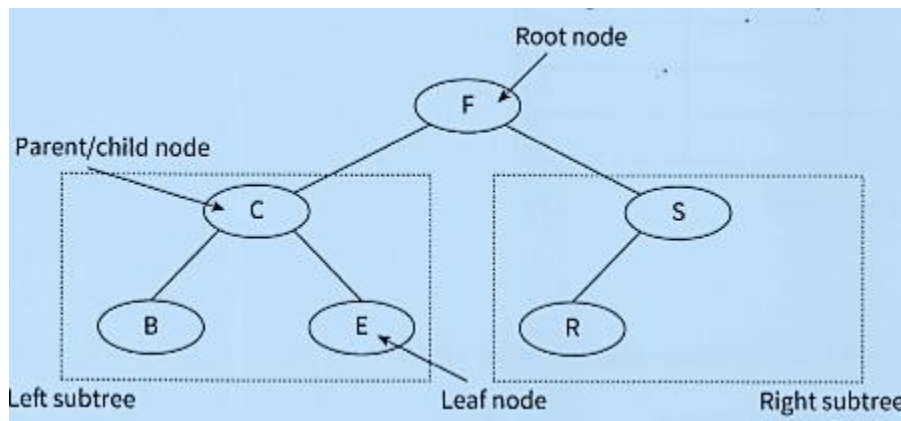


A BINARY TREE

Adding Nodes to a 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 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.

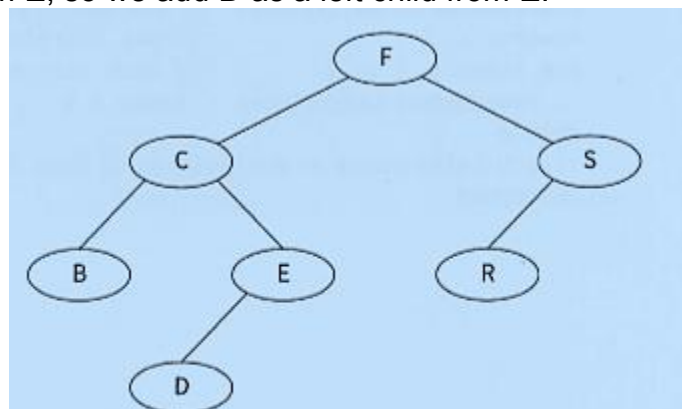


Figure 23.16 Conceptual diagram of adding a node to an ordered binary tree



Create a new binary tree

```
CONSTANT NullPointer = 0 //NullPointer should be set to -1 if u sing a r ray element with
index 0
//Declare record type to store data and pointers
TYPE TreeNode
    DECLARE Data : STRING
    DECLARE LeftPointer : INTEGER
    DECLARE RightPointer : INTEGER
END TYPE

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
    END FOR
    Tree[7].LeftPointer ← NullPointer //last node of free list
END PROCEDURE
```

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].Left Pointer ← NewNodePtr
            ELSE
                Tree[PreviousNodePtr].Right Pointer ← NewNodePtr
            ENDIF
        ENDIF
    ENDIF
END PROCEDURE
```



```
ELSE
    Tree[PreviousNodePtr].RightPointer ← NewNodePtr
ENDIF
ENDIF
ENDIF
END PROCEDURE
```

Finding 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
END FUNCTION
```

Implementing a binary tree in VB

```
Module Module1
    ' NullPointer should be set to -1 if using array element with index 0
    Const NULLPOINTER = -1
    ' Declare record type to store data and pointer
    Structure TreeNode
        Dim Data As String
        Dim LeftPointer, RightPointer As Integer
    End Structure

    Dim Tree(7) As TreeNode
    Dim RootPointer As Integer
    Dim FreePtr As Integer

    Sub InitialiseTree()
        RootPointer = NULLPOINTER ' set start pointer
        FreePtr = 0 ' set starting position of free list
        For Index = 0 To 7 'link all nodes to make free list
            Tree(Index).LeftPointer = Index + 1
            Tree(Index).RightPointer = NULLPOINTER
            Tree(Index).Data = ""
        Next
        Tree(7).LeftPointer = NULLPOINTER 'last node of free list
    End Sub

    Function FindNode(ByVal SearchItem) As Integer
```



```
Dim ThisNodePtr As Integer
ThisNodePtr = RootPointer
Try

    Do While ThisNodePtr <> NULLPTR And Tree(ThisNodePtr).Data <>
SearchItem

        If Tree(ThisNodePtr).Data > SearchItem Then

            ThisNodePtr = Tree(ThisNodePtr).LeftPointer
        Else : ThisNodePtr = Tree(ThisNodePtr).RightPointer
        End If
    Loop
Catch ex As Exception
End Try
Return ThisNodePtr
End Function

Sub InsertNode(ByVal NewItem)
Dim NewNodePtr, ThisNodePtr, PreviousNodePtr As Integer
Dim TurnedLeft As Boolean
If FreePtr <> NULLPTR Then ' there is space in the array
    ' take node from free list and store data item
    NewNodePtr = FreePtr
    Tree(NewNodePtr).Data = NewItem
    FreePtr = Tree(FreePtr).LeftPointer
    Tree(NewNodePtr).LeftPointer = NULLPTR ' check if empty
tree

    If RootPointer = NULLPTR Then
        RootPointer = NewNodePtr
    Else ' find insertion point
        ThisNodePtr = RootPointer
        Do While ThisNodePtr <> NULLPTR

            PreviousNodePtr = ThisNodePtr
            If Tree(ThisNodePtr).Data > NewItem Then
                TurnedLeft = True
                ThisNodePtr = Tree(ThisNodePtr).LeftPointer
            Else
                TurnedLeft = False
                ThisNodePtr = Tree(ThisNodePtr).RightPointer
            End If
        Loop
        If TurnedLeft Then
            Tree(PreviousNodePtr).LeftPointer = NewNodePtr
        Else : Tree(PreviousNodePtr).RightPointer = NewNodePtr
        End If
    End If
Else
    Console.WriteLine("no spce for more data")
End If
End Sub

Sub TraverseTree(ByVal RootPointer)
```



```
If RootPointer <> NULLPTR Then
    TraverseTree(Tree(RootPointer).LeftPointer)
    Console.WriteLine(Tree(RootPointer).Data)
    TraverseTree(Tree(RootPointer).RightPointer)
End If
End Sub

Function GetOption()
    Dim Choice As Char
    Console.WriteLine("1: add data")
    Console.WriteLine("2: find data")
    Console.WriteLine("3: traverse tree")
    Console.WriteLine("4: end program")
    Console.Write("Enter your choice: ")
    Choice = Console.ReadLine()
    Return (Choice)
End Function

Sub Main()
    Dim Choice As Char
    Dim Data As String
    Dim ThisNodePtr As Integer
    InitialiseTree()
    Choice = GetOption()
    Do While Choice <> "4"
        Select Case Choice
            Case "1"
                Console.Write("Enter the value: ")
                Data = Console.ReadLine()
                InsertNode(Data)
                TraverseTree(RootPointer)
            Case "2"
                Console.Write("Enter search value: ")
                Data = Console.ReadLine()
                ThisNodePtr = FindNode(Data)
                If ThisNodePtr = NULLPTR Then
                    Console.WriteLine("Value not found")
                Else
                    Console.WriteLine("value found at: " & ThisNodePtr)
                End If
                Console.WriteLine(RootPointer & " " & FreePtr)
                For i = 0 To 7
                    Console.WriteLine(i & " " & Tree(i).LeftPointer & " " &
Tree(i).Data & " " & Tree(i).RightPointer)
                Next
            Case "3"
                TraverseTree(RootPointer)
            End Select
            Choice = GetOption()
        Loop
    End Sub
End Module
```


Graphs:



A **graph** is a **non-linear** data structure consisting of **nodes** and **edges**.



This is an ADT used to implement **directed** and **undirected** graphs.



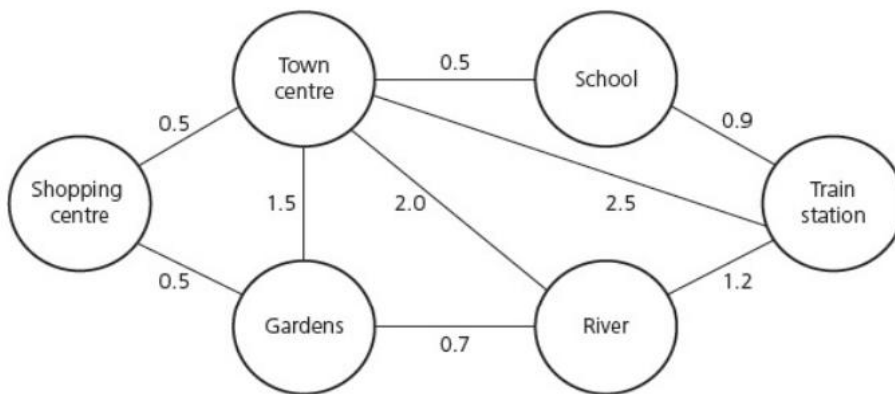
A graph consists of a **set of nodes** and **edges that join a pair of nodes**.



If the edges have a direction from one node to the other **it is a directed graph**.

For example, a graph of the bus routes in a town could be as follows. The distance between each bus stop in kilometres is shown on the graph.

We have already covered graphs in Chapter 18 Artificial Intelligence



Data Dictionary:



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 (see Section 23.11), so that a value can be looked up using a direct-access method.

Here are some examples of VB dictionaries:

```

Dim EnglishFrench As New Dictionary(Of String, String)
EnglishFrench.Add("book", "livre")
EnglishFrench.Add("pen", "stylo")
Console.WriteLine(EnglishFrench.Item("book"))

Dim ComputingTerms As New Dictionary(Of String, String)
ComputingTerms.Add("Boolean", "can be TRUE or FALSE")
ComputingTerms.Add("Bit", "0 or 1")
Console.WriteLine(ComputingTerms.Item("Bit"))
Console.ReadLine()
  
```


Big O notation:

A problem can be solved in different ways, with different algorithms.

Clearly, we want to use time and memory efficiently.



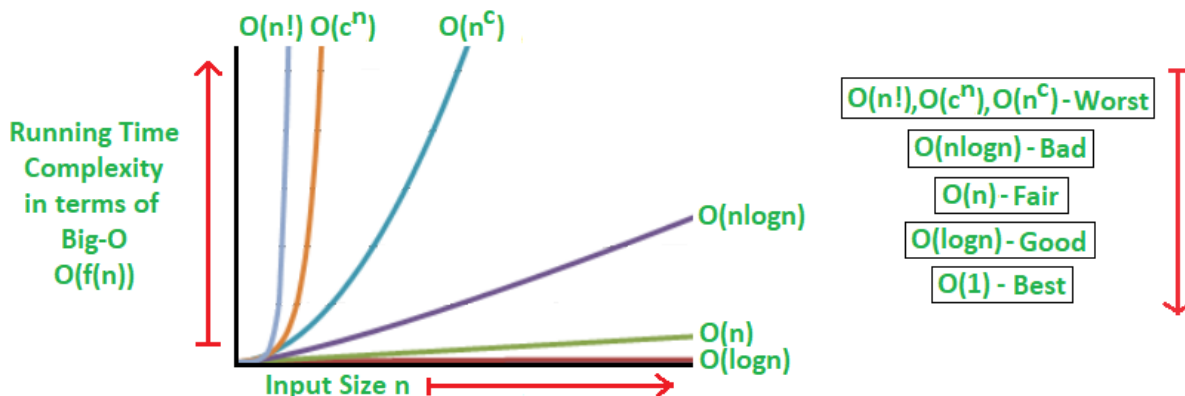
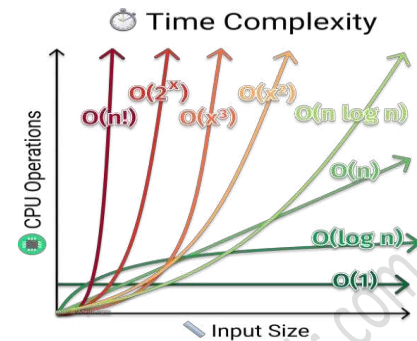
A way of comparing the efficiency of algorithms has been devised using **order of growth** as a function of the size of the input. \



Big O notation is used to **classify algorithms according to how their running time (or space requirements) grows as the input size grows.**



The **letter O** is used because the growth rate of a function is also referred to as 'order of the function'. The worst-case scenario is used when calculating the order of growth for very large data sets.



Consider the **linear search algorithm**.



The worst case scenario is that the item searched for is **the last item** in the list.



The longer the list, the **more comparisons have to be made**.



If the list is twice as long, twice as many comparisons have to be made.

Generally, we can say the order of growth is linear.

We write this as **O(n)**, where **n** is the size of the data set.

Computer Science AS & A Level Coursebook by Sylvia Langfield & Dave Duddell

<https://www.geeksforgeeks.org/abstract-data-types/>

<https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/>

http://btechsmartclass.com/DS/U2_T7.html

http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_5/data_structures/miniweb/pg15.htm

<https://www.geeksforgeeks.org/binary-tree-set-1-introduction/>

<https://www.thecrazyprogrammer.com/2017/08/difference-between-tree-and-graph.html>

<https://www.codeproject.com/Articles/4647/A-simple-binary-tree-implementation-with-VB-NET>

<https://www.dotnetperls.com/dictionary-vbnet> To see complete Dictionary Codes

https://www.tutorialspoint.com/vb.net/vb.net_hashtable.htm

<https://www.tutlane.com/tutorial/visual-basic/vb-hashtable>