Data Structures

Data Structure = Organised Data + Allowed Operations.

If you recall, this is an extension of the concept of data type. We had defined adata type as

data Type = Permitted Data Values + Operations

Further, we had seen that simple data type can be used to built new scalar data types, for example subrange and enumerated type in Pascal. Similarly there are standard data structures which are often used in their own right and can form the basis for complex data structures. One such basic data structure called Array is also discussed in this unit Arrays are basic building block for more complex data structures. Designing and using data structures is an important programming skill. In this and in subsequent units, we are going to discuss various data structures. We may classify these data structures as linear and non-linear data structures. However, this is not the only way to classify data structures. In linear data structure the data items are arranged in a linear sequence like in an array. In a non-linear, the data items are not in sequence. An example of a non linear data structure is a tree. Data structures may also be classified as homogenous and non-homogenous data structures. An Array is a homogenous structure in which all elements are of same type. In non-homogenous structures the elements may or may not be of the same type. Records are common example of non-homogenoes data structures. Another way of classifying data structures is as static or dynamic data structures. Static structures are ones whose sizes and structures associated memory location are fixed at compile time. Dynamic structure are ones which expand or shrink as required during the program execution and their associated memory locations change. Records are a common example of non-homogenous data structures.

In this unit first we have discussed about the performance of an algorithm, You may find it very relevant as You read on the subsequent blocks and develop programs. Then we introduce the array data structure. Then the array declarations in Pascal and C are reviewed. A section is devoted to discussion on how single and multi-dimensional arrays are mapped to storage

Linear vs Nonlinear Data Structures

A data structure is a method for organizing and storing data, which would allow efficient data retrieval and usage. Linear data structure is a structure that organizes its data elements one after the other. Linear data structures are organized in a way similar to how the computer's memory is

organized. Nonlinear data structures are constructed by attaching a data element to several other data elements in such a way that it reflects a specific relationship among them. Nonlinear data structures are organized in a different way than the computer's memory.

Linear data structures

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a list where data elements can only be added or removed from the top of the list. A queue is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

Nonlinear data structures

In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run. Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures. A multidimensional array is simply a collection of one-dimensional arrays. A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements. A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Difference between Linear and Nonlinear Data Structures

Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this nonlinear structure, they might be difficult to be implemented in computer's linear memory compared to implementing linear data structures. Selecting one data structure

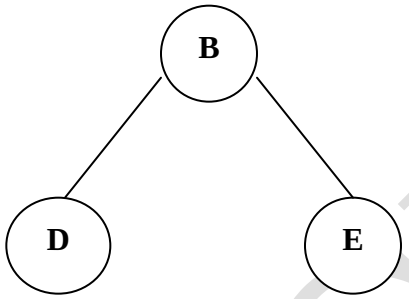
type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.

Tree

An important class of digraph, which involves for the description of hierarchy. A directed tree is an acyclic digraph which has one node called *root* with in degree 0, while other nodes have indegree 1. Every directed tree must have atleast one node. An isolated node is also called as directed tree. The node with outdegree as 0 is called as leaf. The length of the path from root to particular node level of the node. If the ordering of the node at each level is prescribed then the tree is called as ordered tree.

Subtree or forest

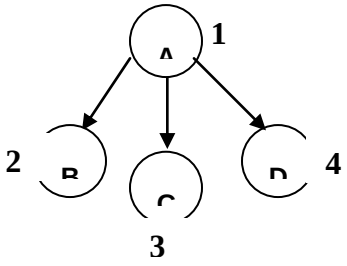
If we delete the root and the edges connecting the root to the nodes at level 1, then we get the Subtree with root as the node at level 1.



Sequential and other representation of Binary Tree

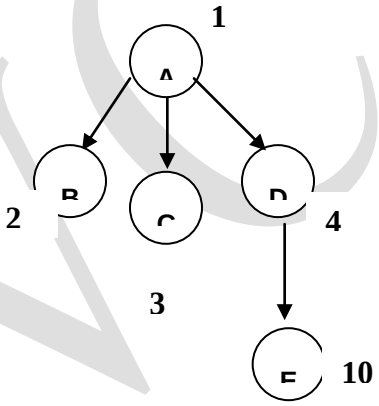
- Sequential Method For Complete Binary Trees
- Sequential Method For Incomplete Binary Trees
- Using Preorder Traversing method
- Using Post order Traversing method
- Representation Using Parent Node

Sequential Method For Complete Binary Trees



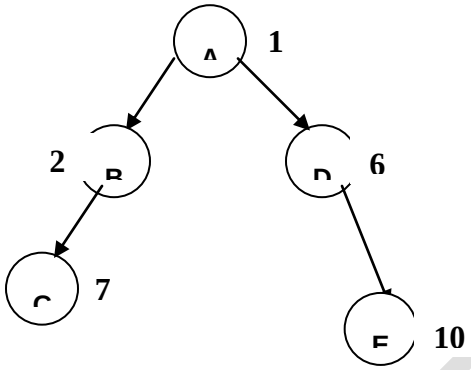
Position	1	2	3	4
Information	A	B	C	D

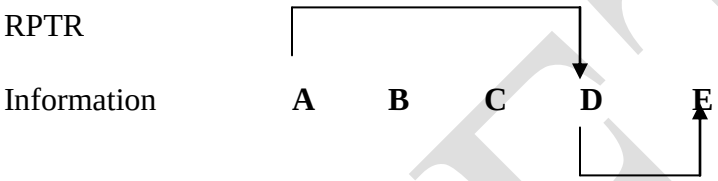
Sequential Method For Incomplete Binary Trees



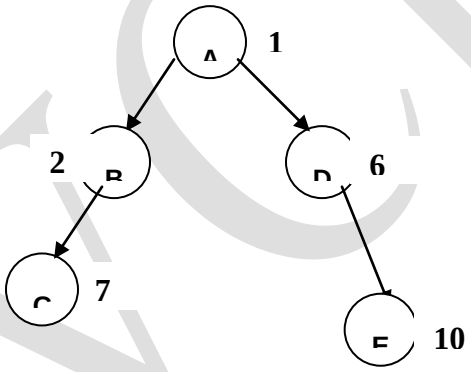
Position	1	2	3	4	5	6	7	8	9	10
Information	A	B	C	D	–	–	–	–	–	E

Using Preorder Traversing method





Using Post order Traversing method

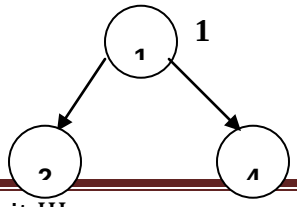


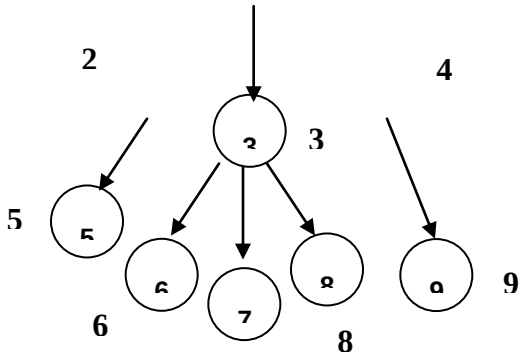
C B E D A

POST

Degree	0	1	0	1	2

Representation **Using Parent Node**





i	1	2	3	4	5	6	7	8	9
Father (i)	0	1	1	1	2	3	3	3	4

Conversion of General Tree to Binary Tree

General Algorithm Used to Convert the General Tree to Binary Tree

1. Create a head node for the binary tree and push its address and level number on the stack.
2. Repeat thru step 6 while there still remains data.
3. Input a current node description.
4. Create a tree node and initialize its contents,
5. If the level number of the current node is greater than that of the top of the stack then connect parent to its left offspring (current node)

Else Remove from the stack all nodes whose level numbers are greater than that of the current node

Connect the left child on the stack to the current node

Remove the left child from the stack

6. Push the current node description on to the stack
7. Finished.

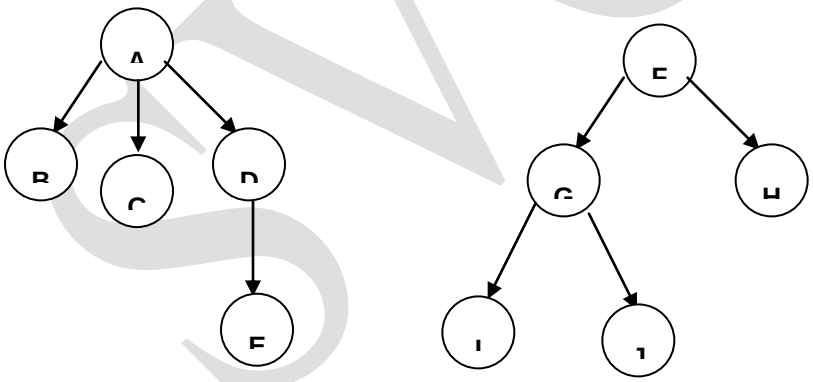
Pseudo – code Instruction

1. HEAD ⇐ NODE
   LPTR (HEAD) ← NULL
   RPTR (HEAD) ← HEAD
   NUMBER [1] ← 0
   LOC [1] ← HEAD
   TOP ← 1
2. **Repea**t thru step 6 **while** there remains input

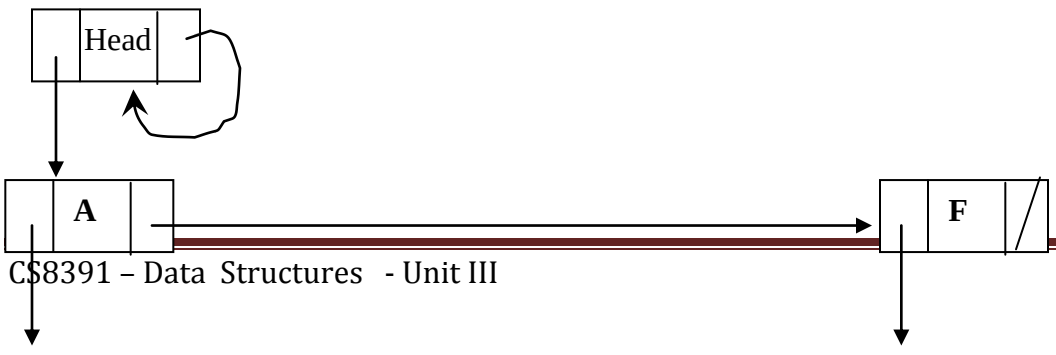
3. **Read** (LEVEL, NAME)
4. NEW ⇐ NODE
   LPTR (NEW) ← RPTR (NEW) ← NULL
   DATA (NEW) ← NAME
5. PRED_LEVEL ← NUMBER [TOP]
   PRED_LOC ← LOC [TOP]
   If LEVEL ← PRED_LEVEL
   Then LPTR (PRED_LOC) ← NEW
   Else Repeat while PRED_LEVEL > LEVEL
   TOP ← TOP – 1
   PRED_LEVEL ← NUMBER [TOP]
   PRED_LOC ← LOC [TOP]
   If PRED_LEVEL<LEVEL
   Then Write ('Mixed Level Numbers')
   Exit
   RPTR (PRED_LOC) ← NEW
   TOP ← TOP – 1

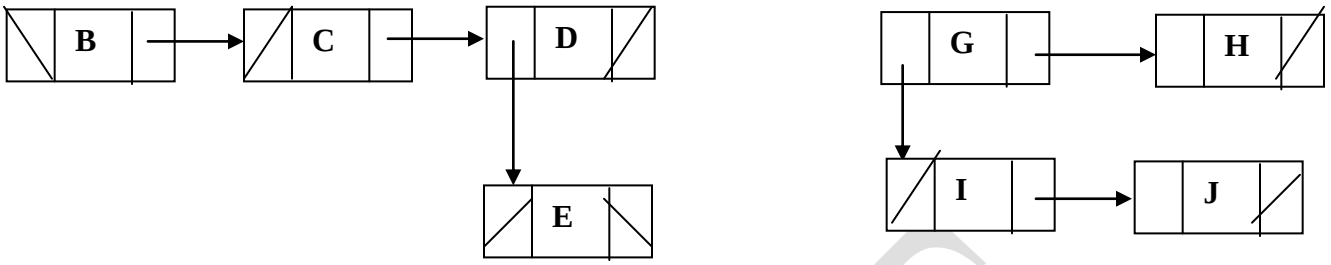
6. TOP ← TOP+1
   NUMBER [TOP] ← LEVEL
   LOC [TOP] ← NEW
7. **Exit**

General Tree



Equivalent Binary Tree for the above General Tree





Traversing Methods

1. Pre – order method
2. In – order method
3. Post – order method
4. Converse Pre – order method
5. Converse In – order method
6. Converse post – order method

Pre – order method

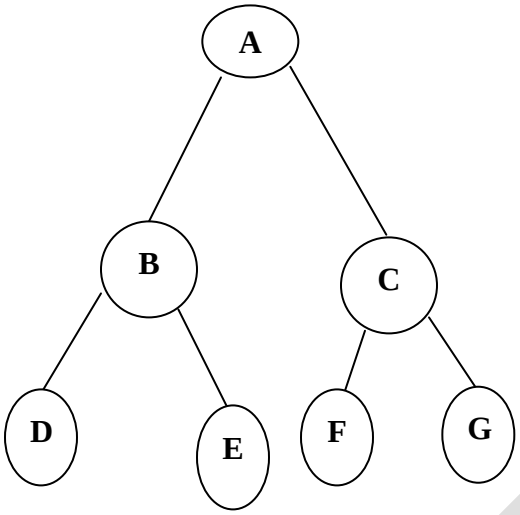
This method gives the tree key value in the following manner: -

1. Process the root
2. Traverse the left subtree
3. Traverse the right Subtree

Procedure PREORDER (T)

1. **If** T ≠ NULL
   then write (DATA (T))
   else write ('Empty Tree')
   return
2. **If** LPTR (T) ≠ NULL
   then Call PREORDER (LPTR (T))
3. **If** RPTR (T) ≠ NULL
   then Call PREORDER (RPTR (T))

4. Return



The sequence obtained by preorder traversal: -

A, B, C, D, E, F, G

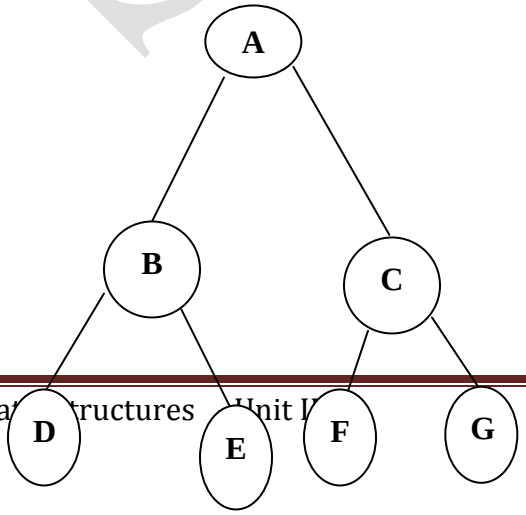
In – order method

This method gives the tree key value in the following manner: -

1. Traverse the left subtree
2. Process the root
3. Traverse the right Subtree

Procedure INORDER (T)

1. **If** T ≠ NULL
   then write ('Empty Tree')
   return
2. **If** LPTR (T) ≠ NULL
   then Call INORDER (LPTR (T))
3. write (DATA (T))
4. **If** RPTR (T) ≠ NULL
   then Call INORDER (RPTR (T))
5. **Return**



The sequence obtained by preorder traversal: -

D, B, E, A, F, C, G

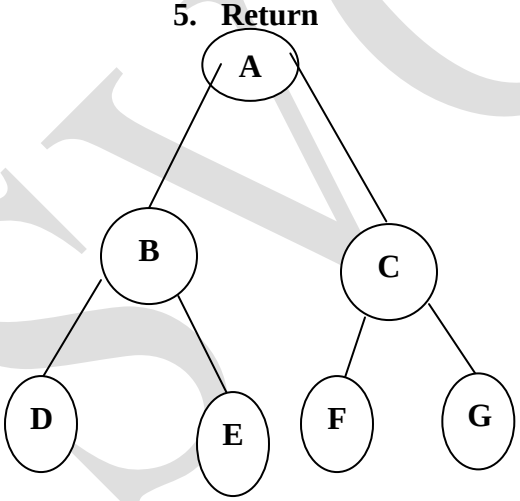
Post – order method

This method gives the tree key value in the following manner: -

1. Traverse the left subtree
2. Traverse the right Subtree
3. Process the root

Procedure POSTORDER (T)

1. **If** T ≠ NULL
   then write ('Empty Tree')
   return
2. **If** LPTR (T) ≠ NULL
   then Call POSTORDER (LPTR (T))
3. **If** RPTR (T) ≠ NULL
   then Call POSTORDER (RPTR (T))
4. **write** (DATA (T))
5. **Return**



The sequence obtained by preorder traversal: -

D, E, B, F, G, C, A

Converse Pre – order method

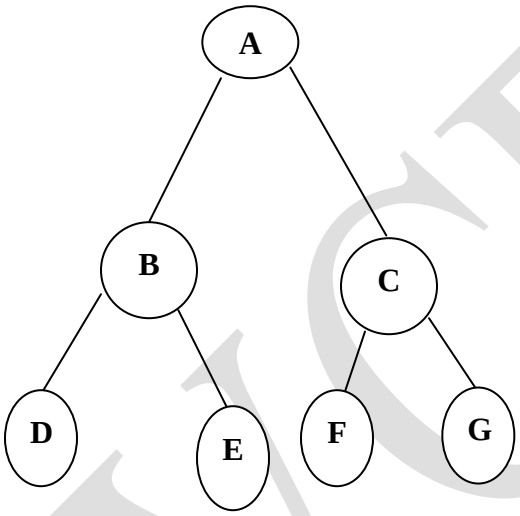
This method gives the tree key value in the following manner: -

1) Process the root

2) Traverse the right Subtree
3) Traverse the left subtree

Procedure CPREORDER (T)

1. **If** T ≠ NULL
   then write (DATA (T))
   else write ('Empty Tree')
   return
2. **If** RPTR (T) ≠ NULL
   then Call CPREORDER (RPTR (T))
3. **If** LPTR (T) ≠ NULL
   then Call CPREORDER (LPTR (T))
4. **Return**



The sequence obtained by preorder traversal: -

A, C, G, F, B, E, D

Converse In – order method

This method gives the tree key value in the following manner: -

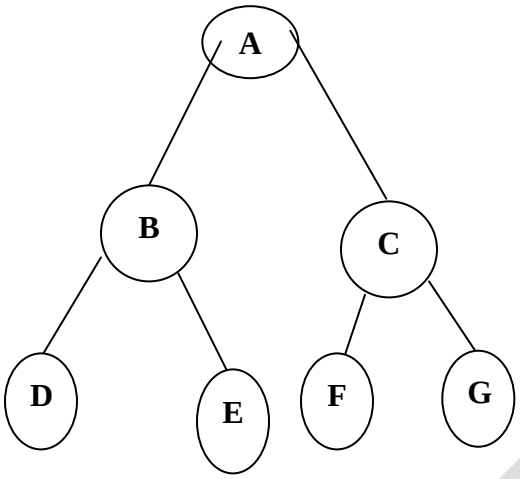
1) Traverse the right Subtree
2) Traverse the left subtree
3) Process the root

Procedure CINORDER (T)

1. If T ≠ NULL
then write ('Empty Tree')
return
2. If RPTR (T) ≠ NULL
then Call CINORDER (RPTR (T))
3. write (DATA (T))
4. If LPTR (T) ≠ NULL

then Call CINORDER (LPTR (T))

5. Return



The sequence obtained by preorder traversal: -

G, C, F, A, E, B, D

Converse Post – order method

This method gives the tree key value in the following manner: -

1. Traverse the right Subtree
2. Traverse the left Subtree
3. Process the root

Procedure CPOSTORDER (T)

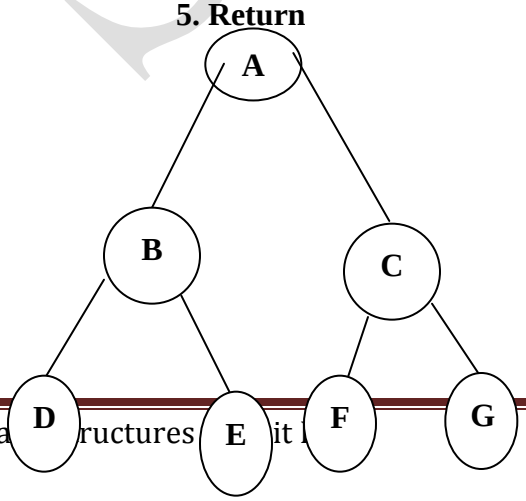
1. If T ≠ NULL
then write ('Empty Tree')
return

2. If RPTR (T) ≠ NULL
then Call POSTORDER (RPTR (T))

3. If LPTR (T) ≠ NULL
then Call POSTR (LPTR (T))

4. write (DATA (T))

5. Return



The sequence obtained by preorder traversal: -

D, E, B, F, G, C, A

7

HEIGHT BALANCED TREE

A binary tree of height h is completely balanced or balanced if all leaves occur at nodes of level h or h-1 and if all nodes at levels lower than h-1 have two. children. According to this definition, the tree in figure 1(a) is balanced, because all leaves occur at levels 3 considering at level 1 and all node's at levels 1 and 2 have two children. Intuitively we might consider a tree to be well balanced if, for each node, the longest paths from the left of the node are about the same length as the longest paths on the right.

More precisely, a tree is height balanced if, for each node in the tree, the height of the left subtree differs from the height, of the right subtree by no more than 1. The tree in figure 2(a) height balanced, but it is not completely balanced. On the other hand, the tree in figure 2(b) is completely balanced tree.

An almost height balanced tree is called an AYL tree after the Russian mathematician G. M. Adelson - Velskii and E. M. Lendis, who first defined and studied this form of a tree. AVL Tree may or may not be perfectly balanced.

Let us determine how many nodes might be there in a balanced tree of height h.

The, root will be the only node at level 1;

Each subsequent levels will be as full as possible i.e. 2 nodes at level 2, 4 nodes at level 3 and so on, i.e. in general there will be 21-1 nodes at level 1. Therefore the number of nodes from level 1 through level h-1 will be

$$1 + 2 + 22 + 23 + ... + 2h-2 = 2h-1 - 1$$

The number of nodes at level h may range from a single node to a maximum of 2h-1 nodes. Therefore, the total number of nodes n of the tree may range for (2h-1-1+1) to (2h-1-1+2h-1)

or 2h-1 to 2h -1.

BUILDING HEIGHT BALANCED TREE

Each node of an AVL tree has the Property that the height of the left subtree is either one more, equal, or one less than the height of the right subtree. We may define a balance factor (BF) as

BF = (Height of Right- subtree - Height of Left- subtree)

Further

If two subtree are of same height

BF = 0

if Right subtree is higher

BF = +1

if Left subtree is higher

BF = -1

For example balance factor are stated near the nodes in Figure 3. BF of the root node is zero because height of the right subtree and the left subtree is three. The BF at the node DIN is -17 because the height of its left subtree is 2 and of right subtree is 1 etc.