UNIT2 TREES

Syllabus

Basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree. Huffman Tree (Concept and Use), Binary Search Tree (BST), BST operations, Threaded binary search tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary search tree, in order traversal of in-order threaded binary search tree.

- #Exemplar/Case Use of binary tree in expression tree-evaluation and Huffman's coding

Objectives

- To learn the basic concept of non linear data structures.
- To learn what a tree is and how it is used.
- To implement the **tree** abstract data type using multiple internal representations.
- To learn different types of tree data structure.
- To see how trees can be used to solve a wide variety of problems

Outcome

At the end of this unit students will be able to

- Understand meaning of non linear data structures.
- Understand tree data structure.
- Implement tree ADT.
- Implement various applications of tree.

Difference Between Linear and Nonlinear Data Structures

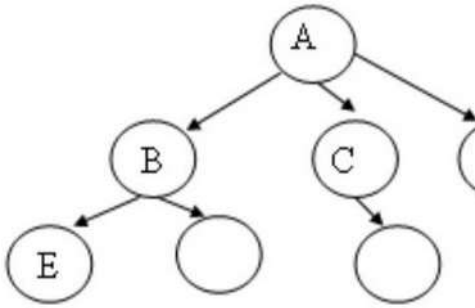
BASIS FOR COMPARISON	LINEAR DATA STRUCTURE	NON-LINEAR DATA STRUCTURE
Basic	In the linear data structure, the data is organized in a linear order in which elements are linked one after the other.	In the non-linear data structure the data element are not stored in a sequent manner rather the element are hierarchically related.
Traversing of the data	The traversing of data in the linear data structure is easy as it can make all the data elements to be traversed in one go, but at a time only one element is directly reachable.	On the contrary, in the non linear data structure, the nodes are not visited sequentially and cannot be traversed in one go.
Ease of implementation	Simpler	Complex
Levels involved	Single level	Multiple level
Examples	Array, queue, stack, linked list, etc.	Tree and graph.

Introduction to trees:

Topic		
Definition	1 hour	2M
Basic terminologies	2M	
Applications of Tree	2M	

Tree

- A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
  - Tree is a sequence of nodes
  - There is a starting node known as a root node
  - Every node other than the root has a parent node.
  - Nodes may have any number of children



A has 3 children, B, C, D
A is parent of B

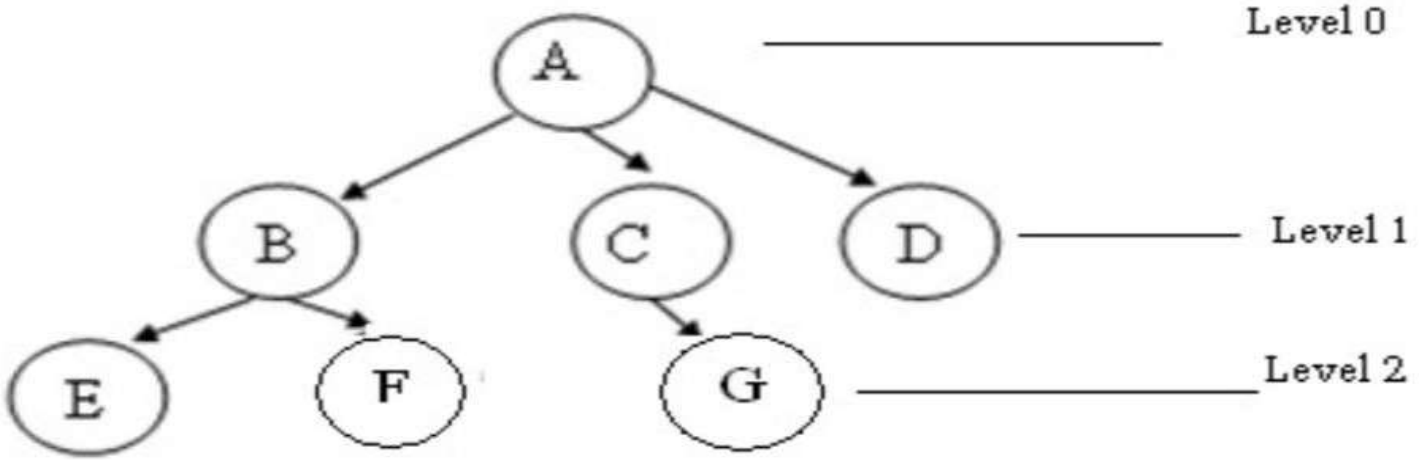
Basic Terminologies:

- Root − Node at the top of the tree is called root.
- Parent − Any node except root node has one edge upward to a node called parent.
- Child − Node below a given node connected by its edge downward is called its child node.
- Sibling – Child of same node are called siblings
- Leaf − Node which does not have any child node is called leaf node.
- Sub tree − Sub tree represents descendants of a node.
- Levels − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.

- keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

Basic Terminologies:

- Degree of a node:
  - The degree of a node is the number of children of that node
- Degree of a Tree:
  - The degree of a tree is the maximum degree of nodes in a given tree
- Path:
  - It is the sequence of consecutive edges from source node to destination node.
  - Height of a node:
  - The height of a node is the max path length from that node to a leaf node.
- Height of a tree:
  - The height of a tree is the height of the root
- Depth of a tree:
  - Depth of a tree is the max level of any leaf in the tree



- ✔ A is the root node
- ✔ B is the parent of E and F
- ✔ D is the sibling of B and C
- ✔ E and F are children of B
- ✔ E, F, G, D are external nodes or leaves
- ✔ A, B, C are internal nodes
- ✔ Depth of F is 2
- ✔ the height of tree is 2
- ✔ the degree of node A is 3
- ✔ The degree of tree is 3

Characteristics of trees

- Non-linear data structure
- Combines advantages of an ordered array
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list

Applications:

- Directory structure of a file store
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- Used in compression algorithms, such as those used by the .jpeg and .mp3 file- formats.

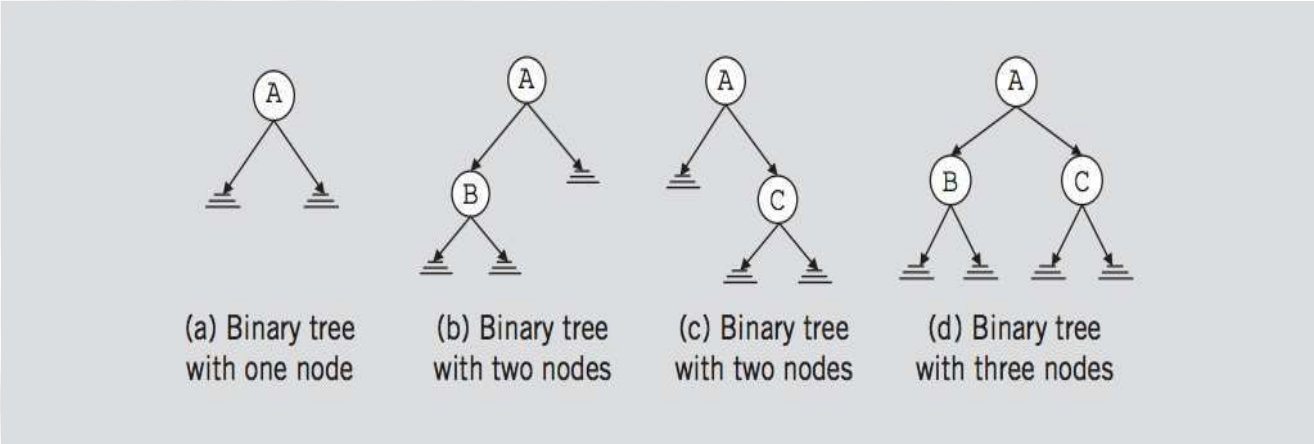
Binary Tree

Introduction to Binary tree	1 hour	
Definition	2M	
Binary tree properties		
Types of Binary Tree	4M	
Conversion of General tree to Binary tree	4M	
Representation techniques	1 hour	4M
Traversal techniques	4M	

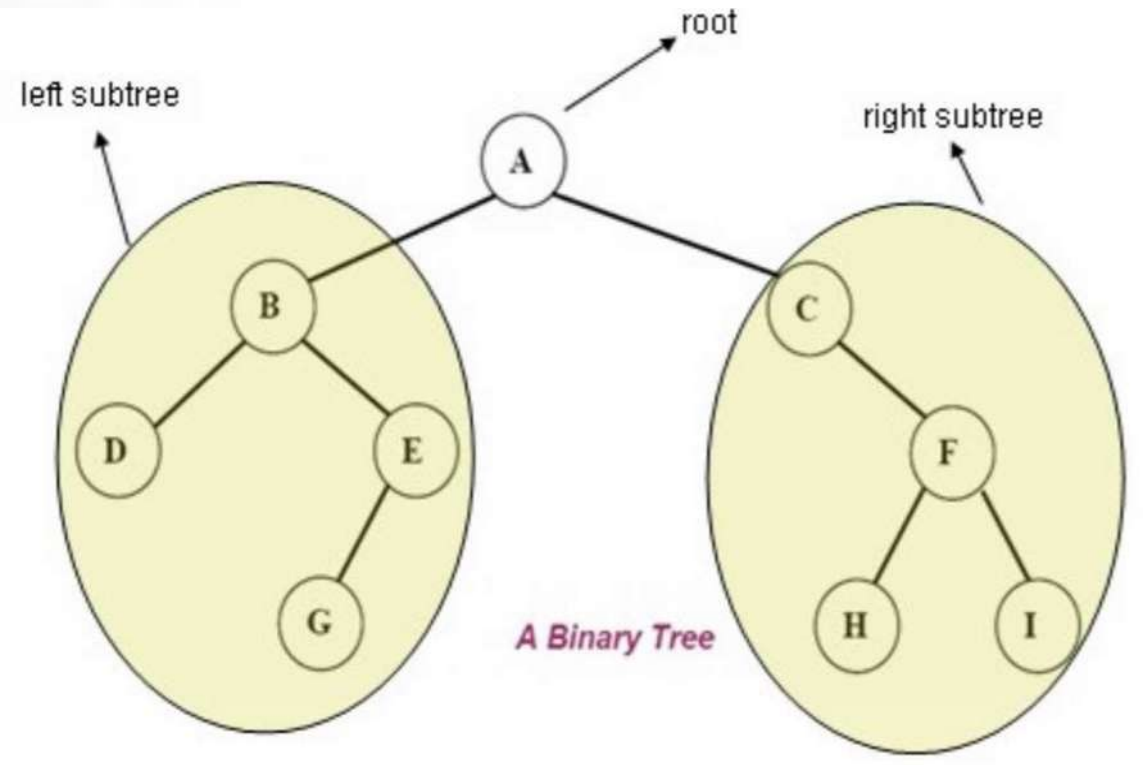
Binary Tree Definition:

- A binary tree, *T*, is either empty or such that

  I. T has a special node called the root node



The following figure shows a binary tree with 9 nodes where A is the root



Binary Tree Properties

- A tree with n nodes has exactly (n-1) edges or branches .
- If a binary tree contains m nodes at level L, it contains at most 2m nodes at level L+1
- Maximum number of nodes in a binary tree h is $2^h - 1$.
- The minimum height of a binary tree with n nodes is $\log_2 (n+1) - 1$
- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most 2L nodes at level L.
- A binary tree with n internal nodes has n+1 external nodes.

Types of Binary Tree

- Complete binary tree
- Strictly binary tree
- Full Binary Tree
- Skewed Binary tree

Full Binary Tree

- A binary tree is said to be full if each of its node has two children or no children at all and all leaf nodes should be on the same level.

Fig.

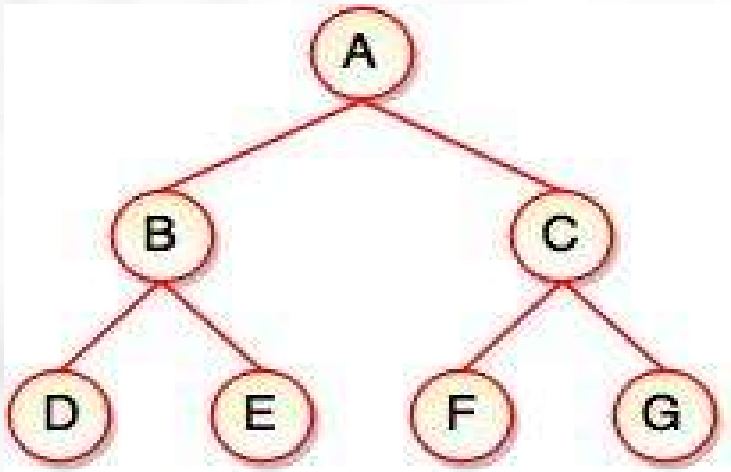
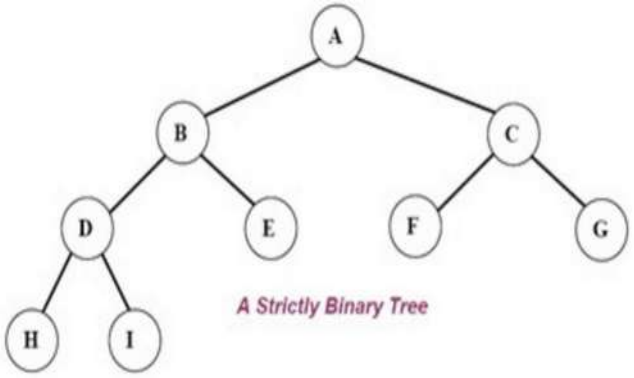


Fig. Full Binary Tree

Strictly Binary Tree

- If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a strictly binary tree.
- Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.
- A strictly binary tree with

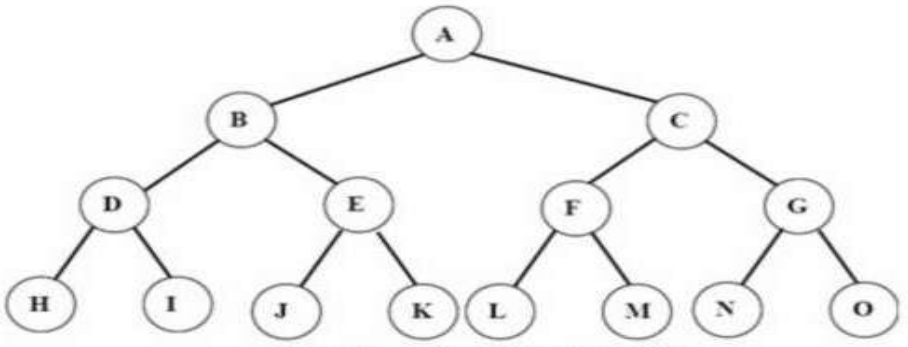
N leaves always contains 2N – 1 nodes



A Strictly Binary Tree

Complete binary tree

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A complete binary tree of depth d is called strictly binary tree if all of whose leaves are at level d.
- A complete binary tree has 2^d nodes at every depth d and 2^d -1 non leaf nodes

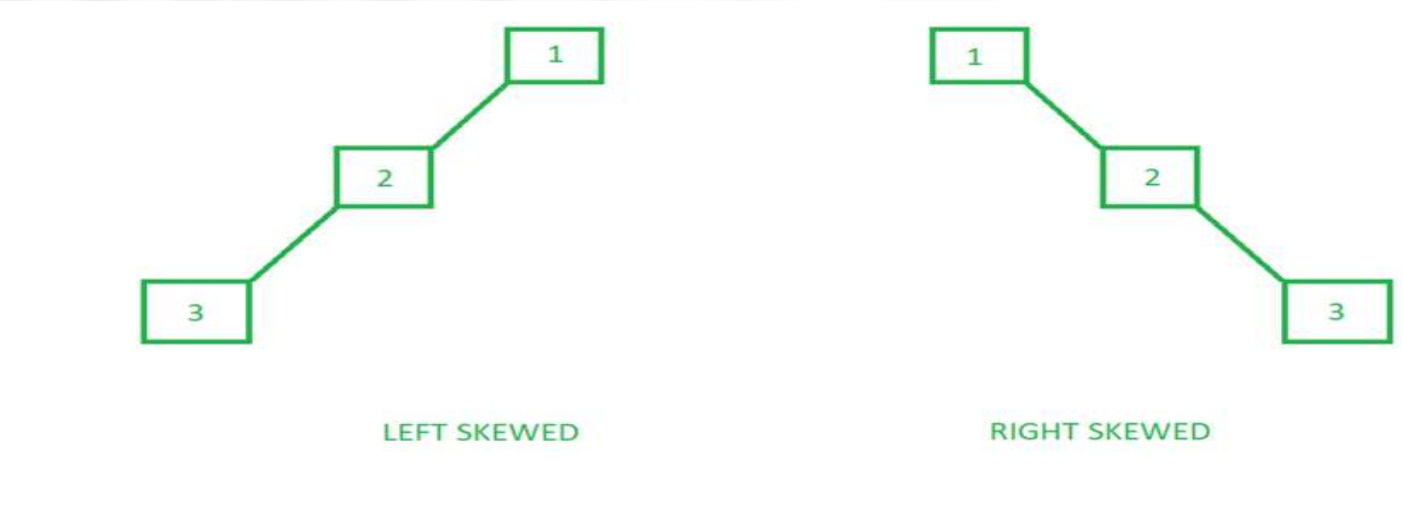


A complete Binary Tree of depth 3

Skewed Binary Tree

A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.

- **Types of skewed tree:**



Conversion of general tree to binary tree

- Algorithm

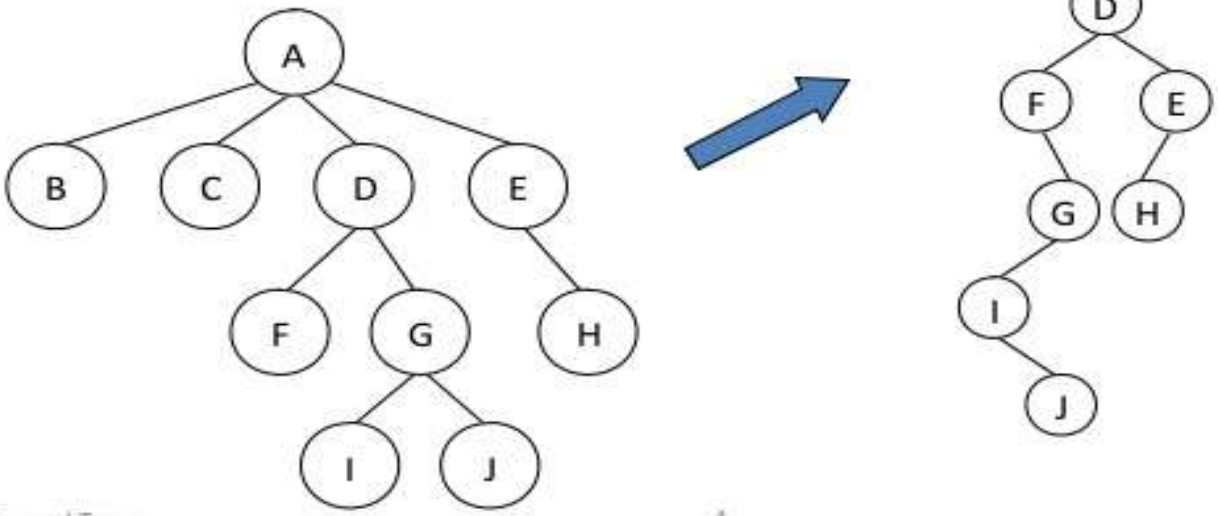
- The root of the Binary Tree is the Root of the General Tree.

- The left child of a node in the General Tree is the Left child of that node in the Binary Tree.

- The right sibling of any node in the General Tree is the Right child of that node in the Binary Tree.

Converting to a Binary Tree

- Binary tree left child
  = leftmost child
- Binary tree right child
  = right sibling



Representation of Binary tree

- Binary tree can be represented using two ways:

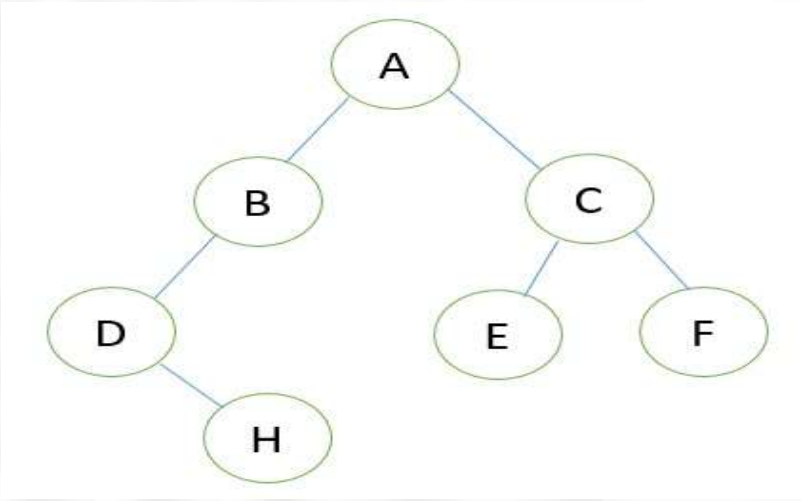
  1) Using sequential organization

  2) Using linked organization

Sequential Representation/Array Representation:

- A binary tree has a simple array representation. Suppose we start numbering the nodes of binary tree from zero onwards, left to right, beginning at the root, then there childs level wise.
- Then we can store the various data items in the corresponding elements of an array.
- Index of left child of a node i= 2 i
- Index of the right child of a node i=2 i + 1
- Index of the parent of a node i=i/2
- Sibling of a node i will be found at the location i+1,if i is a left child of it’s parent.

Tree and it's Array Representation



A	B	C	D	-	E	F	-	H		

Linked organization of Binary Tree:

- To represent node of binary tree in computer memory, the structure is:

```
struct Node {
Node *left_child;
int Data;
Node *right_child;
};
```

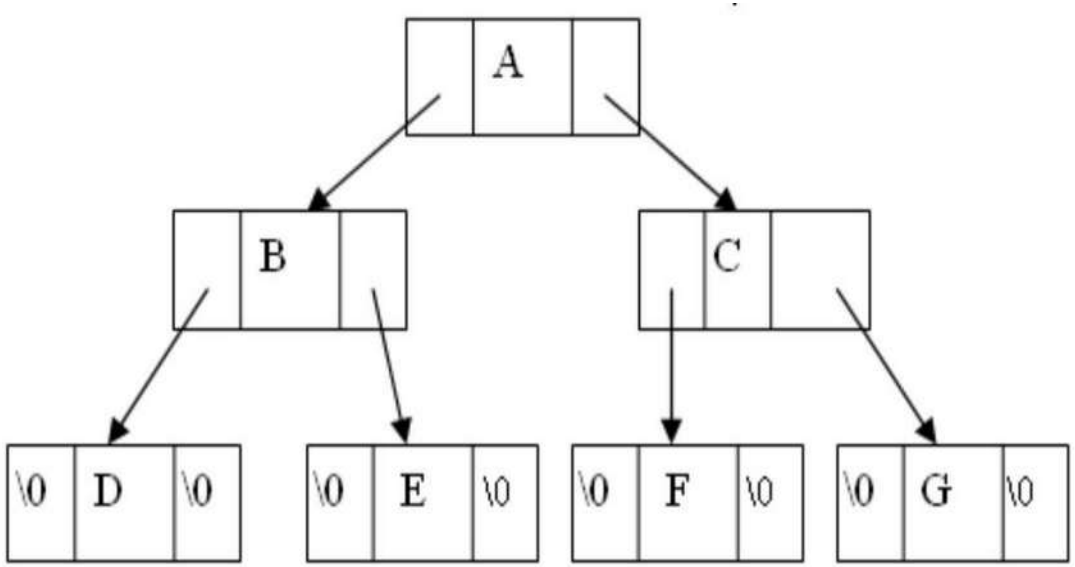


Fig: Structure of Binary tree

Tree traversal

Pre-order, In-order, Post-order

- Pre-order

<root><left><righ
t>

- In-order

<left><root><righ
t>

- Post-order

<left><right><roo
t>

Pre-order Traversal

- The preorder traversal of a nonempty binary tree is defined as follows:
  - Visit the root node
  - Traverse the left sub-tree in preorder
  - Traverse the right sub-tree in preorder

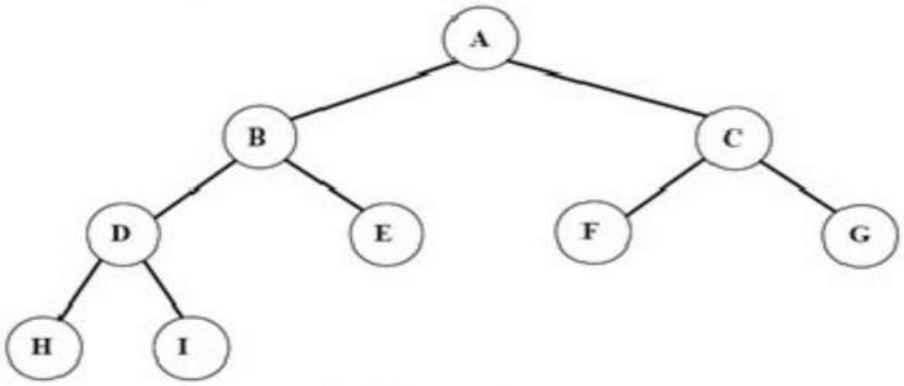


fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

In-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in in-order
  - Visit the root node
  - Traverse the right sub-tree in inorder

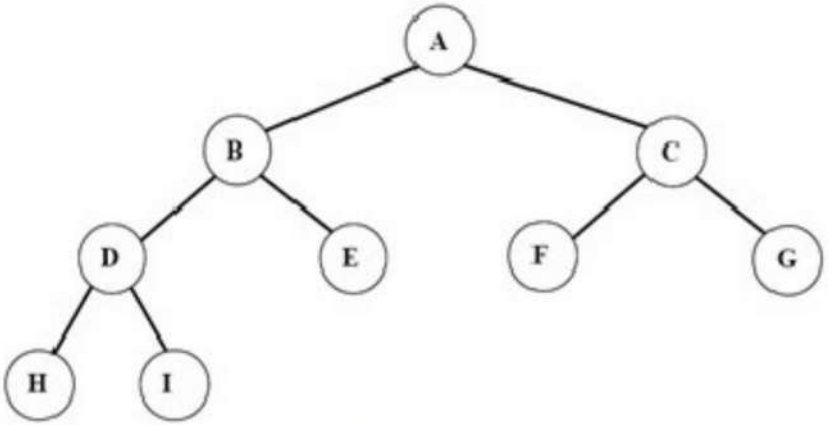


fig Binary tree

Post-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in post-order
  - Traverse the right sub-tree in post-order
  - Visit the root node

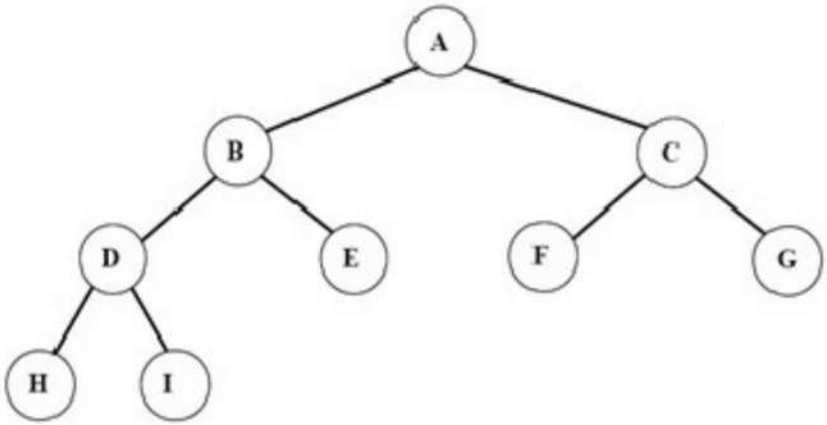


fig Binary tree

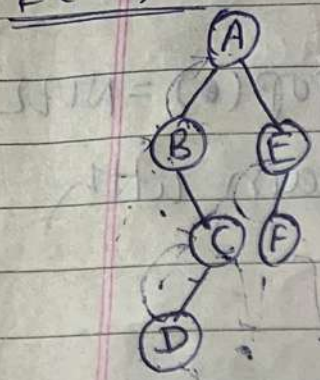
Operations on Binary tree

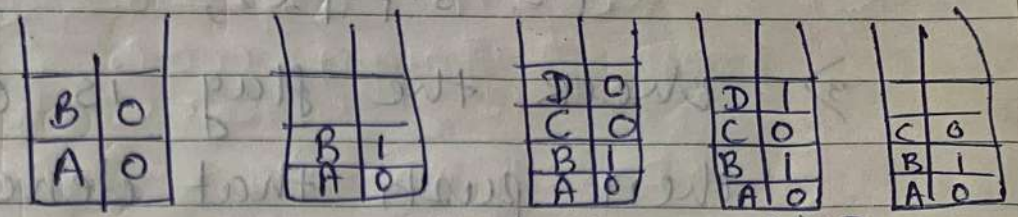
Create/Recursive Traversals	1 hour	6M/4M
Non recursive Traversals	1 hour	6M/4M

```
void preorder(node *root)
{
   if (root != NULL)
   {
      cout<<root->data<<" ";
      preorder(root->left);
      preorder(root->right);
   }
}
```

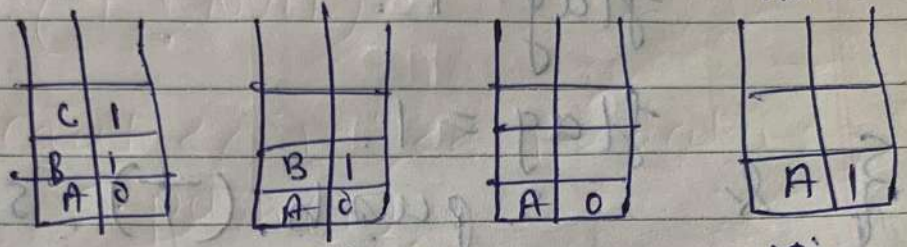
```
void preorder_nonrec(node *T)
{
   stack s;
  while(T != NULL)
   {
      cout<<T->data<<" ";
              s.push(T);
              T=T->left;
       }
   While(!s.empty())
       {
              T=s.pop();
              T=T->right;
              while(T != NULL)
              {
              cout<<T->data<<" ";
                     s.push(T);
                     T=T->left;
              }
       }
}
```

```
void Inorder_nonrec(node *T)
{
   stack s;
  while(T != NULL)
   {
       s.push(T);
              T=T->left;
       }
   While(!s.empty())
       {
              T=s.pop();
              cout<<T->data<<" ";
              T=T->right;
              while(T != NULL)
              {
              s.push(T);
                     T=T->left;
              }
       }
}
```

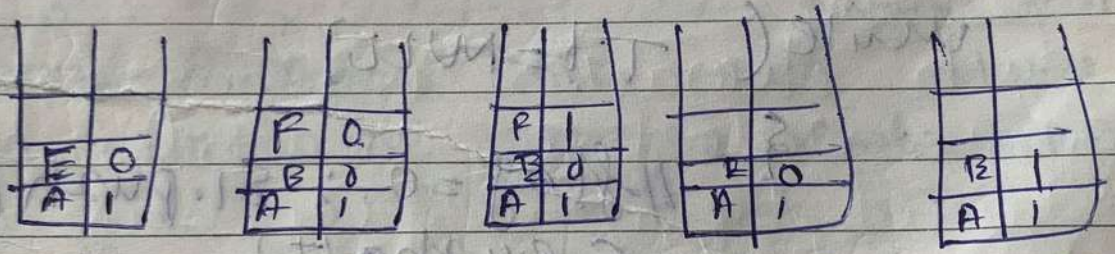




Visit D



Visit C. Visit B.



visit F. visit E visit A.

→ D C B F E A.

1) while going to left, addrs. of the treenode along with flag=0 is pushed onto the stack.
2) During backtracking to traverse the right subtree an element is popped & if flag is 0 then it pushed back with flag = 1.
3) When we return from right subtree pop. & flag = 1

Non-recursive postorder traversal

* 1> while (T != NULL)
{
S.push (T); S1.push (NULL)
T = T → lptr;
}

2> while (! S.empty())
{
T = S.pop();

3> check the flag is 0 if (S1.pop() = NULL) then push that element again with flag 1.
flag = 1.

{ & S.push (T); S1.push ((node *)1);
T = T → right.
while (T != NULL)
{ //flag = 0; S1.push (NULL);
S.push (T);
T = T → lptr;
}

4> else if then flag = 1.
P.P & Print the data.

5> end.

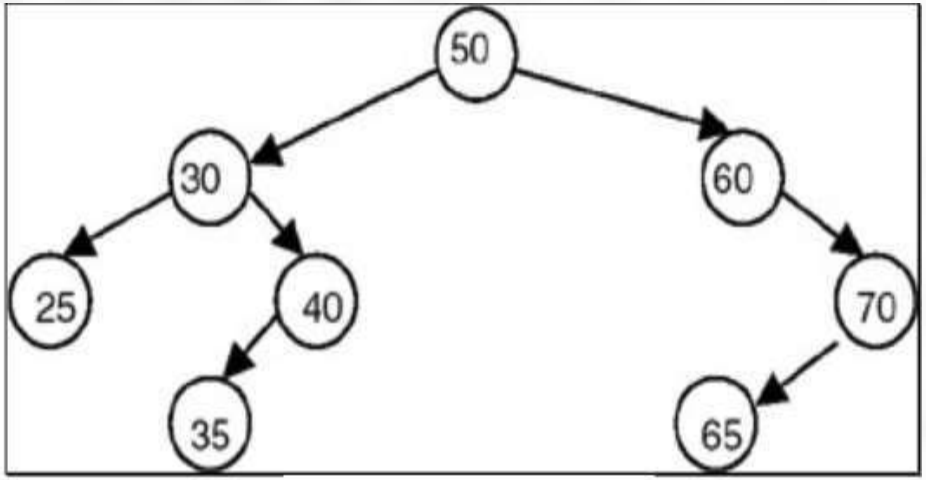
Binary Search Tree (BST)

Topic		
Definition	**2 hour**	**2M**
Need	**2M**	
Operations on BST	**4M**	

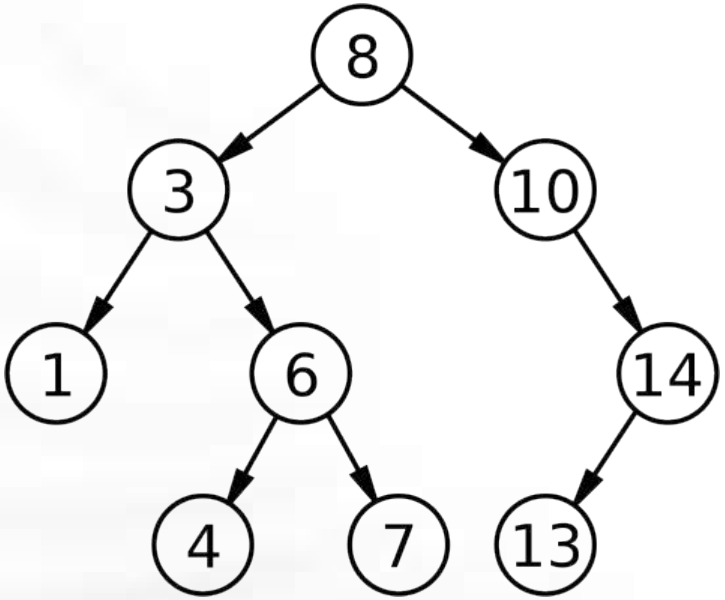
Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:
  - All keys in the left sub-tree of the root are smaller than the key in the root node
  - All keys in the right sub-tree of the root are greater than the key in the root node
  - The left and right sub-trees of the root are again binary search trees

Binary Search Tree (BST)



The binary search tree.



Why Binary Search Tree ?

- Let us consider a problem of searching a list.

- If a list is ordered searching becomes faster if we use contiguous list(array).
- But if we need to make changes in the list, such as inserting new entries or deleting old entries, (SLOWER!!!!) because insertion and deletion in a contiguous list requires moving many of the entries every time.

Why Binary Search Tree ?

- So me may think of using a linked list because it permits insertion and deletion to be carried out by adjusting only few pointers.
- But in an n-linked list, there is no way to move through the list other than one node at a time, permitting only sequential access.
- Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary search tree, we find that we can search for a key in O(logn)

Binary Search Tree (BST)

Time Complexity			
	Array	**Linked List**	**BST**
Search	O(n)	O(n)	O(logn)
Insert	O(1)	O(1)	O(logn)
Delete	O(n)	O(n)	O(logn)

Operations on Binary Search Tree (BST)

- Following operations can be done in BST:
  - Search(k, T): Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
  - Insert(k, T): Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
  - Delete(k, T):Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
  - FindMin(T), FindMax(T): Find minimum and maximum element from the given nonempty BST.

Searching Through The BST

- Compare the target value with the element in the root node
  - ✓ If the target value is equal, the search is successful.
  - ✓ If target value is less, search the left subtree.
  - ✓ If target value is greater, search the right subtree.
  - ✓ If the subtree is empty, the search is unsuccessful.

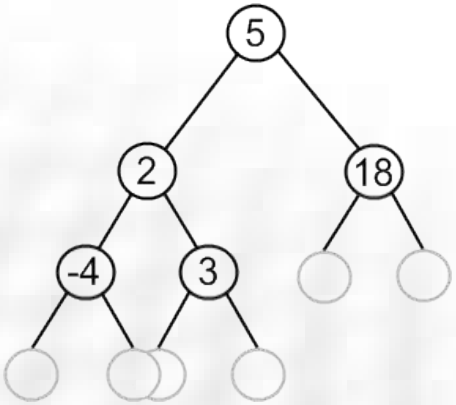
Insertion of a node in BST

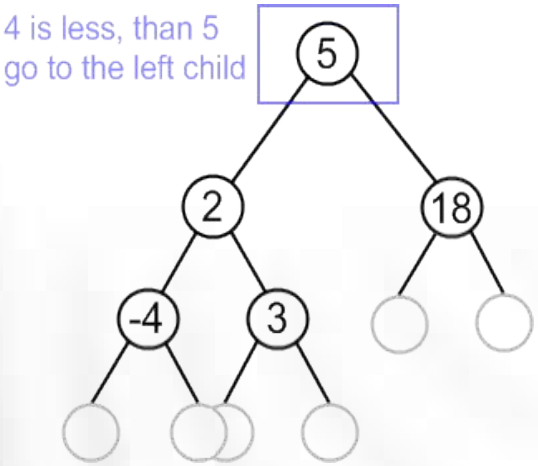
- To insert a new item in a tree, we must first verify that its key is different from those of existing elements.
- If a new value is less, than the current node's value, go to the left subtree, else go to the right subtree.

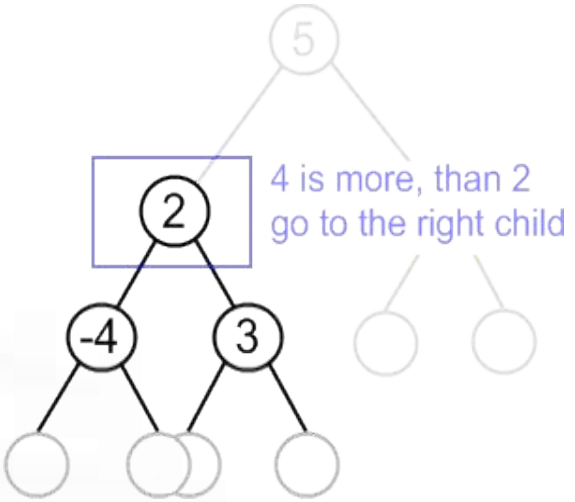
- Following this simple rule, the algorithm reaches a node, which has no left or right subtree.
- By the moment a place for insertion is found, we can say for sure, that a new value has no duplicate in the tree.

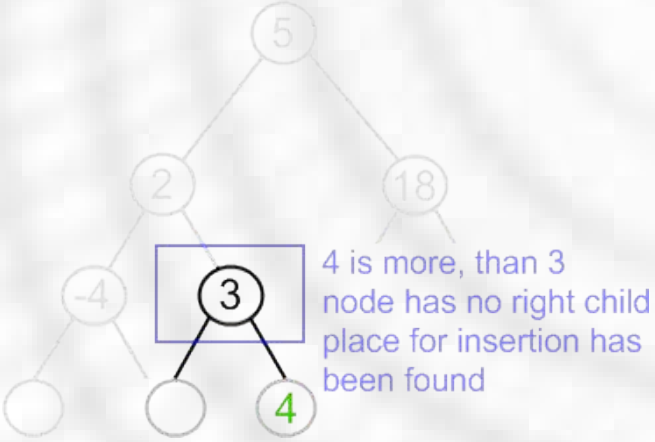
Algorithm for insertion in BST

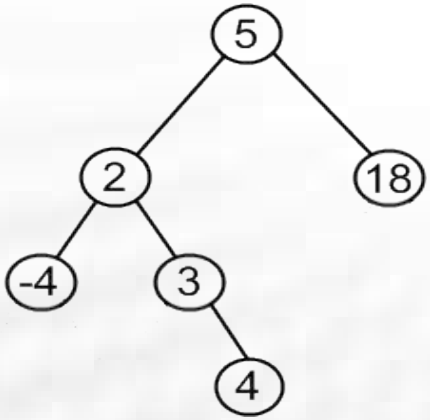
- Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
- if a new value is less, than the node's value:
  - if a current node has no left child, place for insertion has been found;
  - otherwise, handle the left child with the same algorithm.
- if a new value is greater, than the node's value:
  - if a current node has no right child, place for insertion has been found;
  - otherwise, handle the right child with the same algorithm.


5
2
18
-4
3


4 is less, than 5
go to the left child
5
2
18
-4
3


5
2
4 is more, than 2
go to the right child
-4
3


5
2
18
-4
3
4 is more, than 3
node has no right child
place for insertion has
been found
4

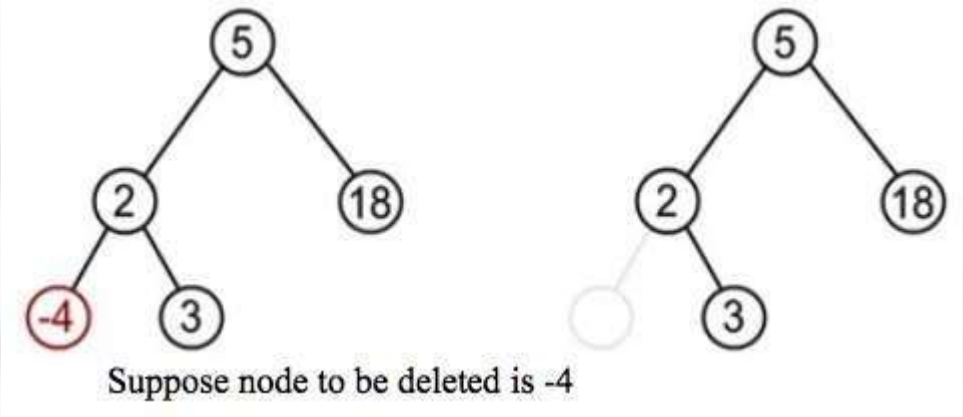

5
2
18
-4
3
4

Deleting a node from the BST

- While deleting a node from BST, there may be three cases:

1.The node to be deleted may be a leaf node:

- In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.

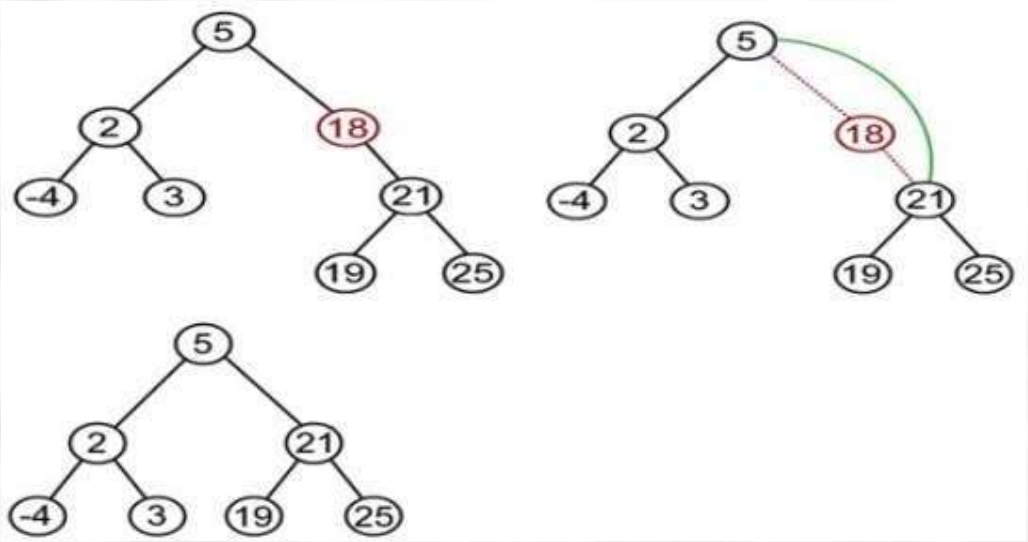


Deleting a node from the BST

1.The node to be deleted has one child

- In this case the child of the node to be deleted is appended to its parent node.

Suppose node to be deleted is 18



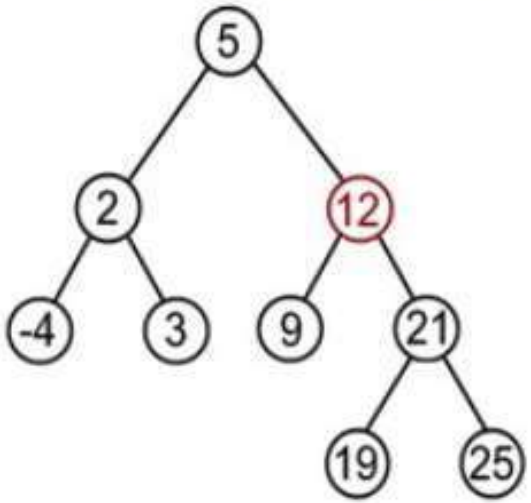
Deleting a node from the BST

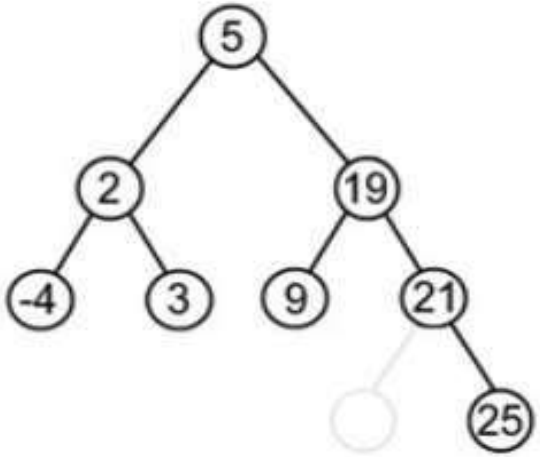
3. the node to be deleted has two children:

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.





Suppose node to deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

Threaded Binary Search Tree

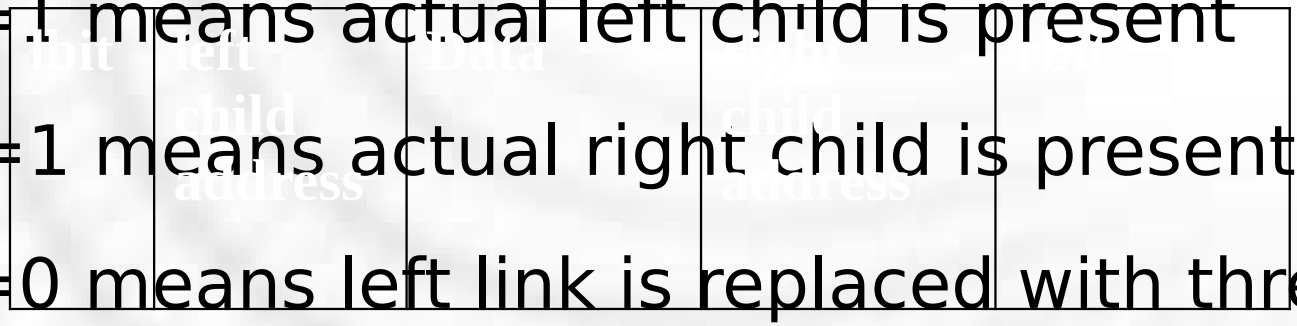
Need	**1 hour**	**2M**
Advantages of TBT	**2M**	
Disadvantages of TBT		
Converting Binary Tree to TBT	**4M**	
Implementation of TBT	**1 hour**	**4M**
Traversal Techniques of TBT	**4M**	

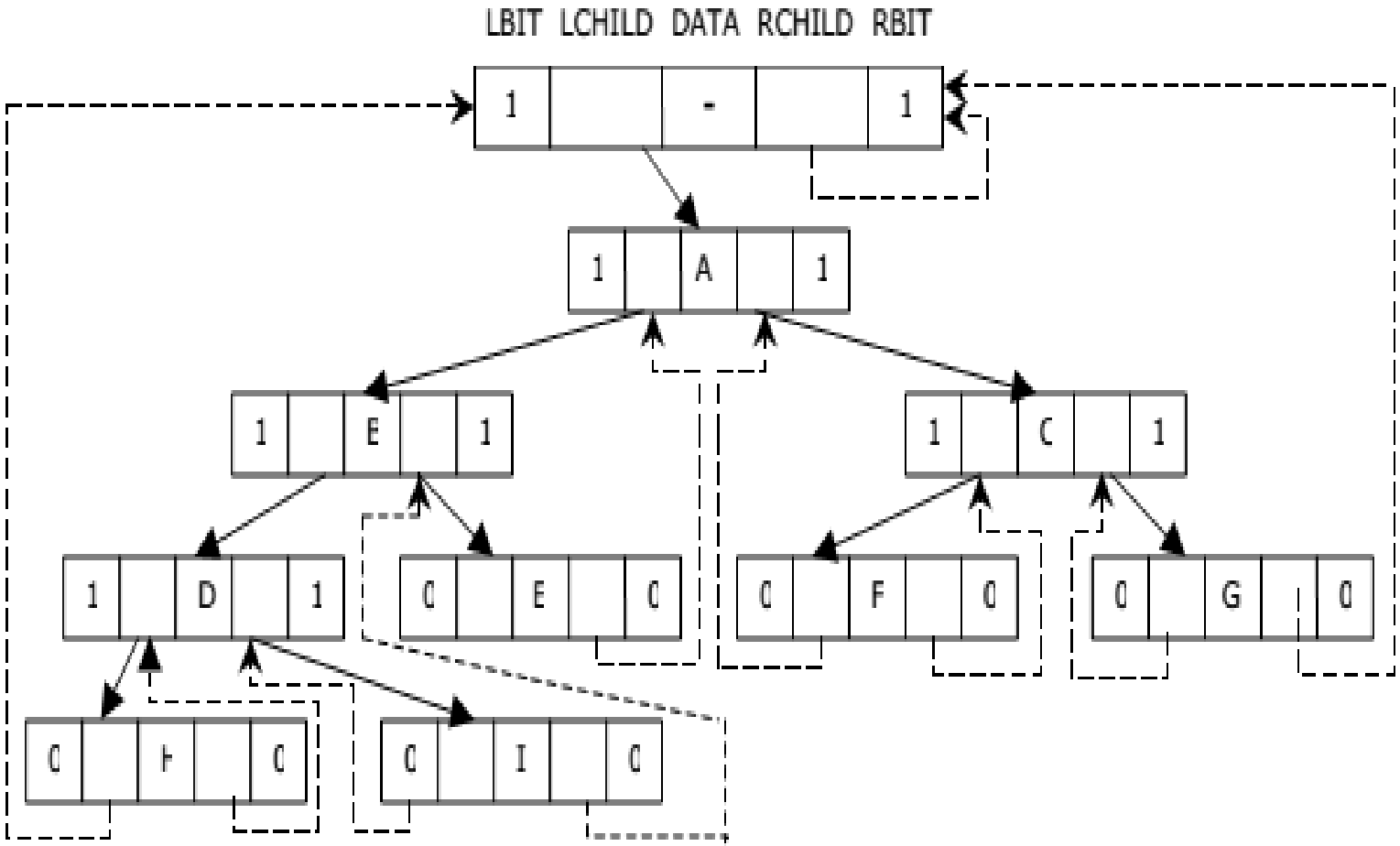
Why TBT?

- In a linked list representation of binary tree, there are more null links than actual pointers. These null links can be replaced by pointers called threads to other nodes.
- A left null link of a node is replaced with the address of it’s inorder predecessor and right null link is replaced with address of it’s inorder successor

Node structure of TBT

- In the memory representation of a tree node we must be able to distinguish between threads and normal pointers so two extra fields lbit and rbit are added.
- lbit=1 means actual left child is present
- rbit=1 means actual right child is present
- lbit=0 means left link is replaced with thread
- rbit=0 means right link is replaced with thread




LBIT LCHILD DATA RCHILD RBIT
1 - 1
1 A 1
1 E 1
1 C 1
1 D 1
0 E 0
0 F 0
0 G 0
0 H 0
0 I 0

Huffman Algorithm 1h

- Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.

- This is a method for the construction of minimum redundancy codes.

- Applicable to many forms of data transmission

- multimedia codecs such as JPEG and MP3

Huffman Algorithm

- 1951, David Huffman found the “most efficient method of representing numbers, letters, and other symbols using binary code”. Now standard method used for data compression.
- In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm. Initially 2 nodes are considered and their sum forms their parent node.
- When a new element is considered, it can be added to the tree.
- Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Huffman Algorithm

- Lets say you have a set of numbers and their frequency of use and want to create a huffman encoding for them

Value	Frequency
1	5
2	7
3	10
4	15
5	20
6	45

Huffman Algorithm

- Creating a Huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:

- The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

12:*
15:4
20:5
45:6

Huffman Algorithm

•

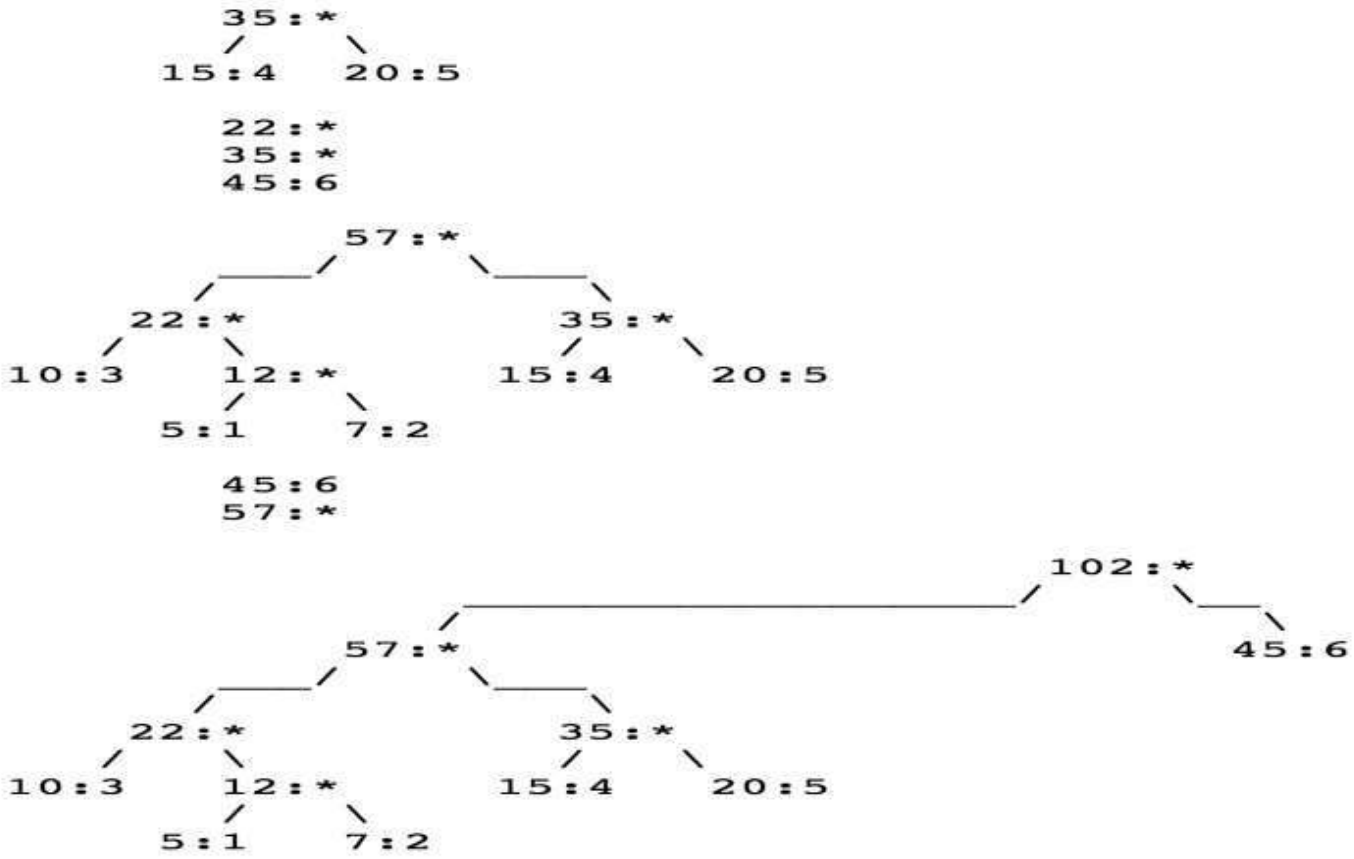
/ \
10:3 12:*
/ \
5:1 7:2

- The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

20:5
22: *
45:6

Huffman Algorithm

You repeat until there is only one element left in the list.



Now the list is just one element containing 102:*, you are done.

Expression Tree 1h

- **Definition:** An expression tree is a **representation of expressions arranged in a tree-like data structure**. In other words, it is a tree with leaves as operands of the expression and nodes contain the operators.
- **Constructing an Expression tree**

Algorithm(scan postfix expression from left to right)

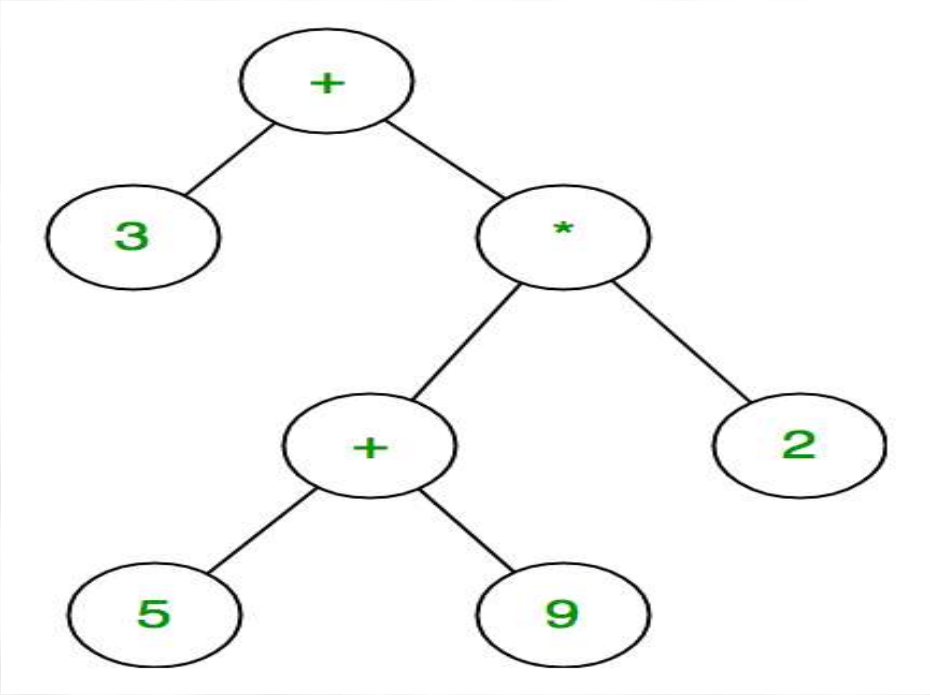
1.Get one symbol at a time

2.If symbol is operand then create a node and push pointer onto stack

3.If symbol is operator then pop pointer to two trees T1 and T2

Example of Expression Tree

- **Expression tree for 3 + ((5+9)*2) would be:**



References

- https://en.wikipedia.org/wiki/Binary_search_tree
- https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_bst.pdf
- https://www.cs.usfca.edu/~galles/visualization/BST.html
- https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf

Thank You