Syllabus Content:

4.1.1 Abstraction

- show understanding of how to model a complex system by only including essential details, using:
  functions and procedures with suitable parameters (as in procedural programming, see Section 2.3)
- ADTs (see Section 4.1.3)
- classes (as used in object-oriented programming, see Section 4.3.1)
- facts, rules (as in declarative programming, see Section 4.3.1)

4.1.2 Algorithms

- 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 an algorithm to implement an insertion sort
- write an algorithm to implement a bubble sort
- show understanding that performance of a sort routine may depend on the initial order of the data and the number of data items
- write algorithms to find an item in each of the following: linked list, binary tree, hash table
- write algorithms to insert an item into each of the following: stack, queue, linked list, binary tree, hash table
- write algorithms to delete an item from each of the following: stack, queue, linked list
- 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

4.1.3 Abstract Data Types (ADT)

- show understanding that an 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

```
TYPE <identifier1>
DECLARE <identifier2> : <data type>
DECLARE <identifier3> : <data type>
        …
ENDTYPE
```

- 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

Contact: 03004003666
Email: majidtahir61@gmail.com

Computational thinking and problem-solving:

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 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.

Data modeling:

Data modeling involves analysing and organising data. 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 (ID) arrays to represent linear lists and two-dimensional (2D) arrays to represent tables or matrices.

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. There are more ways to build data models.

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.

Algorithm design

Algorithm design involves developing step-by-step instructions to solve a problem

Use subroutines to modularize the solution to a problem

Subroutine/sub-program

A subroutine is a self-contained section of program code which performs a specific task and is referenced by a name.

A subroutine resembles a standard program in that it will contain its own local variables, data types, labels and constant declarations.

There are two types of subroutine. These are procedures and functions.

Functions and Procedures

Procedure

A procedure is a subroutine that performs a specific task without returning a value to the part of the program from which it was called.

Function

A function is a subroutine that performs a specific task and returns a value to the part of the program from which it was called.
Note that a function is 'called' by writing it on the right hand side of an assignment statement.

Parameter

A parameter is a value that is 'received' in a subroutine (procedure or function).
The subroutine uses the value of the parameter within its execution.

The action of the subroutine will be different depending upon the parameters that it is passed.
Parameters are placed in parenthesis after the subroutine name.
For example: Square(5) 'passes the parameter 5 – returns 25

By Ref vs. By Val

Parameters can be passed by reference (byref) or by value (byval).

If you want to pass the value of the variable, use the ByVal syntax. By passing the value of the variable instead of a reference to the variable, any changes to the variable made by code in the

subroutine or function will not be passed back to the main code. This is the default passing mechanism when you don't decorate the parameters by using ByVal or ByRef.
If you want to change the value of the variable in the subroutine or function and pass the revised value back to the main code, use the ByRef syntax. This passes the reference to the variable and allows its value to be changed and passed back to the main code.

Example Program in VB – Procedures & Functions

```
Module Module1
'this is a procedure
Sub timestable(ByRef number As Integer)
      For x = 1 To 10
        Console.WriteLine(number & " x " & x & " = " & (number * x))
     Next
End Sub

'this is a function (functions return a value)
Function adder(ByRef a As Integer, ByVal b As Integer)
       adder = a + b
        Return adder
End Function

Sub Main()
timestable(7) 'this is a call (executes a procedure or function)
timestable(3)'this is a second call to the same procedure but now with different data
timestable(9)
Console.ReadKey()
Console.Clear()

Dim x As Integer
      x = adder(2, 3) 'call to function adder which returns a value
      Console.WriteLine("2 + 3 = " & x)
      Console.WriteLine("4 + 6 = " & adder(4, 6)) 'you can simply then code by
putting the call directly into the print statement
       Console.ReadKey()
End Sub
End Module
```

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 three **ADTs** namely **List** ADT, **Stack** ADT, **Queue** ADT.

List ADT

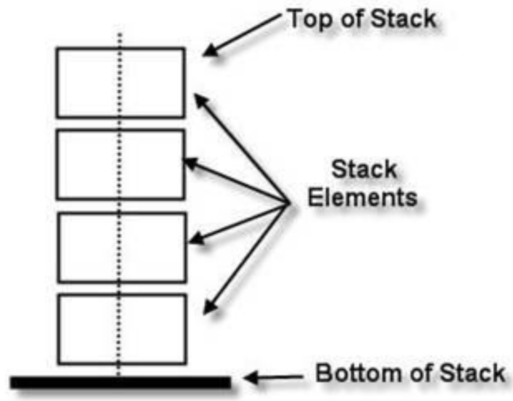
A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

get() – Return an element from the list at any given position.
insert() – Insert an element at any position of the list.
remove() – Remove the first occurrence of any element from a non-empty list.
removeAt() – Remove the element at a specified location from a non-empty list.
replace() – Replace an element at any position by another element.
size() – Return the number of elements in the list.
isEmpty() – Return true if the list is empty, otherwise return false.
isFull() – Return true if the list is full, otherwise return false.

Contact: 03004003666
Email: majidtahir61@gmail.com

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:

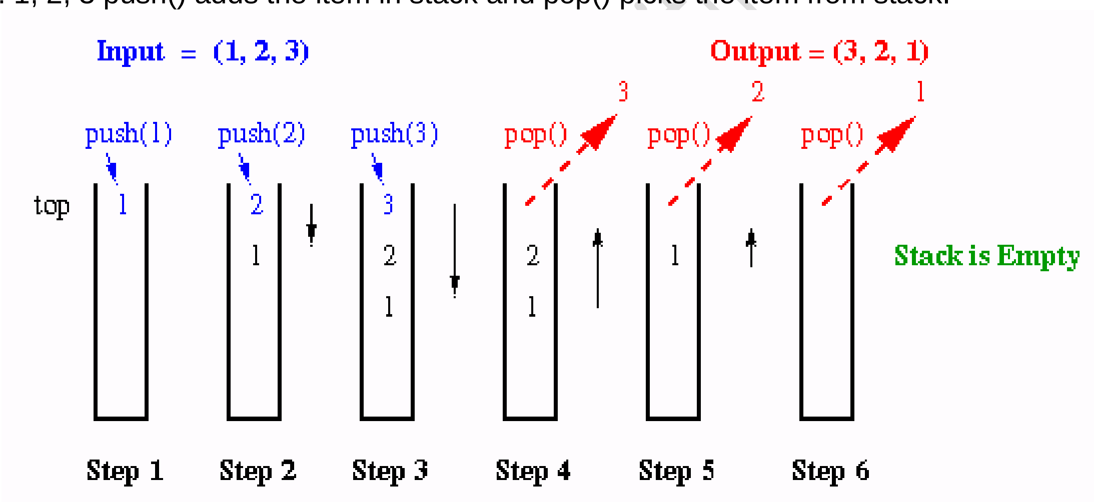


To make a stack, we pile items on top of each other. The item that is accessible is the one on top of the stack. If we try to find an item in the stack and take it out, we are likely to cause the pile of items to collapse.

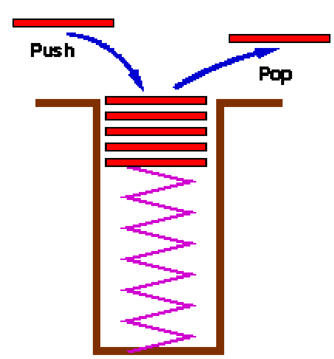
The **BaseofstackPointer** will always point to the first slot in the stack. The **TopOfStackPointer** will point to the last element pushed onto the stack.

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.

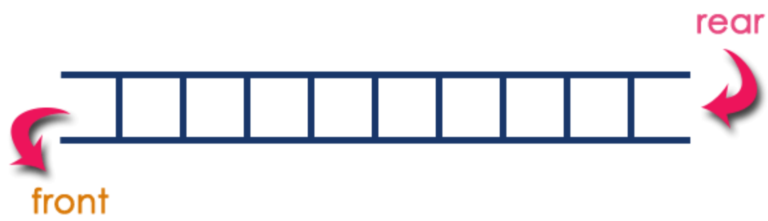


The **'STACK'** is a **Last-In First-Out (LIFO)** List. Only the last item in the stack can be accessed directly.



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.

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.

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.

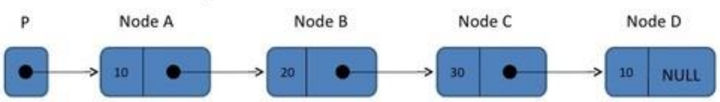
Linked lists

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.

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.

Linked List

- A list implemented by each item having a link to the next item.
- Head points to the first node.
- Last node points to NULL.



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 rep resented by ϕ. 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 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**

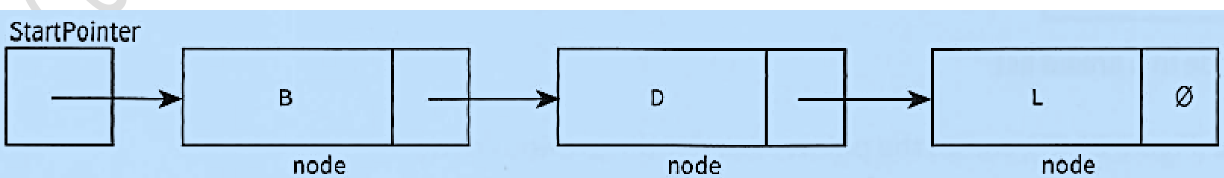


Figure 23.05 Conceptual diagram of a linked list

Contact: 03004003666
Email: majidtahir61@gmail.com

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**.

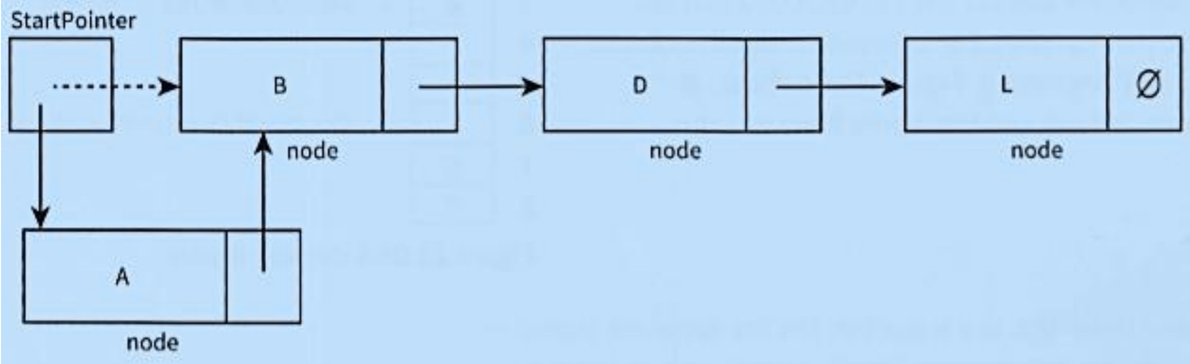


Figure 23.06 Conceptual diagram of adding a new node to the beginning of a linked list

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, Band 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**.

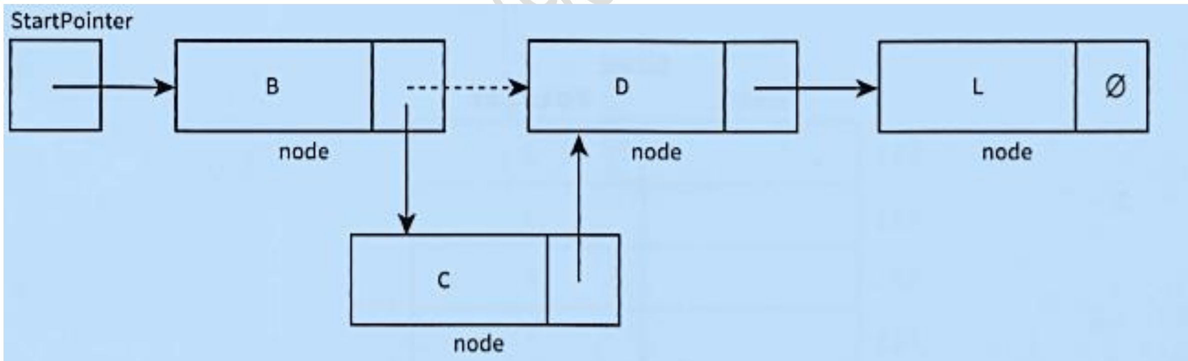


Figure 23.10 Conceptual diagram of adding a new node into a linked list

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.

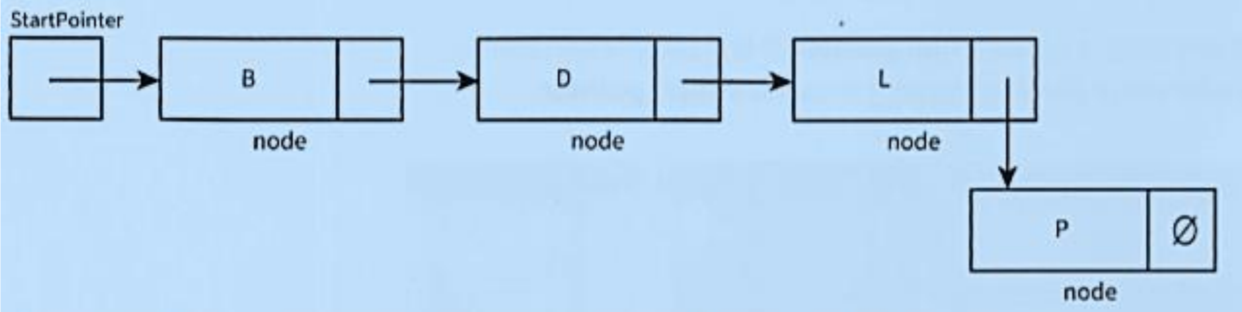


Figure 23.07 Conceptual diagram of adding a new node to the end of a linked list

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**

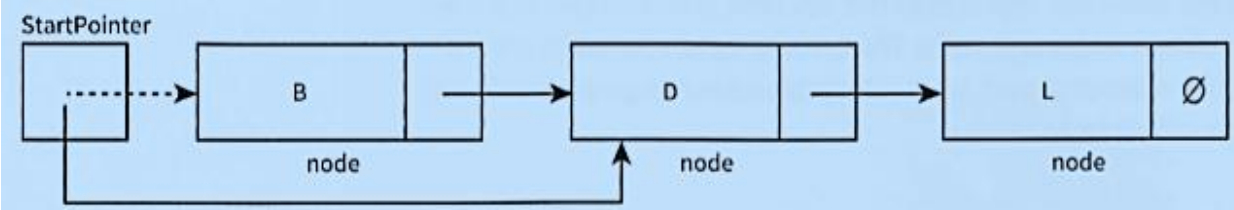


Figure 23.08 Deleting the first node in a linked list

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.

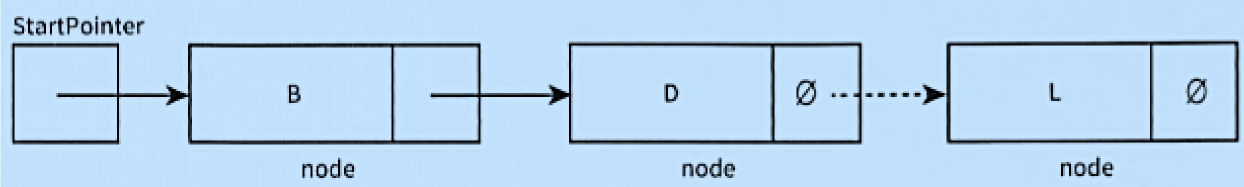


Figure 23.09 Conceptual diagram of deleting the last node of a linked list

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.

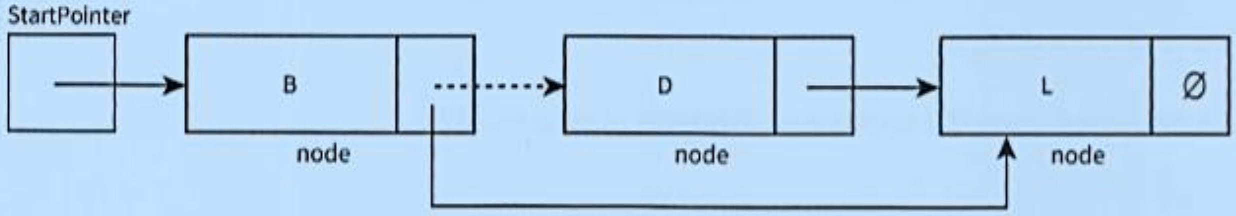


Figure 23.11 Conceptual diagram of deleting a node within a linked list

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

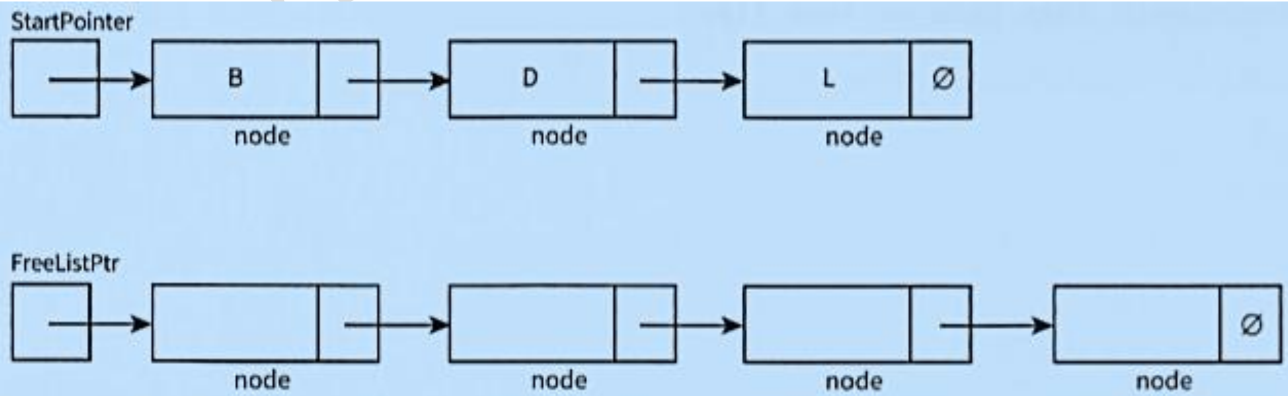


Figure 23.12 Conceptual diagram of a linked list and a free list

- 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.

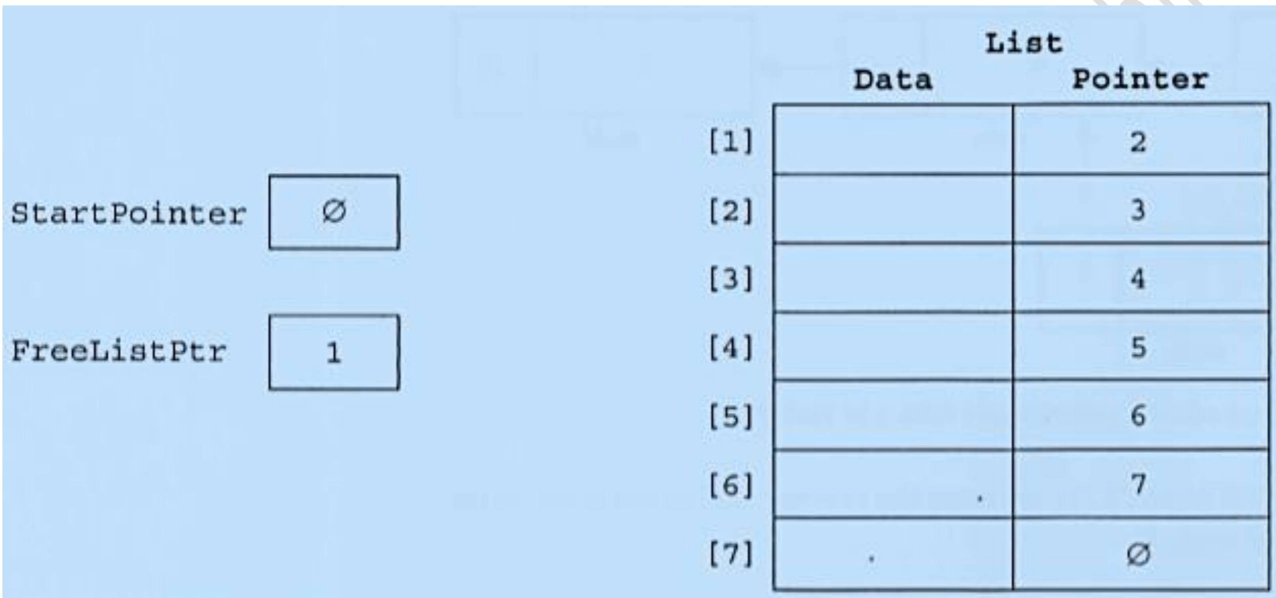


Figure 23.13 A linked list before any nodes are used

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

Create a new linked list

```
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 ← NullPointer          // set start pointer, start of list
       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
       NEXT
       List[7].Pointer ←  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 NULLPOINTER = -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 = NULLPOINTER    ' 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 = NULLPOINTER      'last node of free list
    End Sub
```

Insert a new node into an ordered linked list

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

Insert a new node into an ordered linked list

```
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.

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

Contact: 03004003666
Email: majidtahir61@gmail.com

VB

```
Sub insert (ByVal itemAdd)
    Dim tempPointer As Integer
    If heapStartPointer = nullPointer Then
       Console.WriteLine("Linked List full")
    Else
       tempPointer = startPointer          Adjusting the pointers and adding the item
       startPointer = heapStartPointer
       heapStartPointer = myLinkedListPointers(heapStartPointer)
       myLinkedList(startPointer) = itemAdd
       myLinkedListPointers(startPointer) = tempPointer
    End if
End Sub
```

```
Sub InsertNode(ByVal NewItem)
Dim ThisNodePtr, NewNodePtr, PreviousNodePtr As Integer

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
            PreviousNodePtr = NULLPOINTER
            ThisNodePtr = StartPointer         ' start at beginning of list
Try
Do While (ThisNodePtr <> NULLPOINTER) 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 = NULLPOINTER 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 <> NULLPOINTER 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
```


Contact: 03004003666
Email: majidtahir61@gmail.com

VB Code

```
Sub DeleteNode(ByVal DataItem)
        Dim ThisNodePtr, PreviousNodePtr As Integer
        ThisNodePtr = StartPointer
        Try                ' start at beginning of list

            Do While ThisNodePtr <> NULLPOINTER 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 <> NULLPOINTER 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 = NULLPOINTER Then
            Console.WriteLine("No data in list")
        End If
        Do While CurrentNodePtr <> NULLPOINTER ' 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
```


Contact: 03004003666
Email: majidtahir61@gmail.com

VB Program for Linked Lists

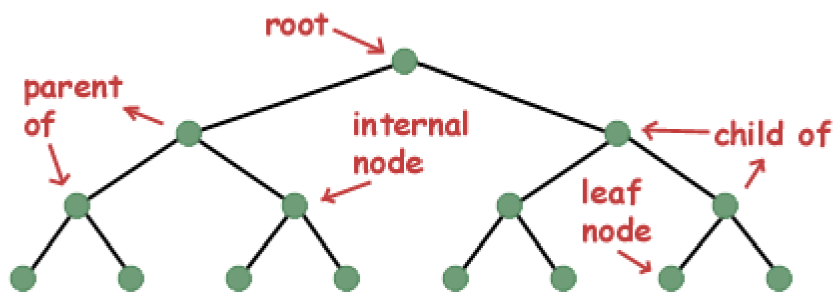
```
Module Module1
            ' NullPointer should be set to -1 if using array element with index 0
        Const NULLPOINTER = -1     ' Declare record type to store data and pointer
    Structure ListNode
        Dim Data As String
        Dim Pointer As Integer
    End Structure

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

    Sub InitialiseList()
        StartPointer = NULLPOINTER        ' 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 = NULLPOINTER       'last node of free list
    End Sub

    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 <> NULLPOINTER 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 = StartPointer
        Try          ' start at beginning of list

            Do While ThisNodePtr <> NULLPOINTER 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 <> NULLPOINTER Then ' node exists in list
```

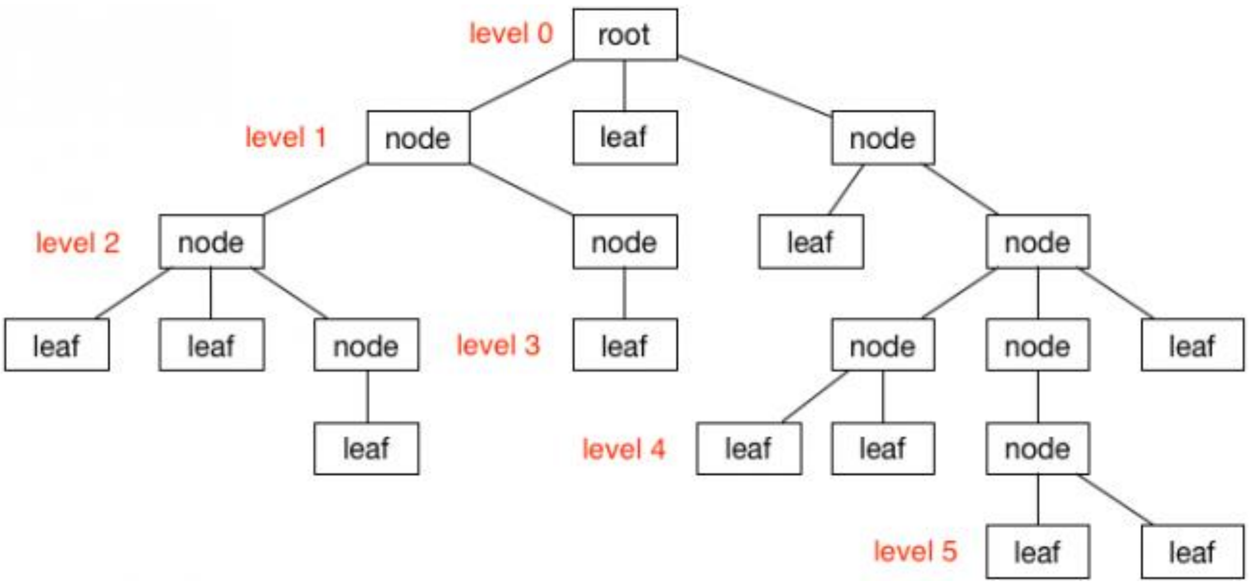
```
            If ThisNodePtr = StartPointer Then ' first node to be deleted

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

    Sub InsertNode(ByVal NewItem)

        Dim ThisNodePtr, NewNodePtr, PreviousNodePtr As Integer
        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
            PreviousNodePtr = NULLPOINTER
            ThisNodePtr = StartPointer              ' start at beginning of list
            Try
                Do While (ThisNodePtr <> NULLPOINTER) 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 = NULLPOINTER 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 = NULLPOINTER Then
            Console.WriteLine("No data in list")
        End If
        Do While CurrentNodePtr <> NULLPOINTER ' 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'.



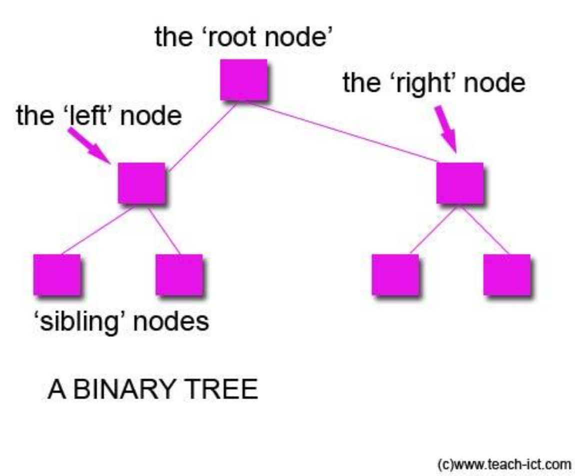


In computer science binary trees are used for different purposes.
In this chapter, you will use an ordered binary tree ADT as a binary search tree.

Tree Vocabulary:

The TREE is a general data structure that describes the relationship between data items or 'nodes'.
The parent node of a binary tree has only two child nodes.

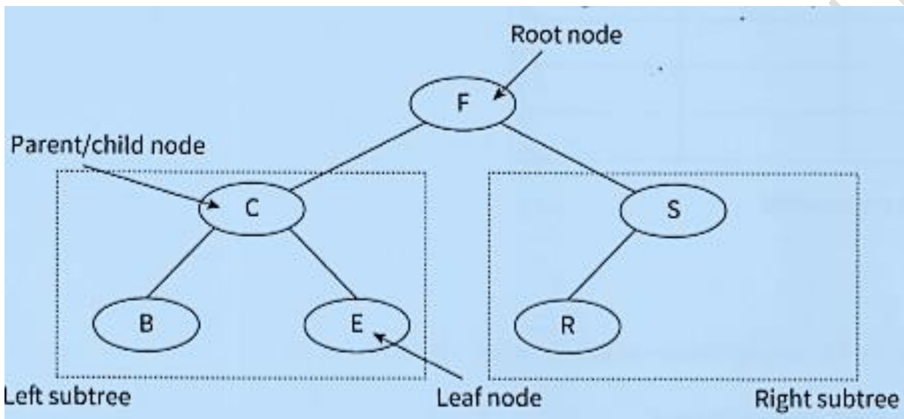
- Each data item within a **tree** is called a **node**
- The highest data item in **tree** is called **root** or **root node**
- Below the **root** lie a number of **other nodes**. The **root** is the **parent** of **nodes** immediately linked to it and these are **children** of **parent node**.
- If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



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**.

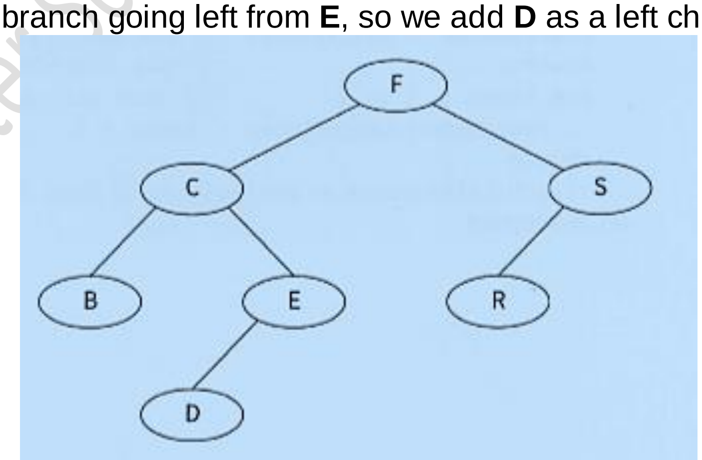


Figure 23.16 Conceptual diagram of adding a node to an ordered binary tree

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[l : 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
```


Contact: 03004003666
Email: majidtahir61@gmail.com

```
                        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 <> NULLPOINTER 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 <> NULLPOINTER 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 = NULLPOINTER             ' check if empty
tree
            If RootPointer = NULLPOINTER Then
                RootPointer = NewNodePtr
            Else  ' find insertion point
                ThisNodePtr = RootPointer
                Do While ThisNodePtr <> NULLPOINTER

                    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 <> NULLPOINTER 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 = NULLPOINTER 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
```


Contact: 03004003666
Email: majidtahir61@gmail.com

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.

Contact: 03004003666
Email: majidtahir61@gmail.com

[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
```

Hash Function using Visual Studio:

Dictionaries:

Dictionary. This collection allows fast key lookups. A generic type, it can use any types for its keys and values. Its syntax is at first confusing.
Many functions. Compared to alternatives, a Dictionary is easy to use and effective. It has many functions (like ContainsKey and TryGetValue) that do lookups.
Add example. This subroutine requires 2 arguments. The first is the key of the element to add. And the second is the value that key should have.

Note:Internally, Add computes the key's hash code value. It then stores the data in the hash bucket.

And:Because of this step, adding to Dictionary collections is often slower than adding to other collections like List.

VB.NET program that uses Dictionary Of String

```
Module Module1
    Sub Main()
        ' Create a Dictionary.
        Dim dictionary As New Dictionary(Of String, Integer)
        ' Add four entries.
        dictionary.Add("Dot", 20)
        dictionary.Add("Net", 1)
        dictionary.Add("Perls", 10)
        dictionary.Add("Visual", -1)
```

```
    End Sub
End Module
```

Add, error. If you add keys to the Dictionary and one is already present, you will get an exception. We often must check with ContainsKey that the key is not present.

Alternatively: You can catch possible exceptions with Try and Catch. This often causes a performance loss.

VB.NET program that uses Add, causes error

```
Module Module1
    Sub Main()
        Dim lookup As Dictionary(Of String, Integer) =
            New Dictionary(Of String, Integer)
        lookup.Add("cat", 10)
        ' This causes an error.
        lookup.Add("cat", 100)
    End Sub
End Module
```

Output

Unhandled Exception: System.ArgumentException:

An item with the same key has already been added.

at System.ThrowHelper.ThrowArgumentException...

ContainsKey. This function returns a Boolean value, which means you can use it in an If conditional statement. One common use of ContainsKey is to prevent exceptions before calling Add.

Also:Another use is simply to see if the key exists in the hash table, before you take further action.

Tip: You can store the result of ContainsKey in a Dim Boolean, and test that variable with the = and <> binary operators.

VB.NET program that uses ContainsKey

```
Module Module1
    Sub Main()
        ' Declare new Dictionary with String keys.
        Dim dictionary As New Dictionary(Of String, Integer)

        ' Add two keys.
        dictionary.Add("carrot", 7)
        dictionary.Add("perl", 15)

        ' See if this key exists.
        If dictionary.ContainsKey("carrot") Then
            ' Write value of the key.
            Dim num As Integer = dictionary.Item("carrot")
            Console.WriteLine(num)
```

```
        End If

        ' See if this key also exists (it doesn't).
        If dictionary.ContainsKey("python") Then
            Console.WriteLine(False)
        End If
    End Sub
End Module
```

References:
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