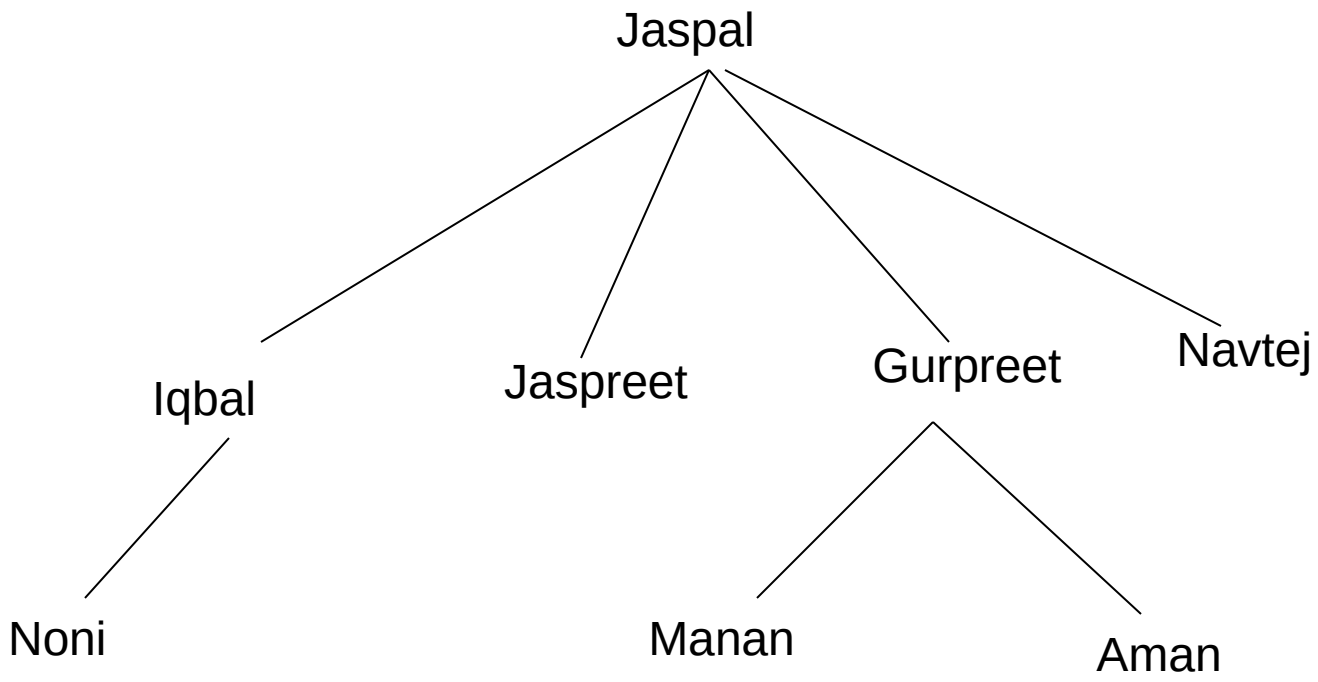
Trees

- **Objectives:**
  - Describe different types of trees.
  - Describe how a binary tree can be represented using an array.
  - Describe how a binary tree can be represented using an linked lists.
  - Implement various operations on binary tree.

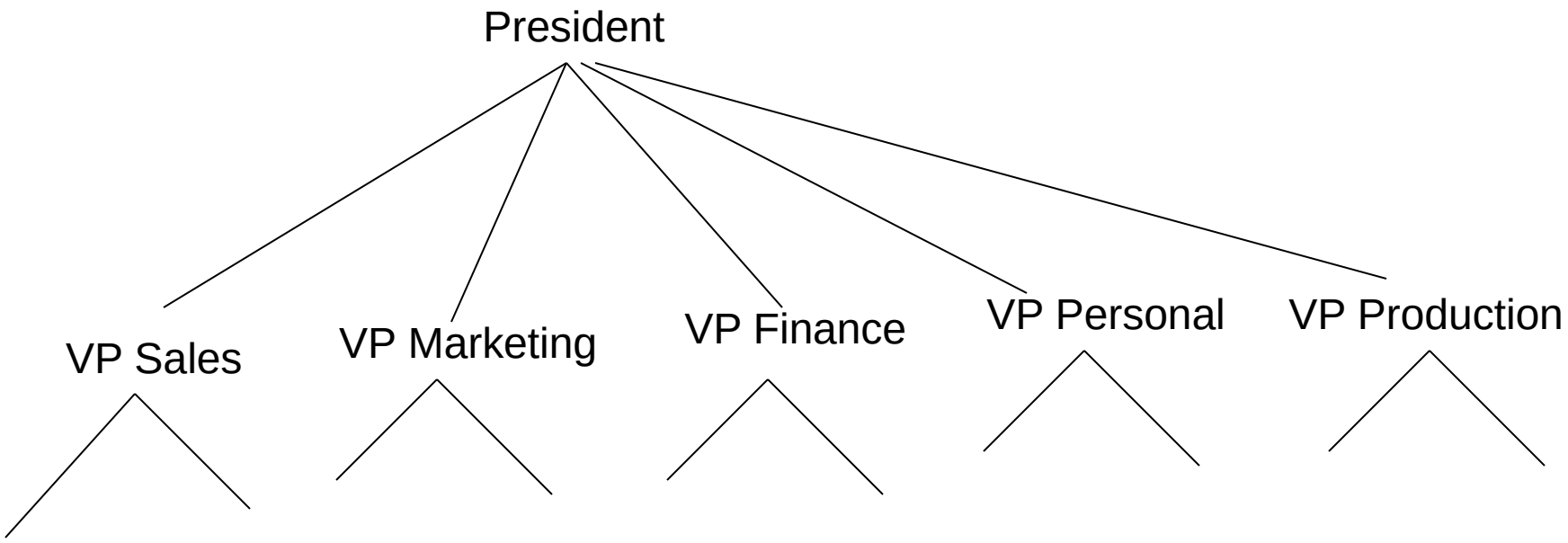
Introduction

- Arrays, linked list ,stacks and queues are used to represent the linear and tabular data.
- These structure are not suitable for representing hierarchical data.
- In hierarchical data we have
  - ancestor, descendant
  - Superior, subordinate etc.

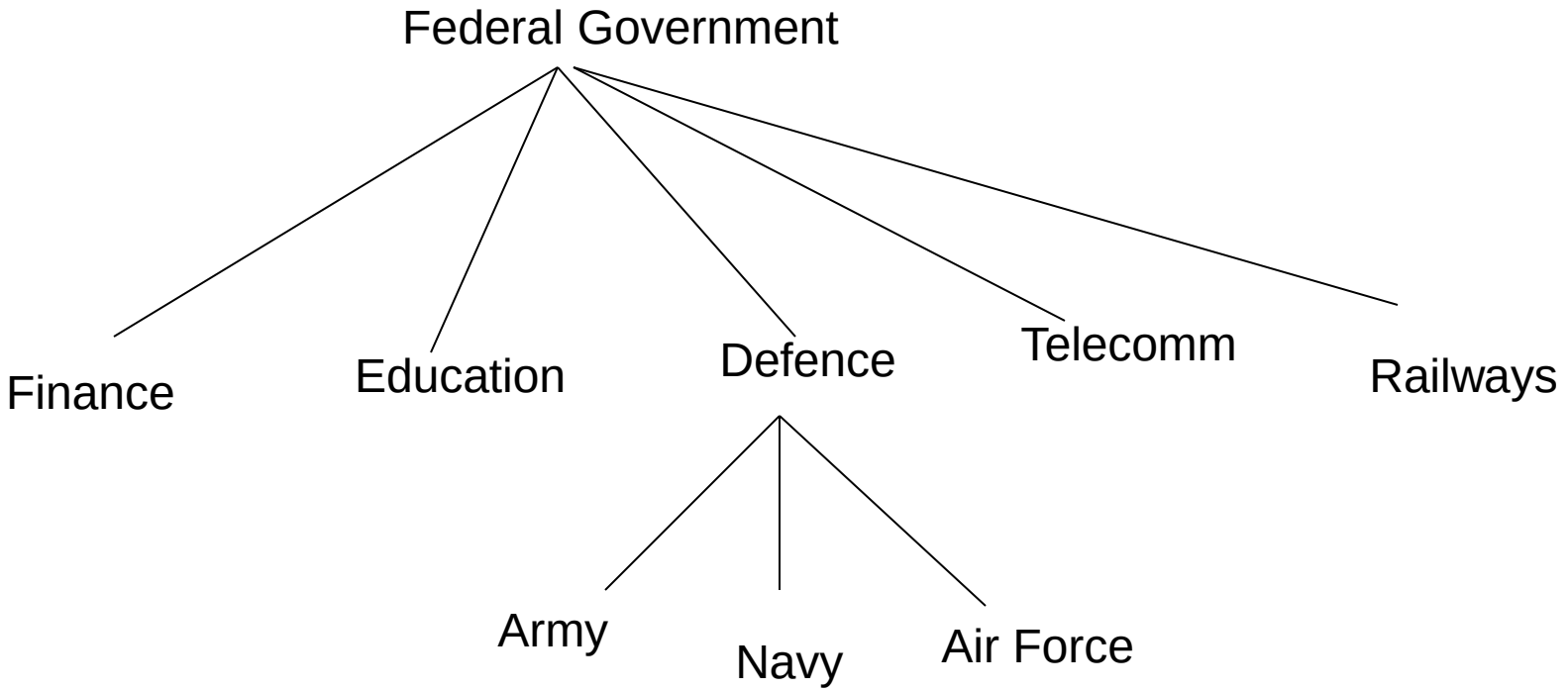
Family structure



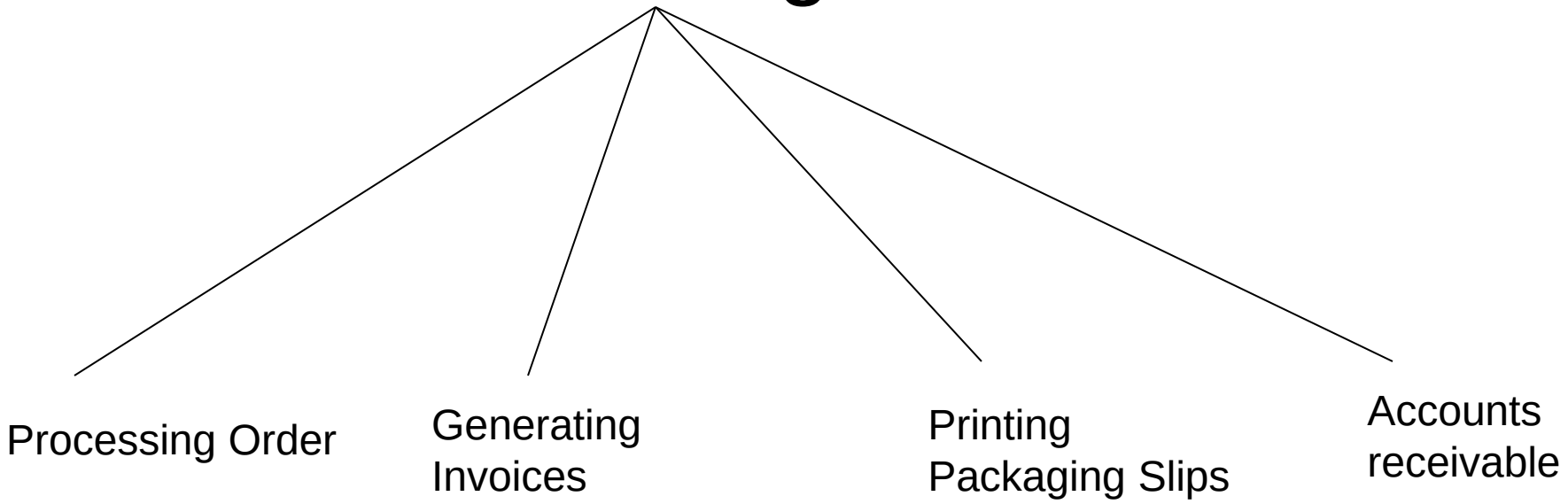
Business Corporate Structure



Federal government Structure



Modular structure of a computer Program



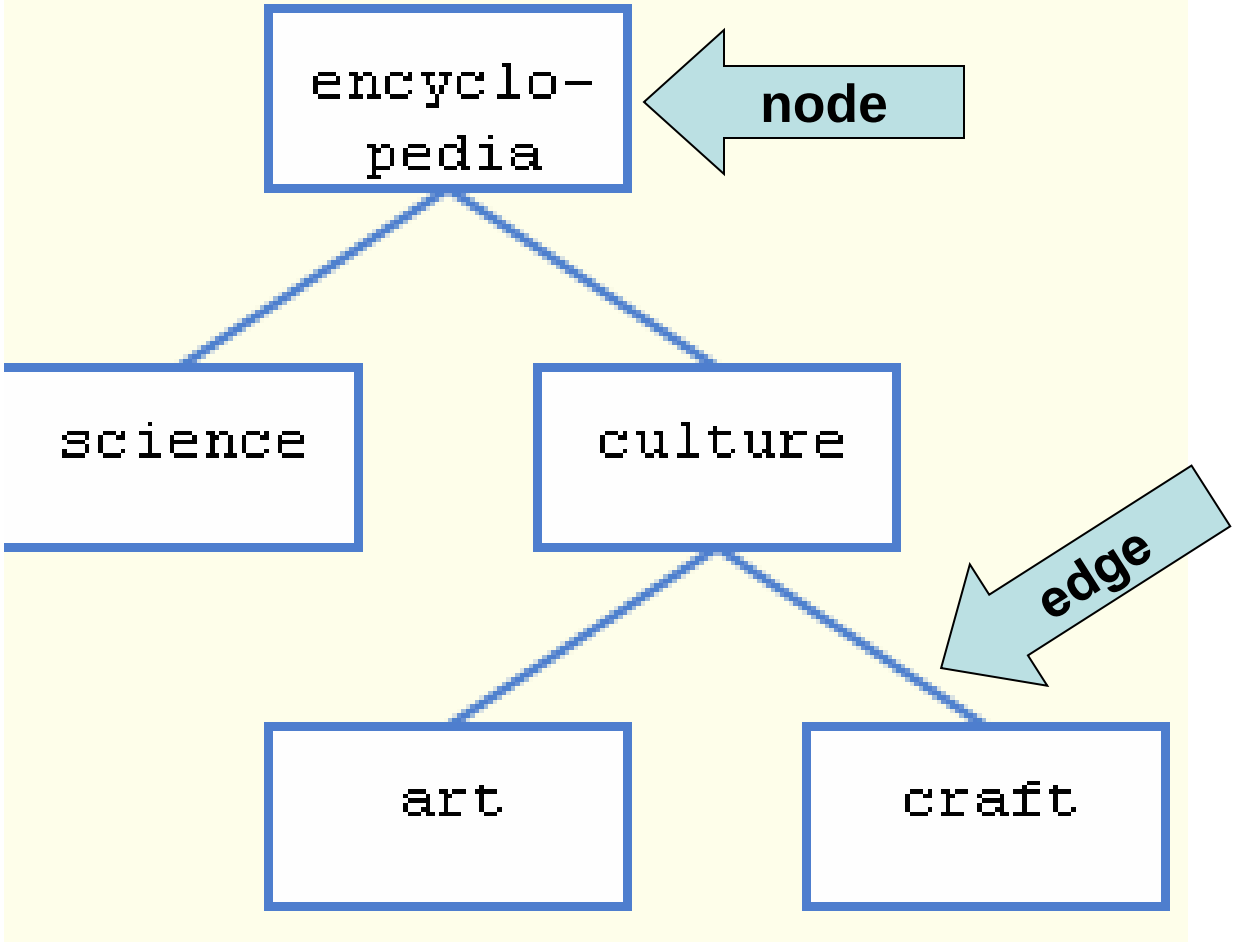
TREE DEFINED

A tree T is a finite non empty set of elements. One of these elements is called the root, and the remaining elements, if any, are portioned into trees, which are called the sub trees of T.

Introduction to Tree

- Fundamental data storage structures used in programming.
- Combines advantages of an ordered array and a linked list.
- Searching as fast as in ordered array.
- Insertion and deletion as fast as in linked list.

Tree (example)



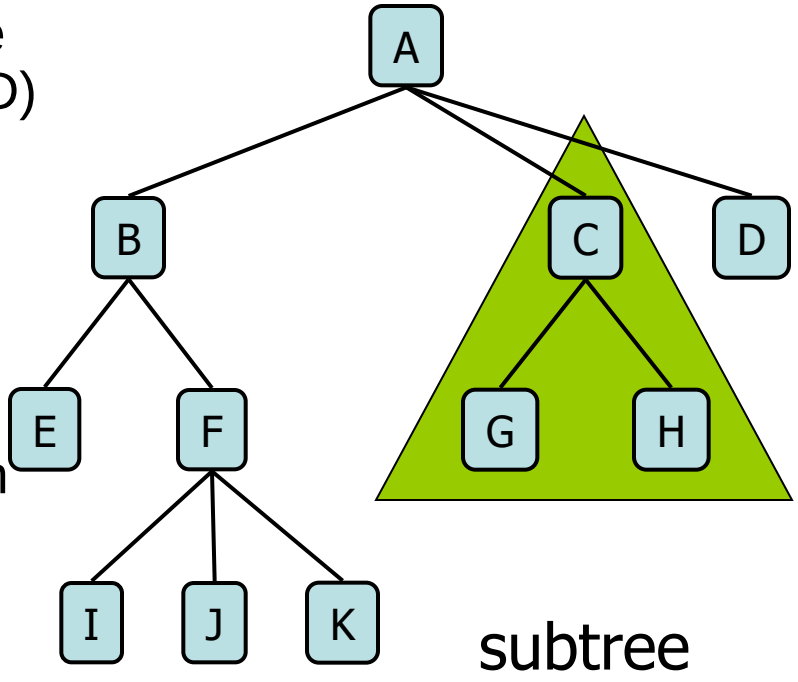
Tree characteristics

- Consists of *nodes* connected by *edges*.
- Nodes often represent entities (complex objects) such as people, car parts etc.
- Edges between the nodes represent the way the nodes are related.
- Its easy for a program to get from one node to another if there is a line connecting them.
- The only way to get from node to node is to follow a path along the edges.

Tree Terminology

- **Root:** node without parent (A)
- **Internal node:** node with at least one child (A, B, C, F)
- **External node** (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Depth of a node**: number of ancestors
- **Height of a tree:** maximum depth of any node (3)
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- **Degree of an element:** no. of children it has

- **Subtree:** tree consisting of a node and its descendants



Tree Terminology

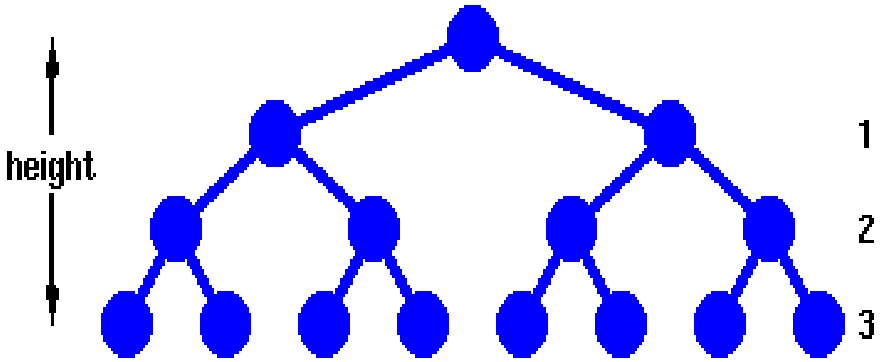
- **Path:** Traversal from node to node along the edges results in a sequence called path.
- **Root:** Node at the top of the tree.
- **Parent:** Any node, except root has exactly one edge running upward to another node. The node above it is called parent.
- **Child:** Any node may have one or more lines running downward to other nodes. Nodes below are children.
- **Leaf:** A node that has no children.
- **Subtree:** Any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.

Tree Terminology

- **Visiting:** A node is visited when program control arrives at the node, usually for processing.
- **Traversing:** To traverse a tree means to visit all the nodes in some specified order.
- **Levels:** The level of a particular node refers to how many generations the node is from the root. Root is assumed to be level 0.
- **Keys:** Key value is used to search for the item or perform other operations on it.

Binary Trees

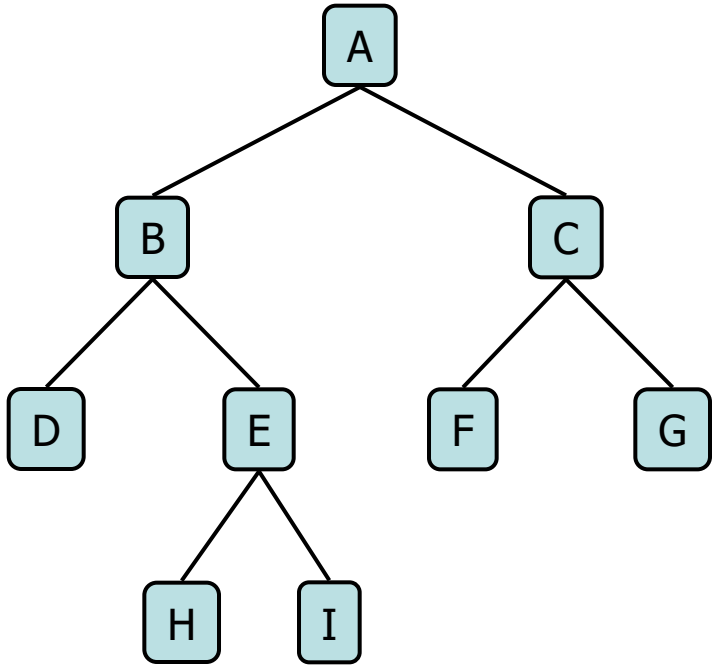
- Every node in a binary tree can have at most two children.
- The two children of each node are called the left child and right child corresponding to their positions.
- A node can have only a left child or only a right child or it can have no children at all.



Binary Trees

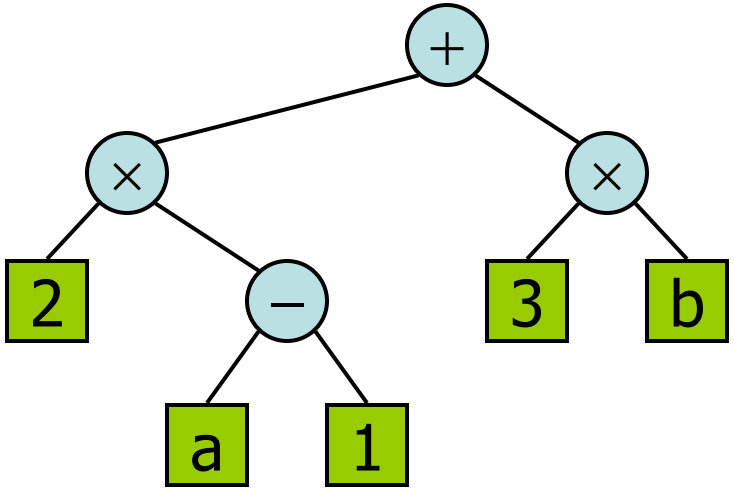
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for **proper** binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$

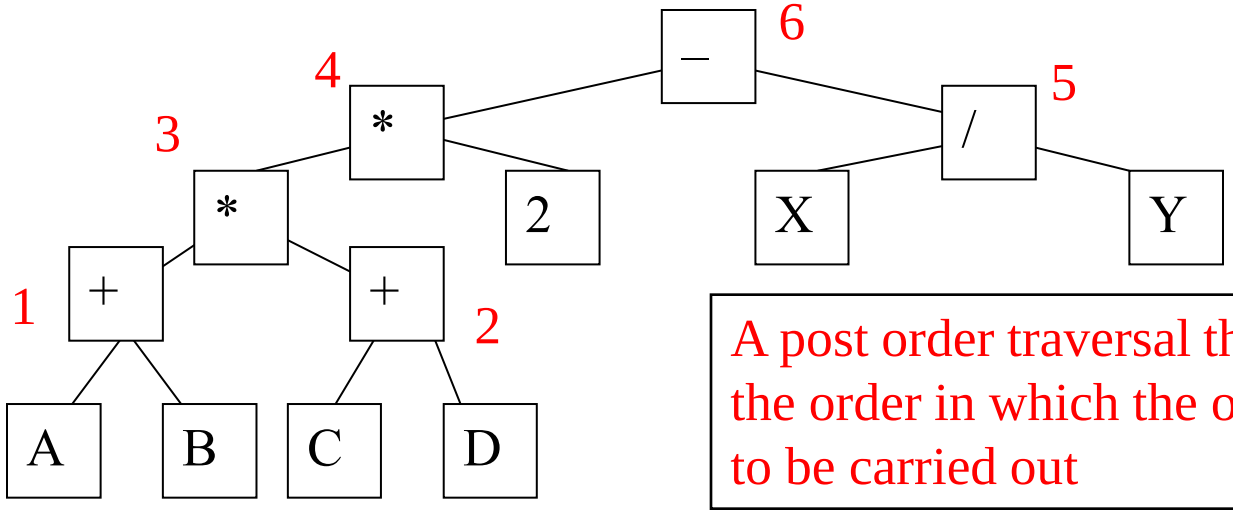


Compiling arithmetic expressions

- We can represent an arithmetic expression such as

  (A + B) * (C + D) * 2 – X / Y

  as a binary tree, in which the leaves are constants or variables and the nodes are operations:



A post order traversal then gives us the order in which the operations have to be carried out

Some Types of Binary Trees (continued)

FIGURE 8.4
Expression Tree

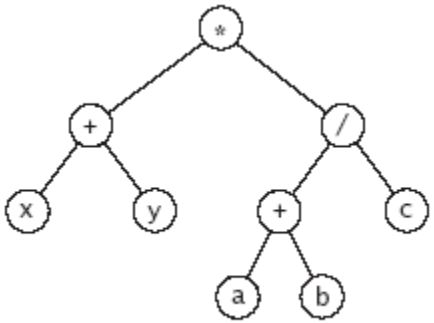
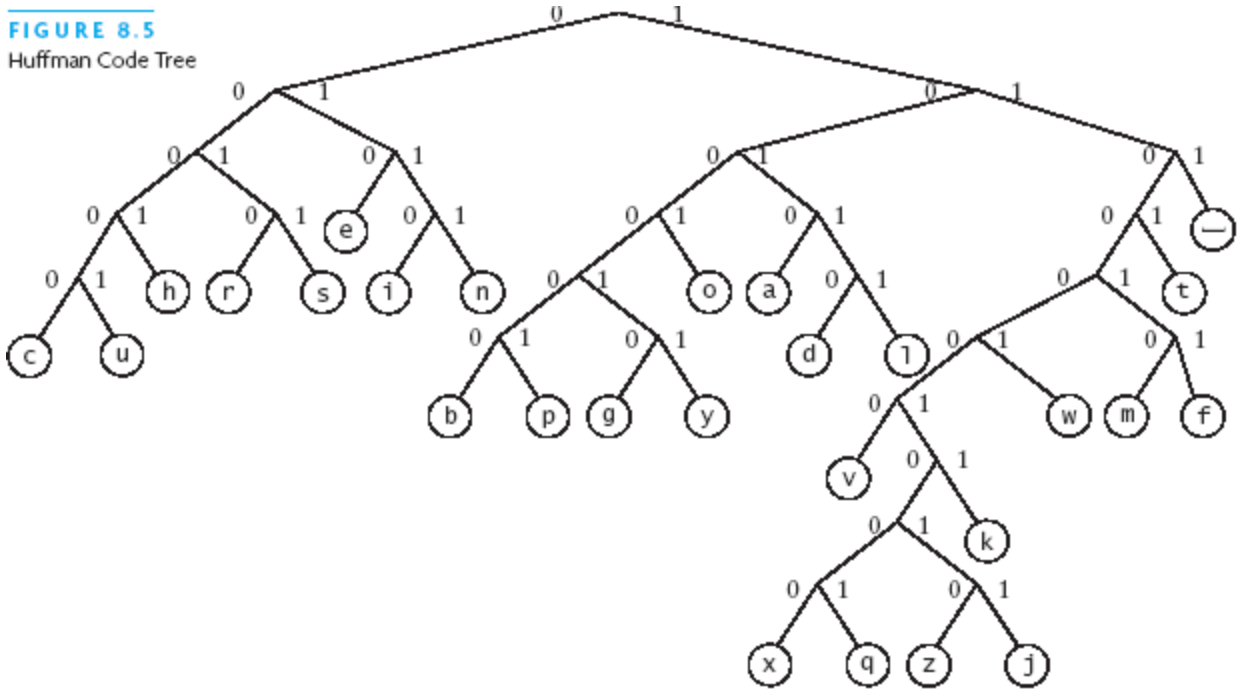
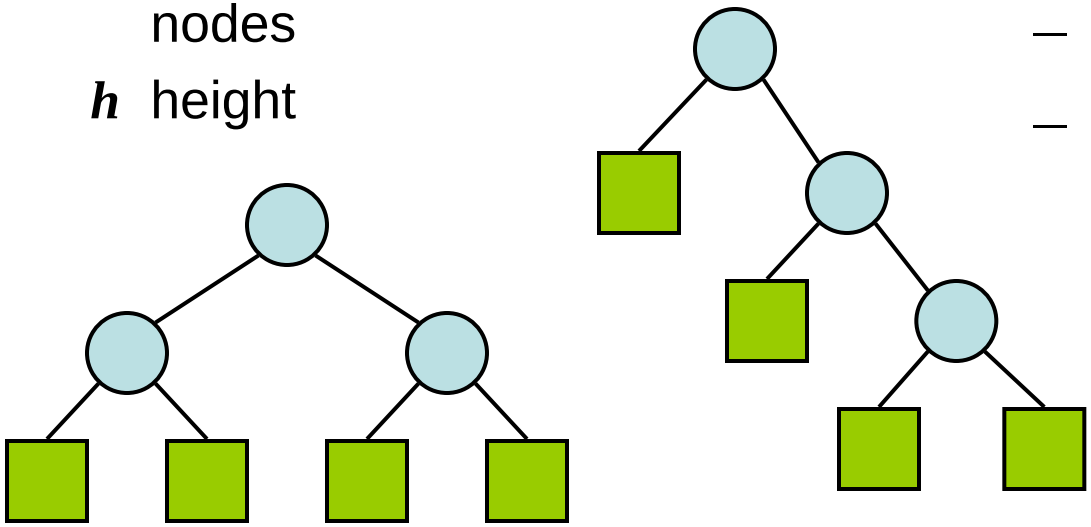


FIGURE 8.5
Huffman Code Tree



Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$

Binary Search Trees

- Binary Search Trees:

A binary search tree T is a binary tree that may be empty. A non empty binary search tree satisfies the following properties:

1. Every element has a key (or value) and no two elements have the same key i.e. all keys are unique.
2. The keys, if any, in the left subtree of the root are smaller than the key in the node.
3. The keys, if any, in the right subtree of the root are larger than the key in the node.
4. The left and right subtrees of the root are also binary search trees.

BST

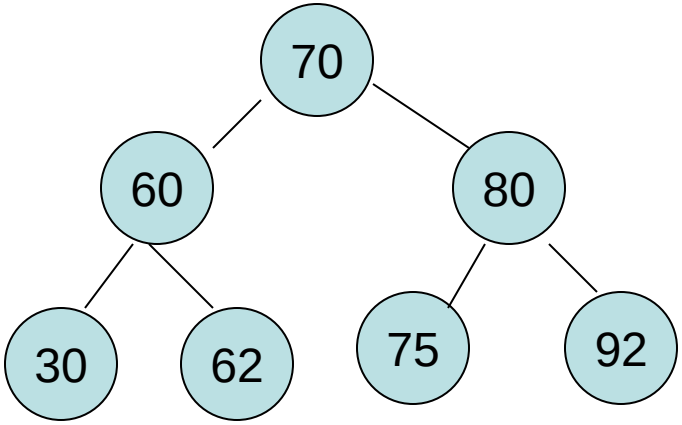


Fig.(a)

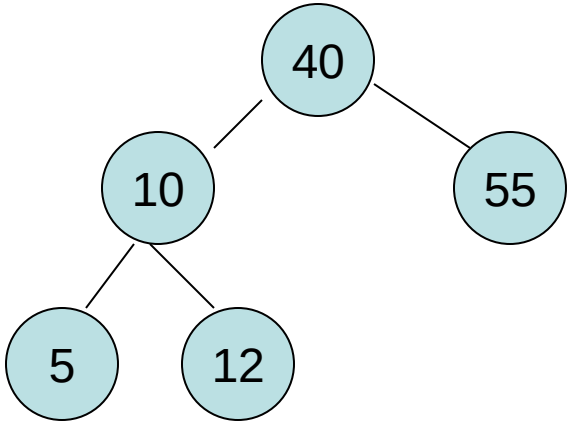


Fig. (b)

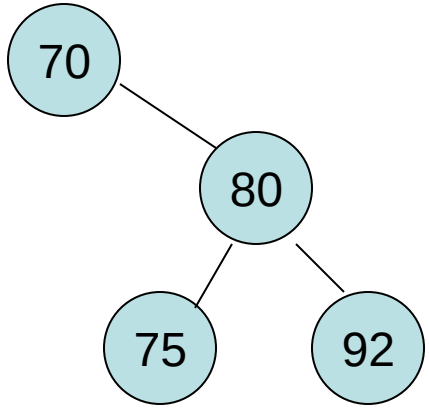


Fig. ©

Binary search trees

Representation of Binary & BST

- Array Representation
- Linked list representation

Array Representation

- In this representation the binary tree is represented by storing each element at the array position corresponding to the number assigned to each number (nodes)
- In this representation, a binary tree of height h requires an array of size (2^h-1), in the worst case

Array Representation

- For simplicity , I am considering the array is indexed with an index set beginning from 1 not 0.

1	2	3	4	5	6	7
70	60	80	30	62	75	92

Array representation of BST in fig (a)

1	2	3	4	5	6	7
40	10	55	5	12		

Array representation of BST in fig (b)

1	2	3	4	5	6	7
70		80			75	92

Array representation of BST in fig (c)

BST

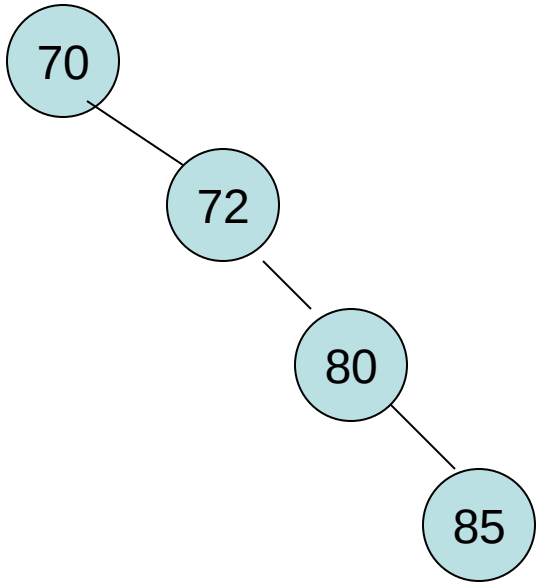


Fig. (d)

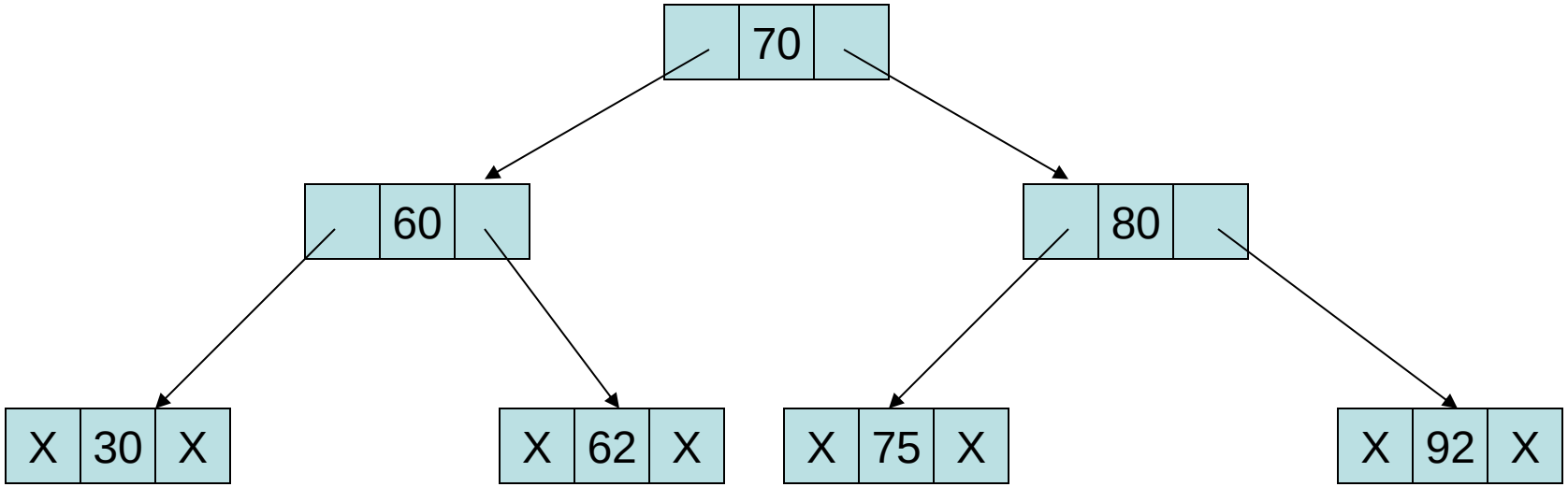
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
70		72				80								85

Disadvantages of array representation

- This scheme of representation is quite wasteful of space when binary tree is skewed or number of elements are small as compared to its height.
- Array representation is useful only when the binary tree is full or complete.

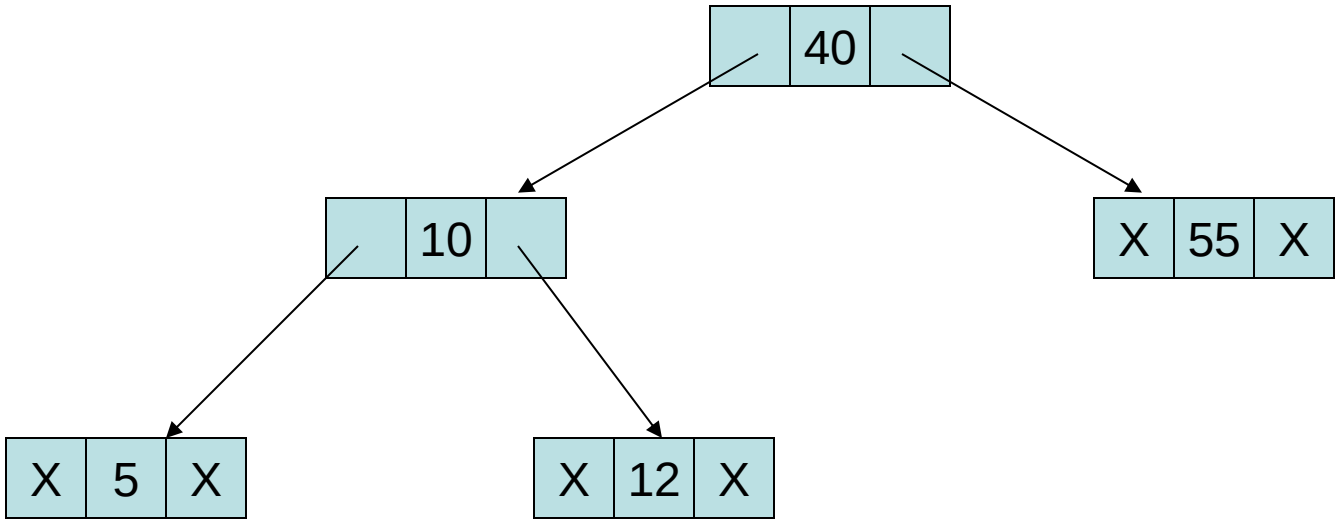
Linked representation

- The most popular and practical way of representing a binary tree is using links (pointers).



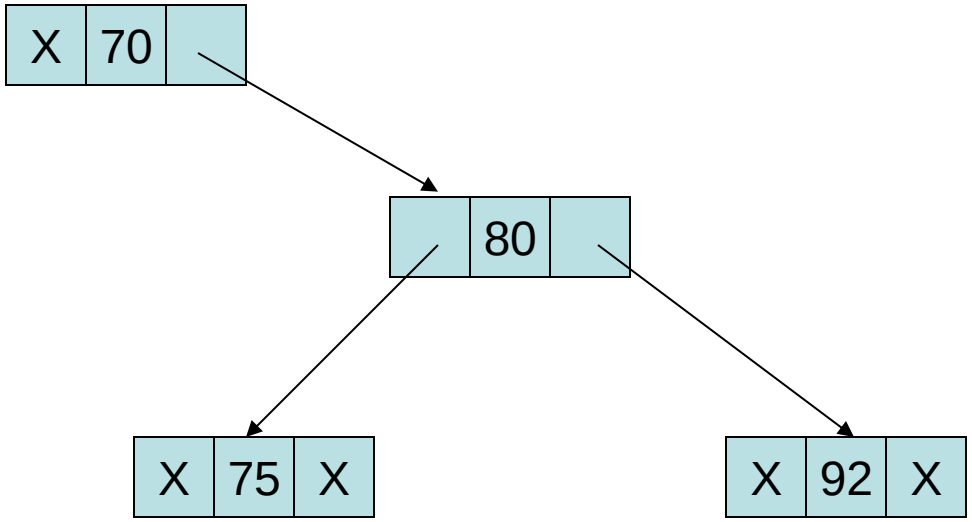
Linked representation of binary search tree of figure (a)

Linked representation



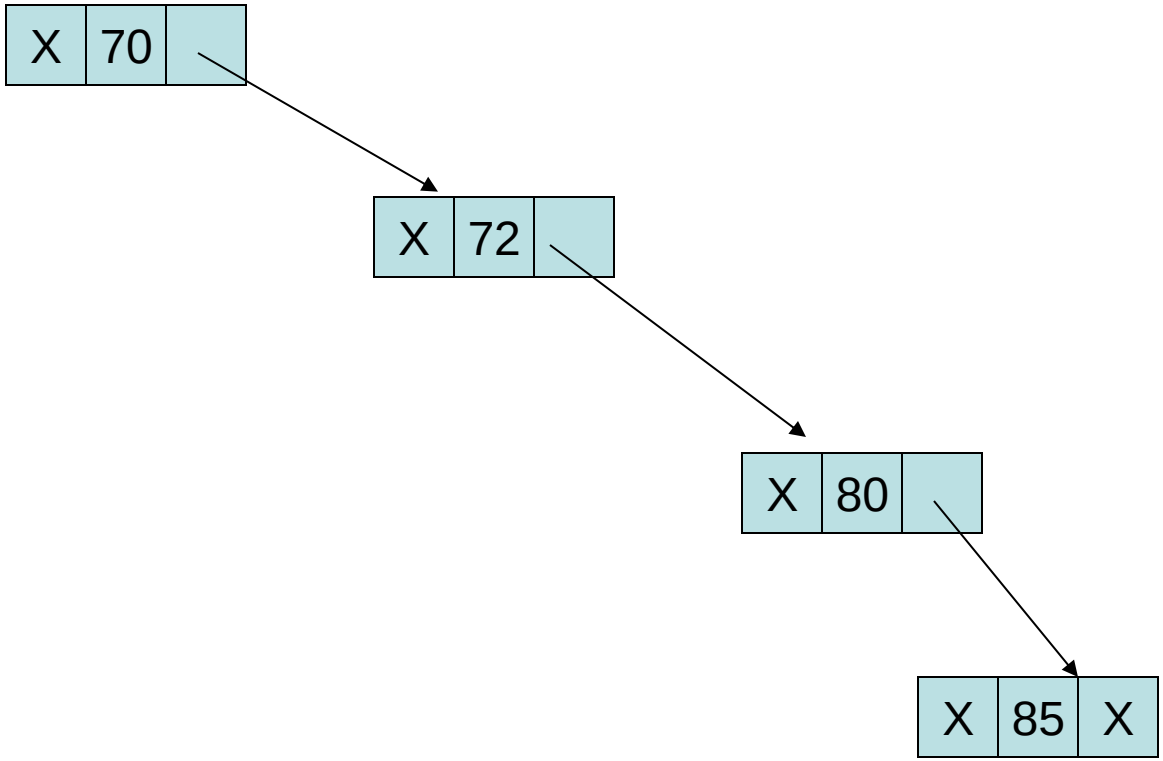
Linked representation of binary search tree of figure (b)

Linked representation



Linked representation of binary search tree of figure (c)

Linked representation



Linked Representation

- In linked representation, each element is represented by a node that has exactly two link fields.
- Field names are left and right.
- Each node has data field called info.

The node structure is defined as:

```
struct nodetype
{
        struct nodetype *left;
        int info;
        struct nodetype *right;
};
```

Implementation of binary tree will consider following declarartion

```
typedef struct nodetype
{
        struct nodetype *left;
        int info;
        struct nodetype *right;
}BST;
BST root;
```

Here , I have defined a new datatype and given it name BST, then I have declared a pointer variable root to represent the binary (search) tree.

Common operation on binary and binary search trees

Some of the operations that are commonly performed on binary as well as binary search trees:

- Creating an empty tree.
- Traverse it
- Determine its height.
- Determine the number of elements in it.
- Determine the no of internal nodes i.e. non leaf nodes.
- Determine the no of external nodes i.e. leaf nodes.
- Determine its mirror image.
- Remove it from memory.

Operations that are performed on only with binary search trees

1. Insert a new node.
2. Search an element.
3. Find the smallest element.
4. Find the largest element.
5. Delete a node.

1. Creating an empty Binary (Search) Trees

```
void createtree(BST **tree)
{
        *tree=NULL;
}
```

2. Traverse a binary tree

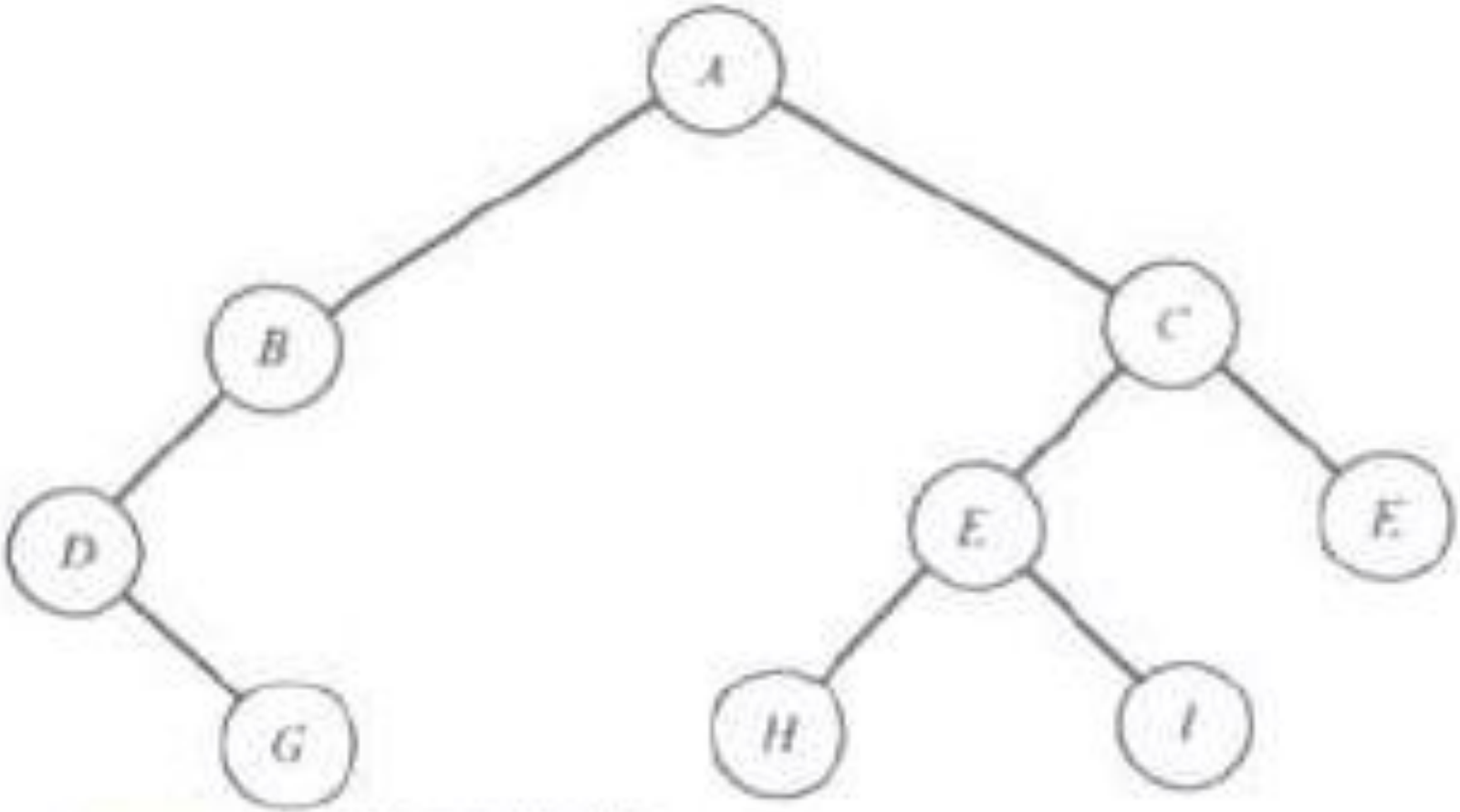
- Preorder (depth first order)
  1. Visit the root
  2. Traverse the left subtree in preorder.
  3. Traverse the right subtree in preorder.
- Inorder (Symmetric order)
  1. Traverse the left subtree in inorder.
  2. Visit the root.
  3. Traverse the right subtree in inorder.

Traverse a binary tree

- Postorder

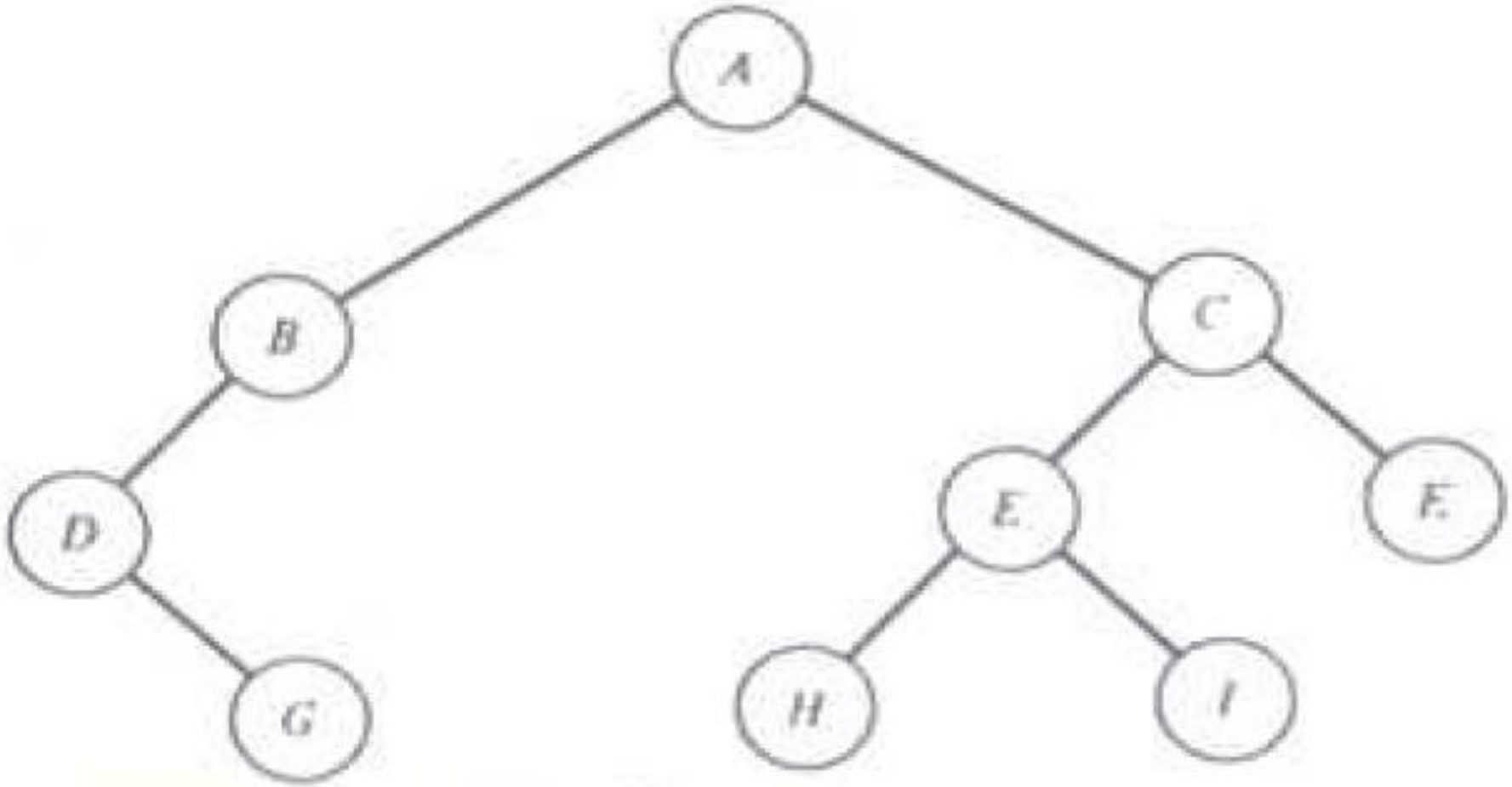
1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Traverse a binary tree



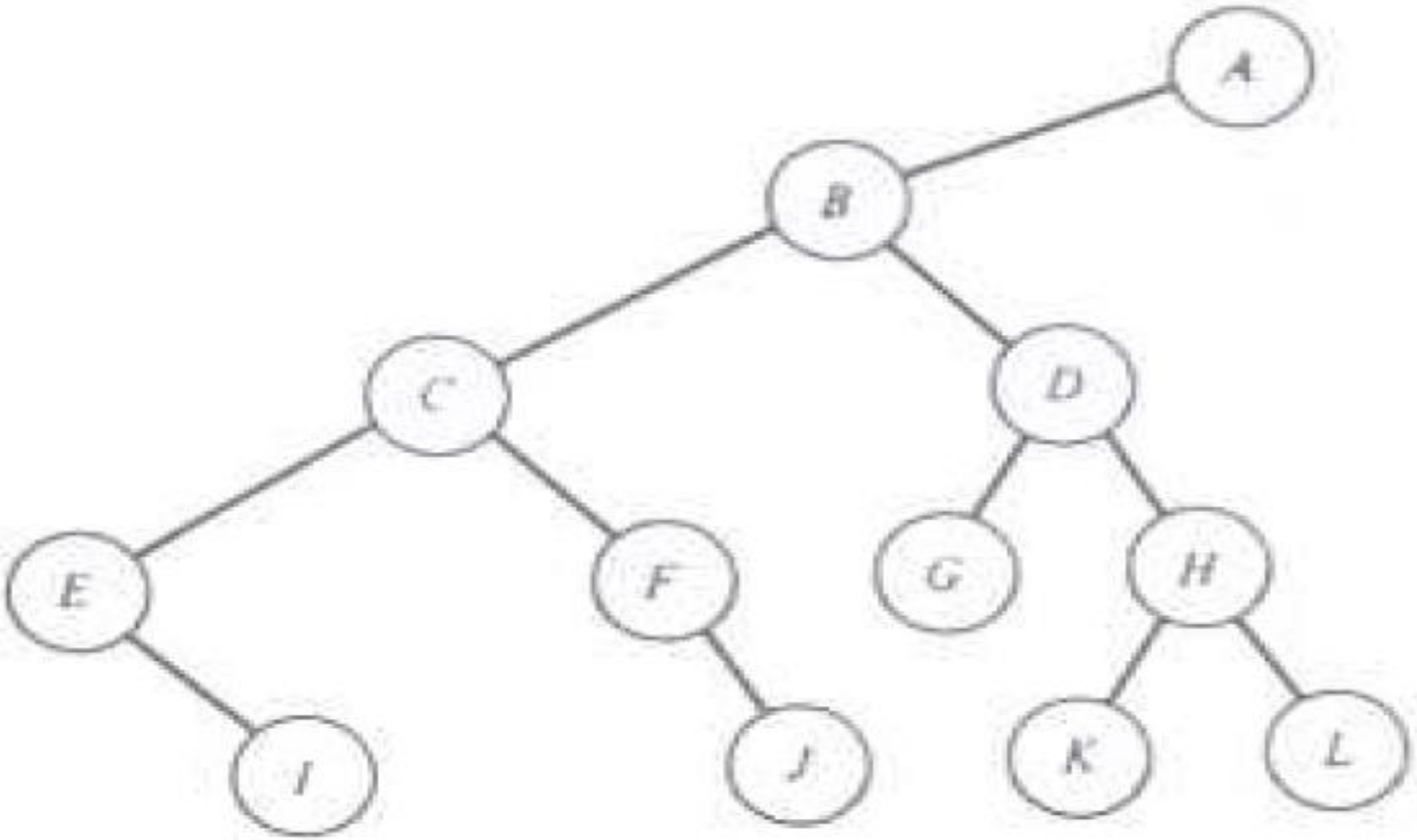
Preorder: *ABDGCEHIF*
Inorder: *DGBAHEICF*
Postorder: *GDBHIEFCA*

Traverse a binary tree



Preorder: *ABDGCEHIF*
Inorder: *DGBAHEICF*
Postorder: *GDBHIEFCA*

Traverse a binary tree



Preorder ABCEIFJDGHKL
Inorder EICFJBGDKHLA
Postorder IEJFCGKLHDBA

Traversing of Binary (search) tree

1. **Pre-order Traversal:**

```
void pre(BST *tree)
{
   if(tree!=NULL)
   {
            printf(“%d”,tree->info);
            pre(tree->left);
            pre(tree->right);
   }
}
```

IN-Order Traversal

```
void inorder(BST *tree)
{
    if(tree!=NULL)
    {
        inorder(tree->left);
        printf("%d",tree->info);
        inorder(tree->right);
    }
}
```

Post order traversal

```
void postorder(BST *tree)
{
        if(tree!=NULL)
        {
        postorder(tree->left);
        postorder(tree->right);
        printf(“%d”,tree->info);

        }
}
```

Example

- Pre-order: A B D E G H C F
- In order: D B G E H A C F

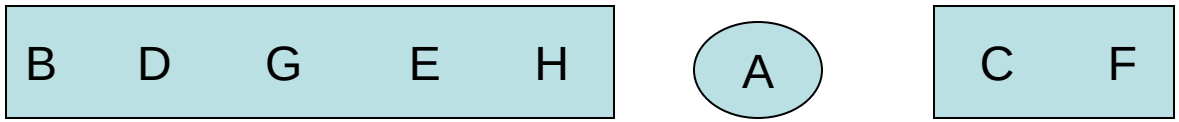
- Draw the binary tree T.

Example

Pre- Order:

Root | Left subtree L_{TA} | Right subtree R_{TA}

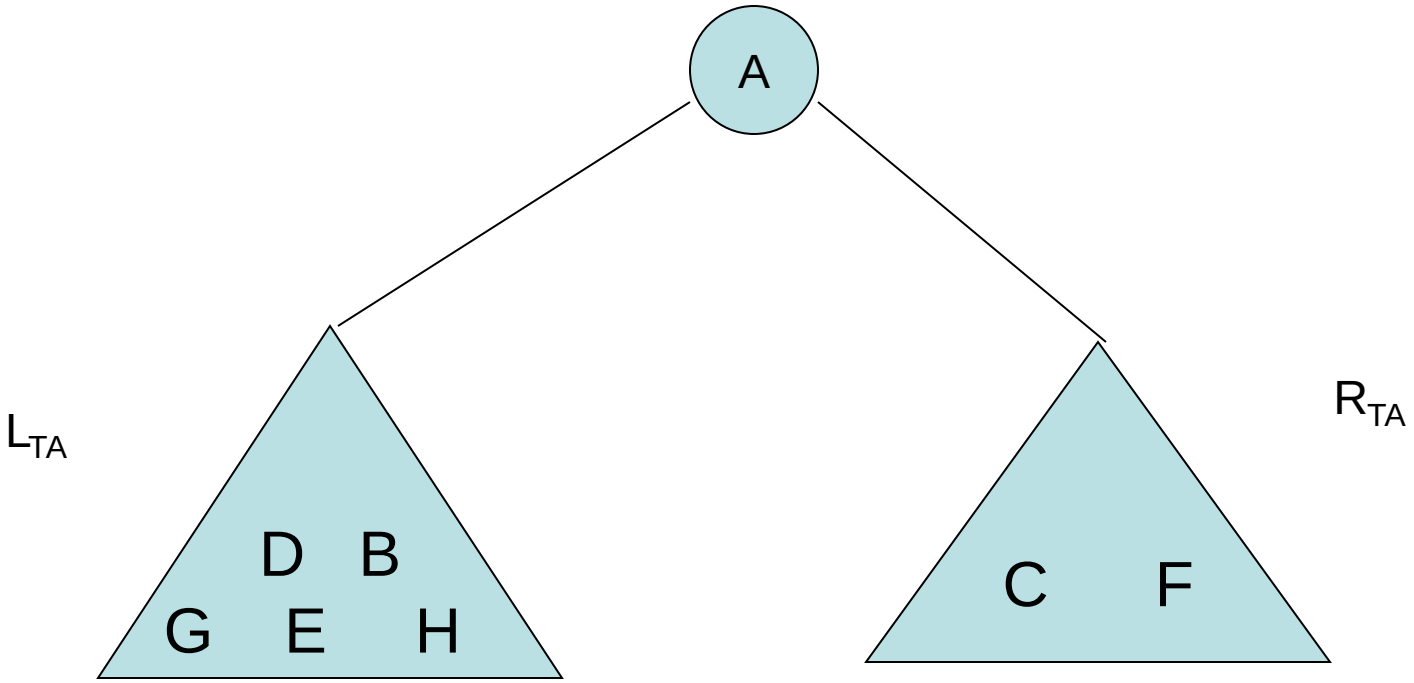
***In*- Order:**



Left subtree L_{TA} | Root | Right subtree R_{TA}

Example:

Partial Tree

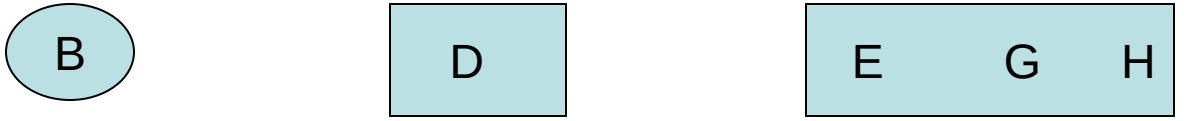


Pre- Order: B D E G H

In- Order: D B G E H

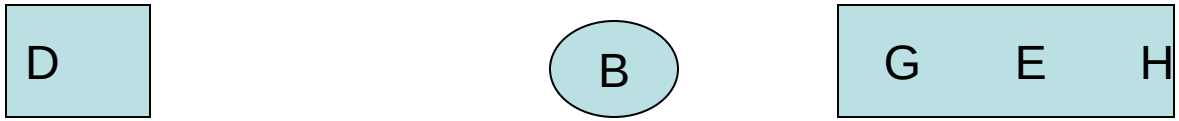
Example

Pre- Order:



Root | Left subtree L_{TB} | Right subtree R_{TB}

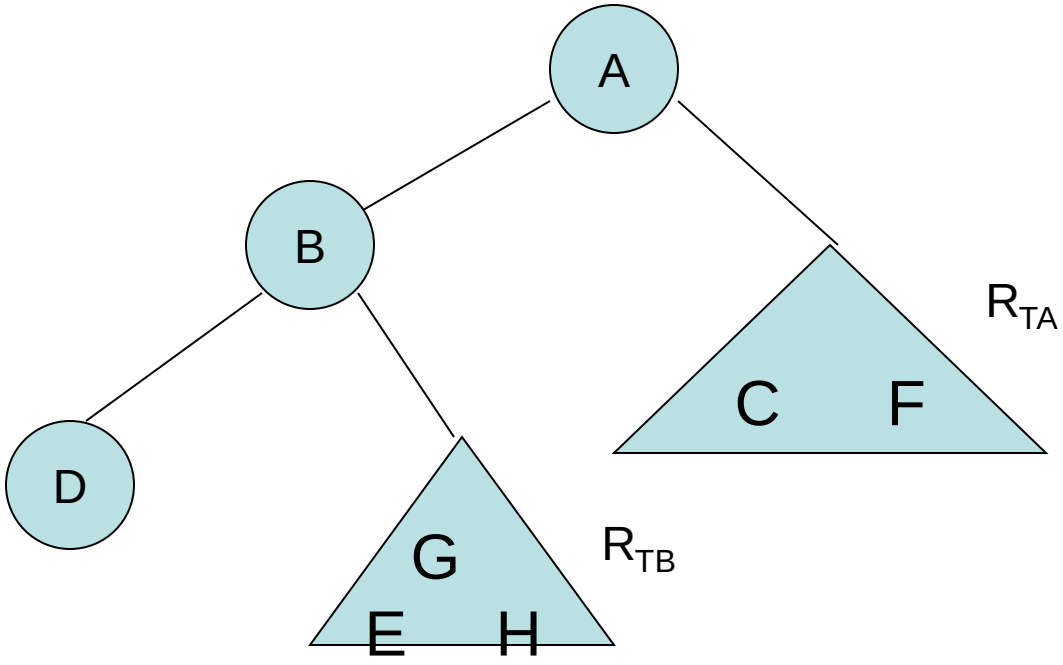
***In*- Order:**



Left subtree L_{TB} | Root | Right subtree R_{TB}

Example:

Partial Tree

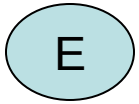


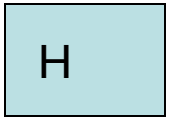
Pre- Order: E G H

In- Order: G E H

Example

Pre- Order:



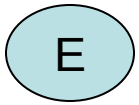


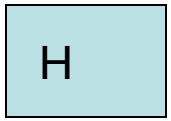
Root

Left subtree L_{TE}

Right subtree R_{TE}

***In*- Order:**





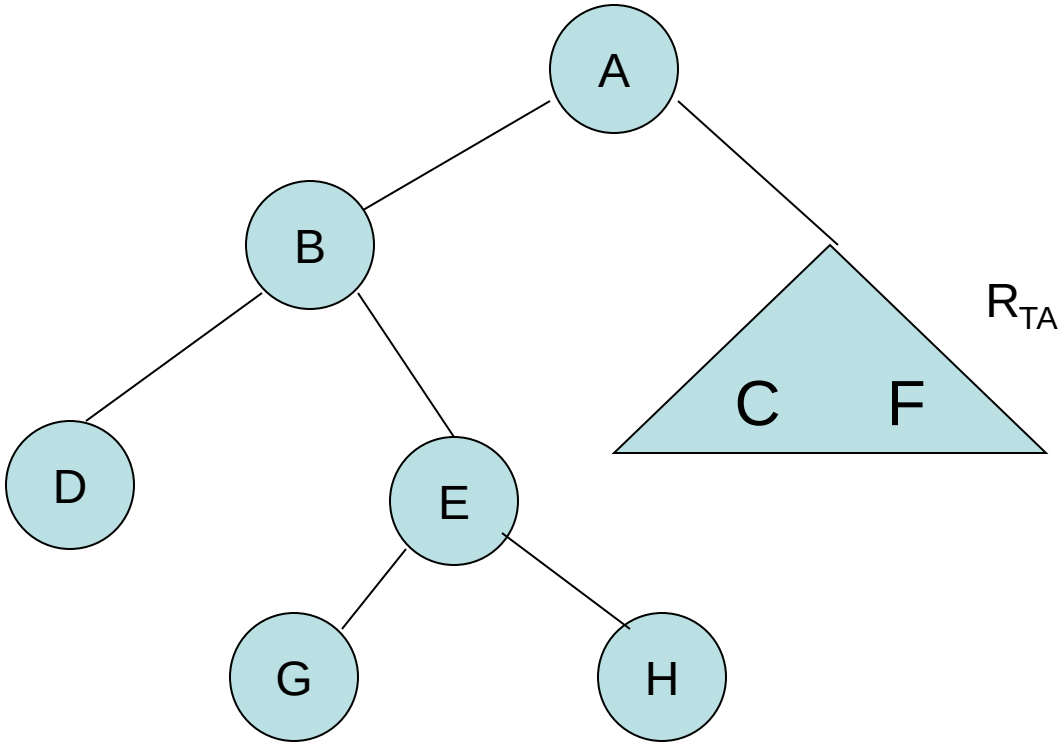
Left subtree L_{TE}

Root

Right subtree R_{TE}

Example:

Partial Tree



Pre- Order: C F

In- Order: C F

Example

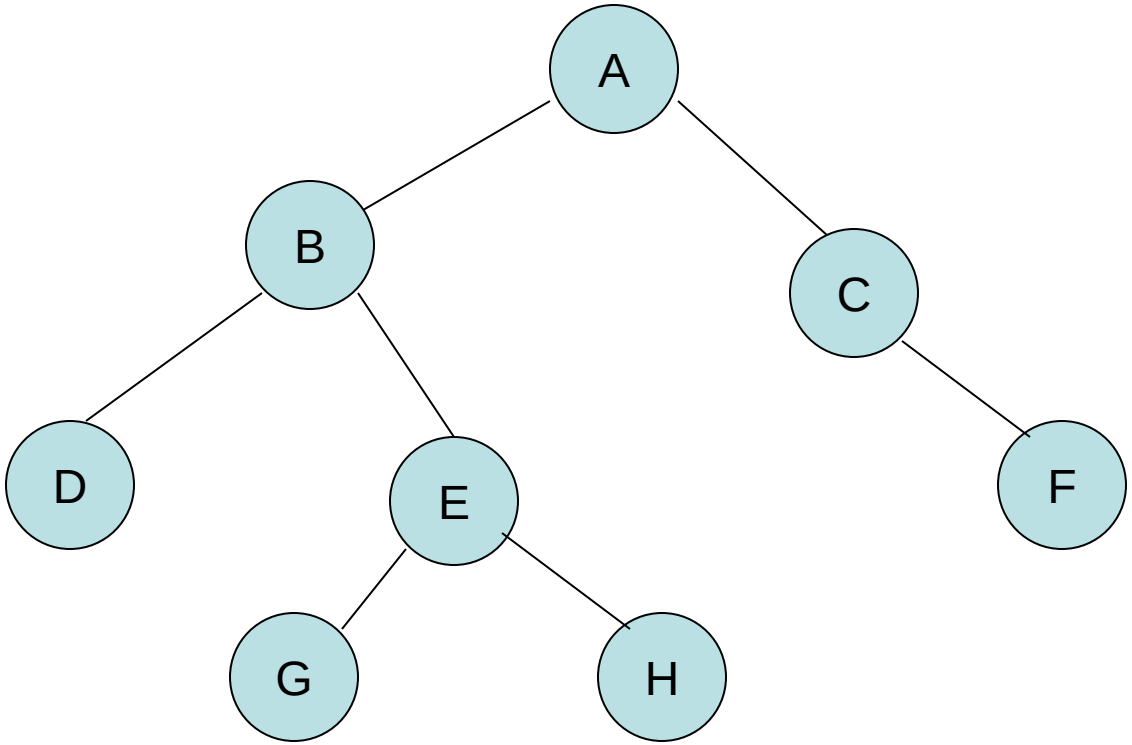
Pre- Order:

Root | Left subtree L_{TC} | Right subtree R_{TC}

***In*- Order:**

Left subtree L_{TC} | Root | Right subtree R_{TC}

Example:



Determining the height of the binary tree

- We find the height of the left sub tree and right sub tree of given node.
- Height of the binary tree at a given node will be equal to maximum height of the left and right sub tree plus 1.
- Program in the next slide shows the implementation of various steps required.

Determining the height of the binary tree

```
Int height(BST *tree)
{
    int lheight,rheight;
    if(tree==NULL)
    {
        return 0;
    }
    else
    {
        lheight=height(tree->left);
        rheight=height(tree->right);
        if(lheight>rheight)
            return ++lheight;
        else
            return ++rheight;
    }
}
```

Determining number of nodes/elements

- Any of the traversal scheme can be used to determine the number of element.
- The approach we use that is
  - Number of elements in the tree is equal to number of nodes in left sub tree plus number of nodes in right sub tree plus one.

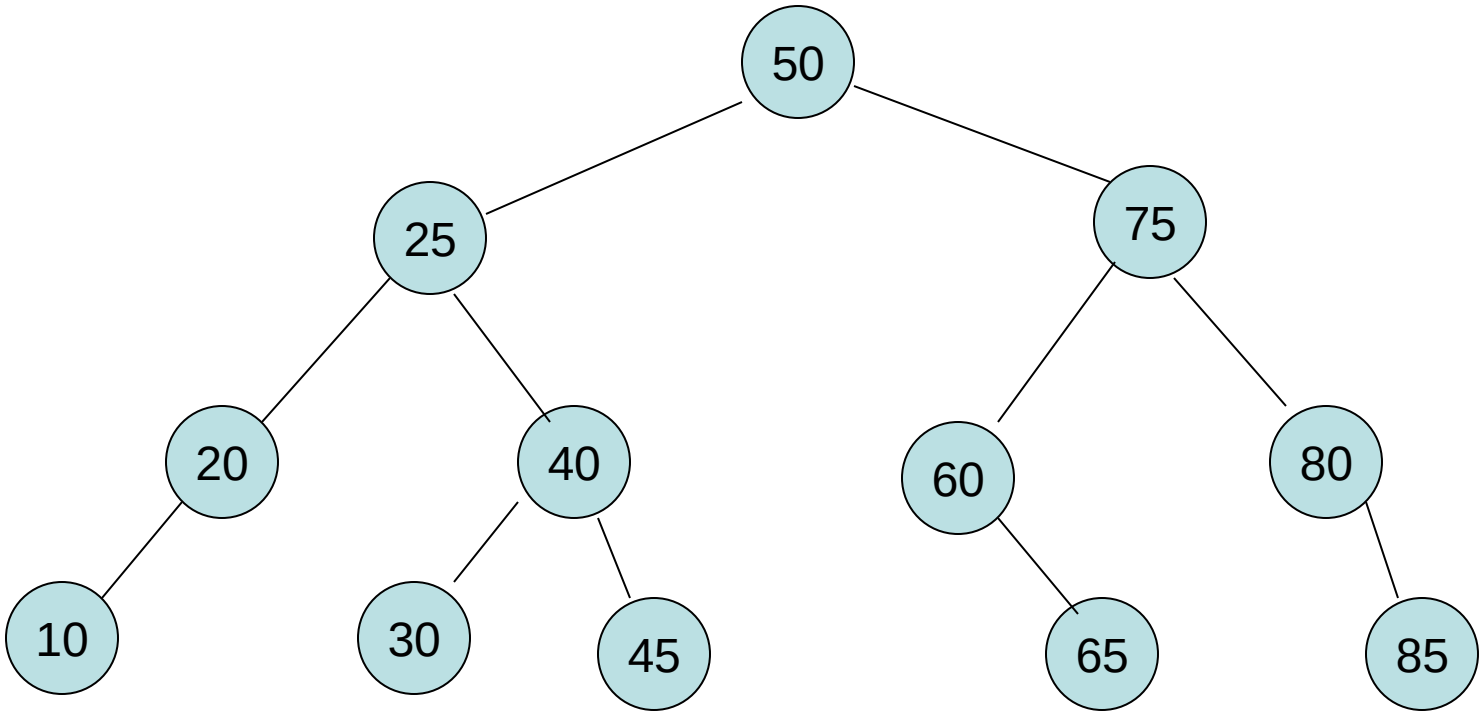
Determining number of nodes/elements

```
Int totalnodes(BST *tree)
{
    if(tree==NULL)
        return 0;
    else
        return (totalnodes(tree->left)+totalnodes(tree->right) + 1);
}
```

Inserting a New Element

- If the binary search tree is initially empty, then the element is inserted as root node.
- Otherwise the element is inserted as terminal node.
- If the element is less than the element in the root node, then the element is inserted in the left sub tree else right sub tree.

BST



Suppose you want to insert 55

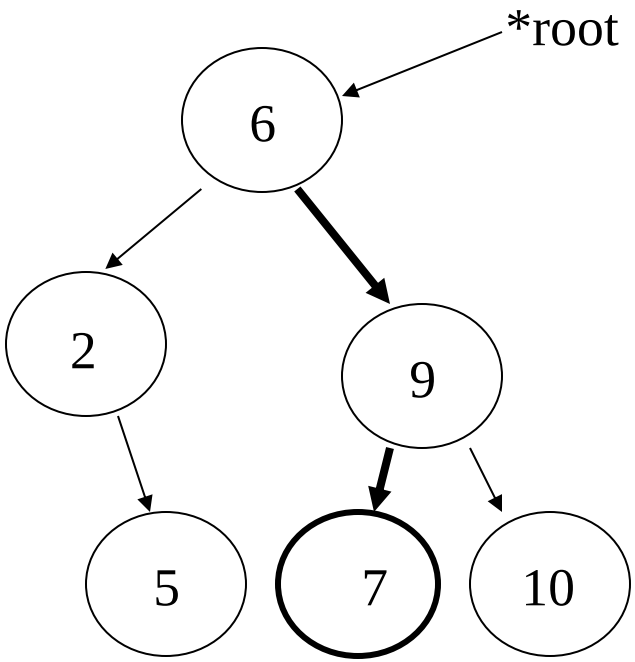
Inserting a new element

```
Void insertelement(BST *tree, int element)
{
	BST *ptr,*nodeptr,*parentptr;
	ptr=(BST*)malloc(sizeof(BST));
	ptr->info=element;
	ptr->left=ptr->right=NULL;
	if(tree==NULL)
		*tree=ptr;
	else
	{
		parentptr=NULL;
		nodeptr=*tree;
		while(nodeptr!=NULL)
		{
			parentptr=nodeptr;
			if(element<nodeptr->info)
				nodeptr=nodeptr->left;
			else
				nodeptr=nodeptr->right;
		}
		if(element<parentptr->info)
			parentptr->left=ptr;
		else
			parentptr->right=ptr;
	}
}
```

Searching an element

- The element in binary search tree can be searched very quickly.
- Search operation on binary tree is similar to applying binary search technique to a sorted linear array.
- The element to be searched will be compared with root node.
- If matches with the root node then search terminates here.
- Otherwise search is continued in the left sub tree if the element is less then the root or in the right sub tree if the element is greater then the root.

The Find Operation…



Suppose I try to find the node with 7 in it. First go down the right subtree, then go down the left subtree.

Searching an element

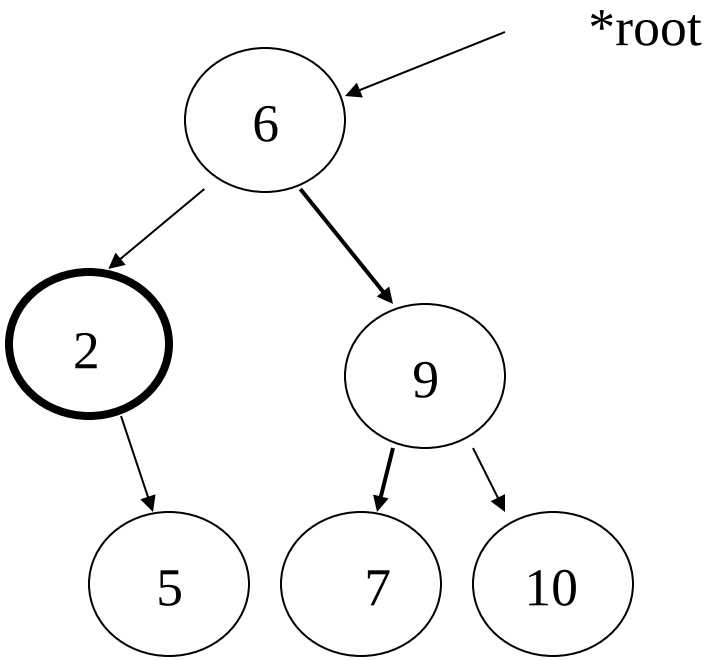
```
BST searchelement(BST *tree, int value)
{
   if((tree->info==value)||tree==NULL)
        return tree;
   else if(value<tree->info)
                return searchelement(tree->left, value);
        else
                return searchelement(tree->right, value);
}
```

Finding smallest node

- Because of the property of binary search tree, we know that the smallest node in the tree will be one of the nodes in the left sub tree, if it exists;
- otherwise node itself will be the smallest node.

```
BST smallest(BST *tree)
{
   if((tree==NULL)||(tree->left==NULL))
        return tree;
   else
        return smallest(tree->left);
}
```

The FindMin Operation…



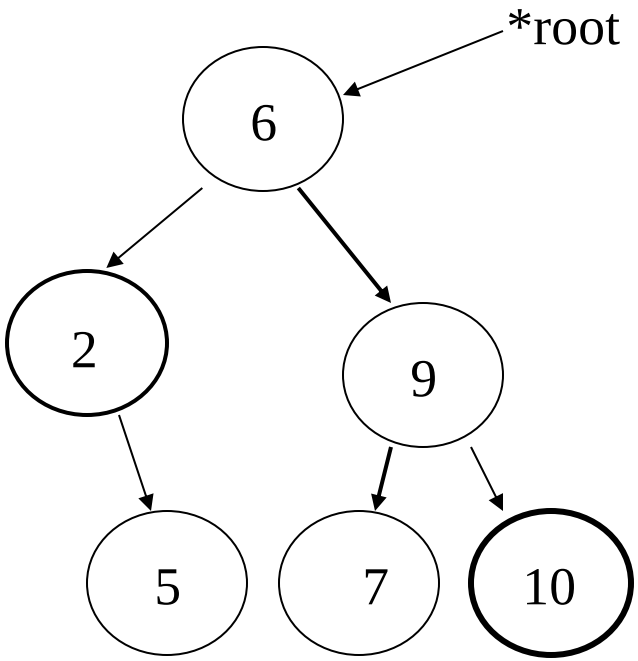
This function returns a pointer to the node containing the smallest element in the tree. It does so by following the left side of the tree.

Finding largest node

- Because of the property of binary search tree, we know that the largest node in the tree will be one of the nodes in the right sub tree, if it exists;
- otherwise node itself will be the largest node.

```
BST largest(BST *tree)
{
   if((tree==NULL)||(tree->right==NULL))
        return tree;
   else
        return largest(tree->right);
}
```

The FindMax Operation…



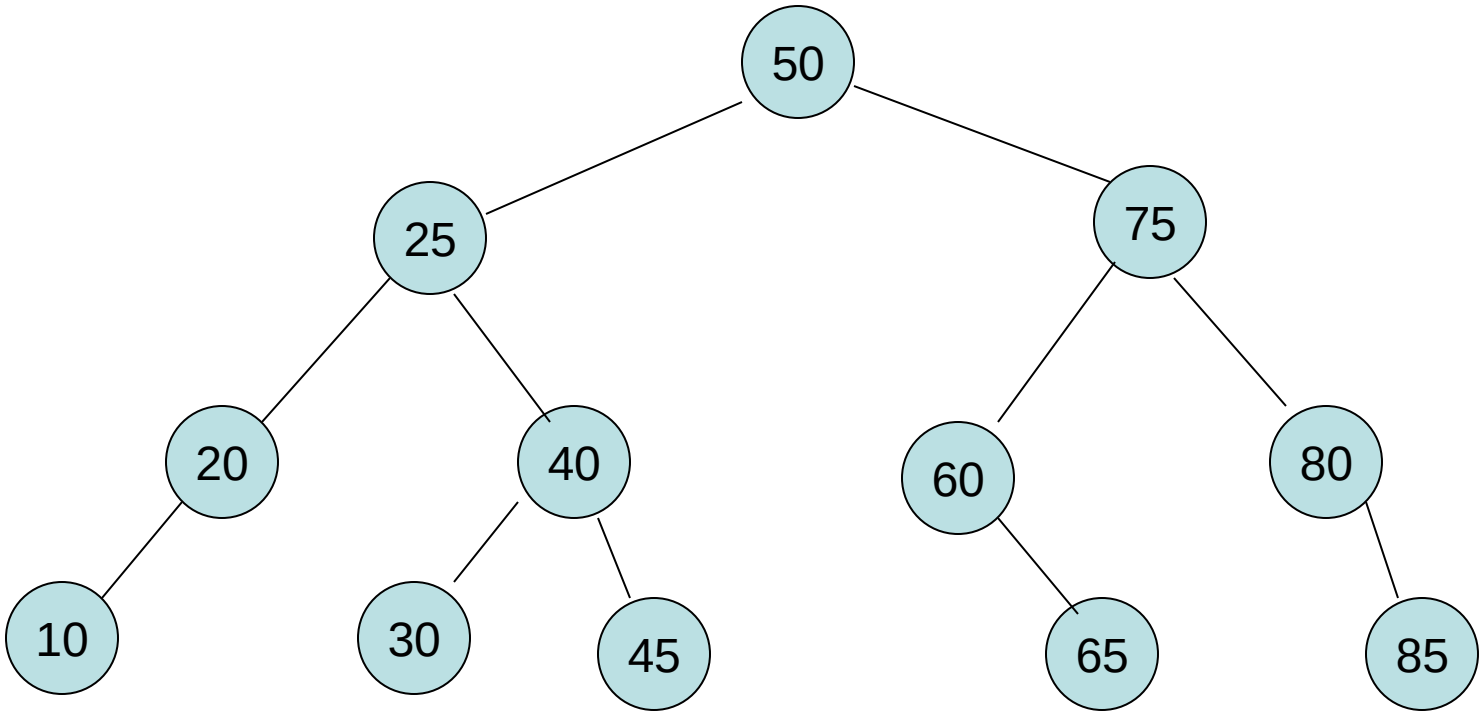
This function returns a pointer to the node containing the largest element in the tree. It does so by following the right side of the tree.

Deleting a node

To delete a node, the following possibilities may arise:

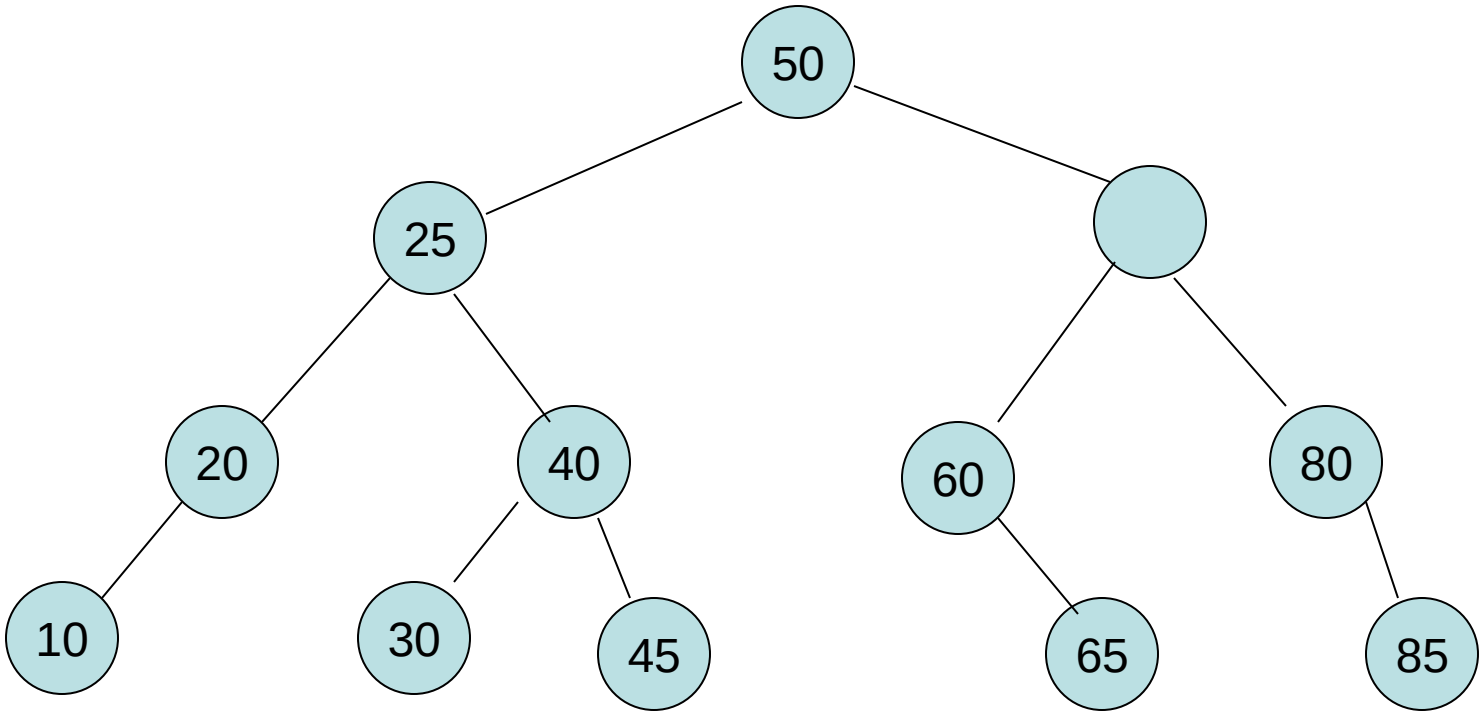
- **Node is terminal node**: in this case if node is left child of its parent, then the left pointer of its parent is set to NULL, otherwise it will be right child of its parent and accordingly pointer of its parent is set to NULL.
- **Node having only one child:** in this case, the appropriate pointer of its parent is set to child, thus by passing it.
- **Node having two children:** predecessor replaces node value, and then the predecessor of node is deleted.

BST



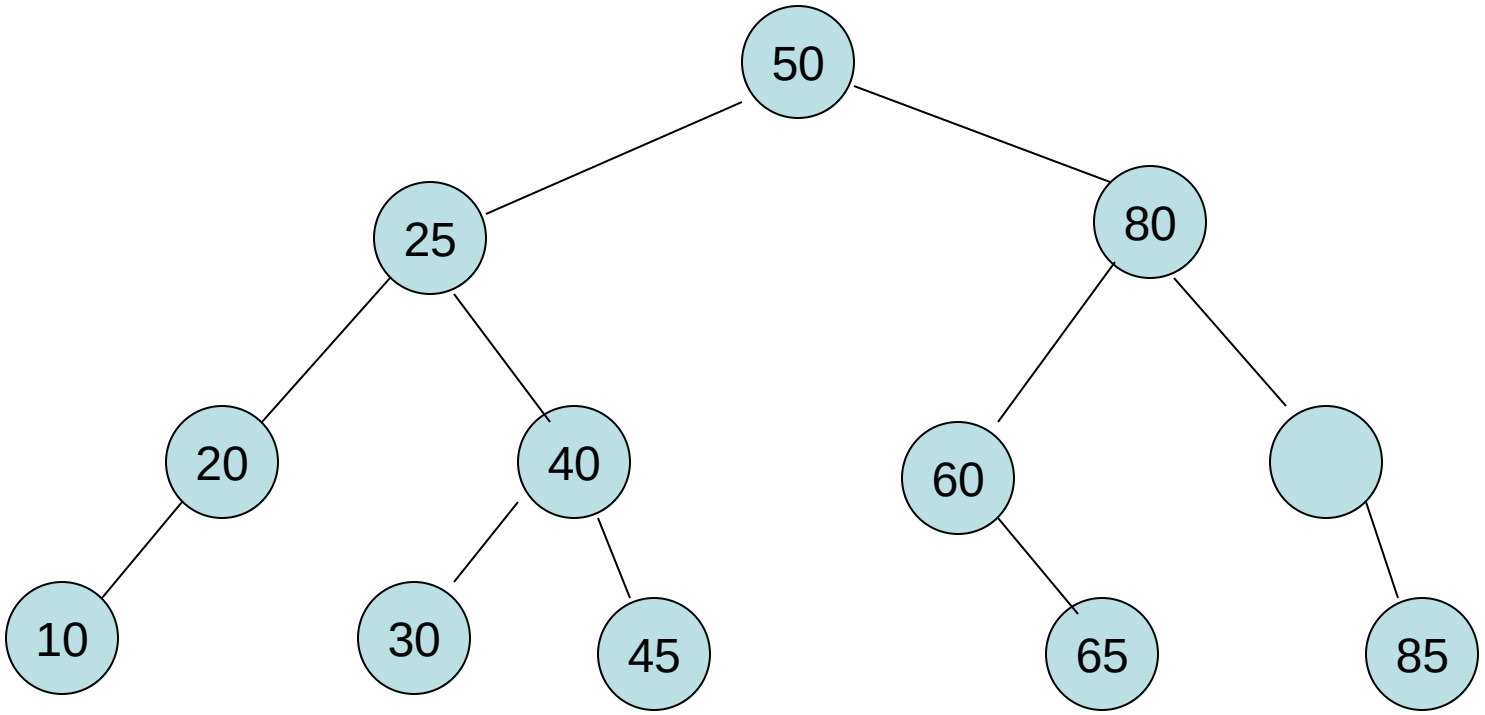
Suppose we want to delete 75

BST



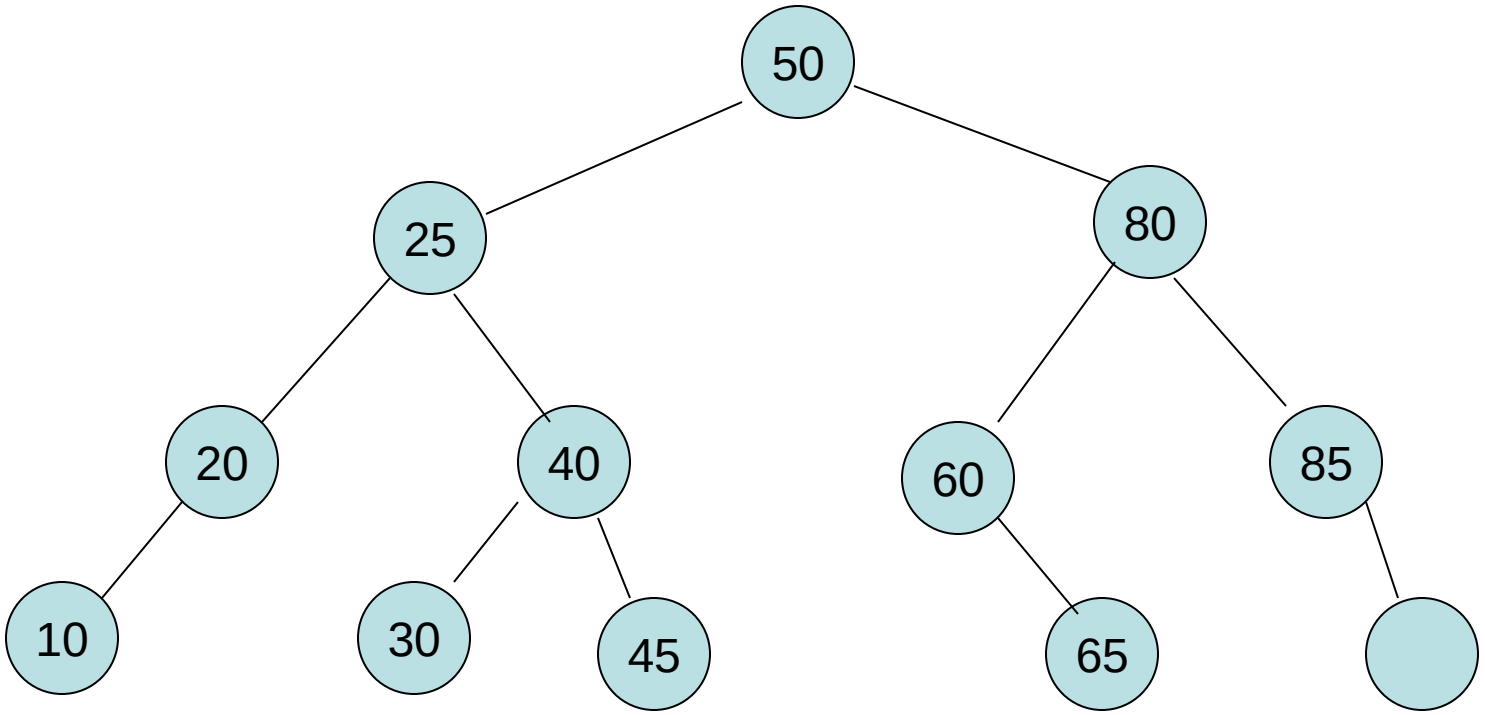
50<75 proceed right , delete 75

BST



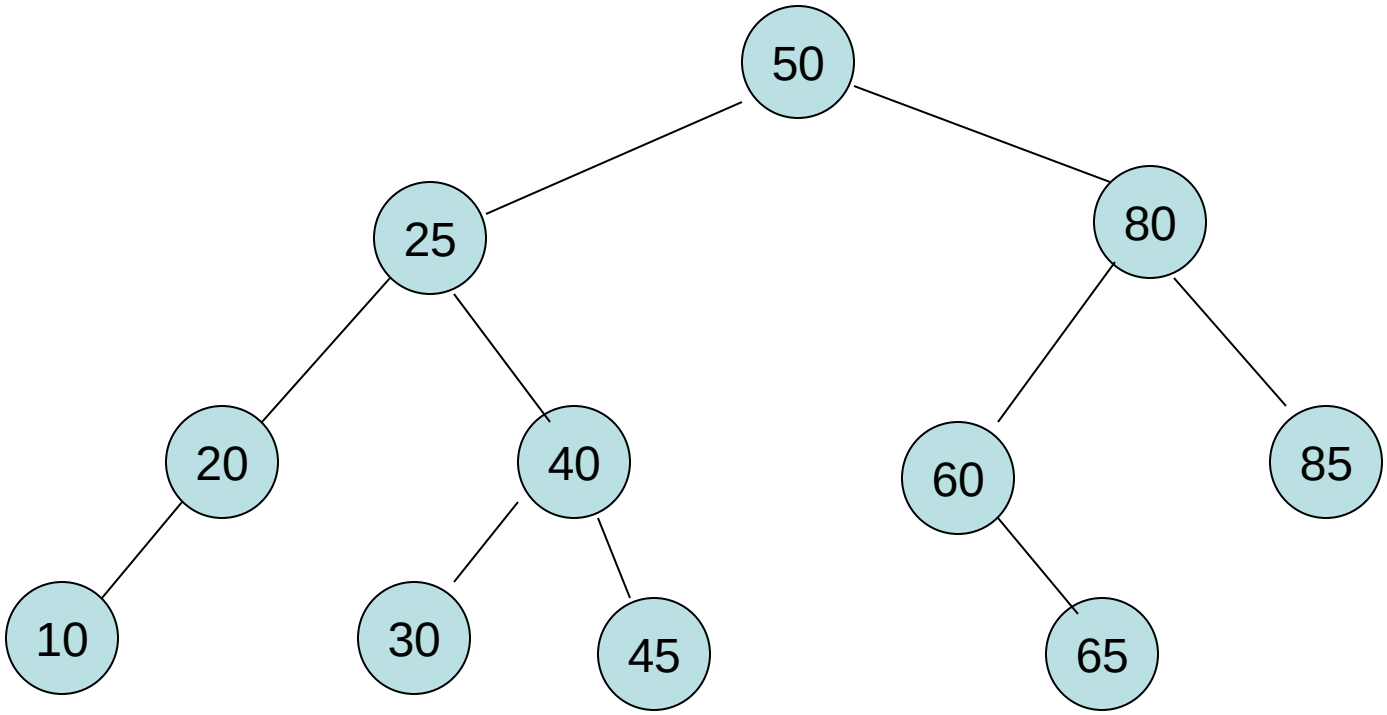
Since node has both right and left child, if right sub tree is opted find the smallest node, if left sub tree is opted find the largest node

BST



Since node has child, if right sub tree is opted find the smallest node, if left sub tree is opted find the largest node

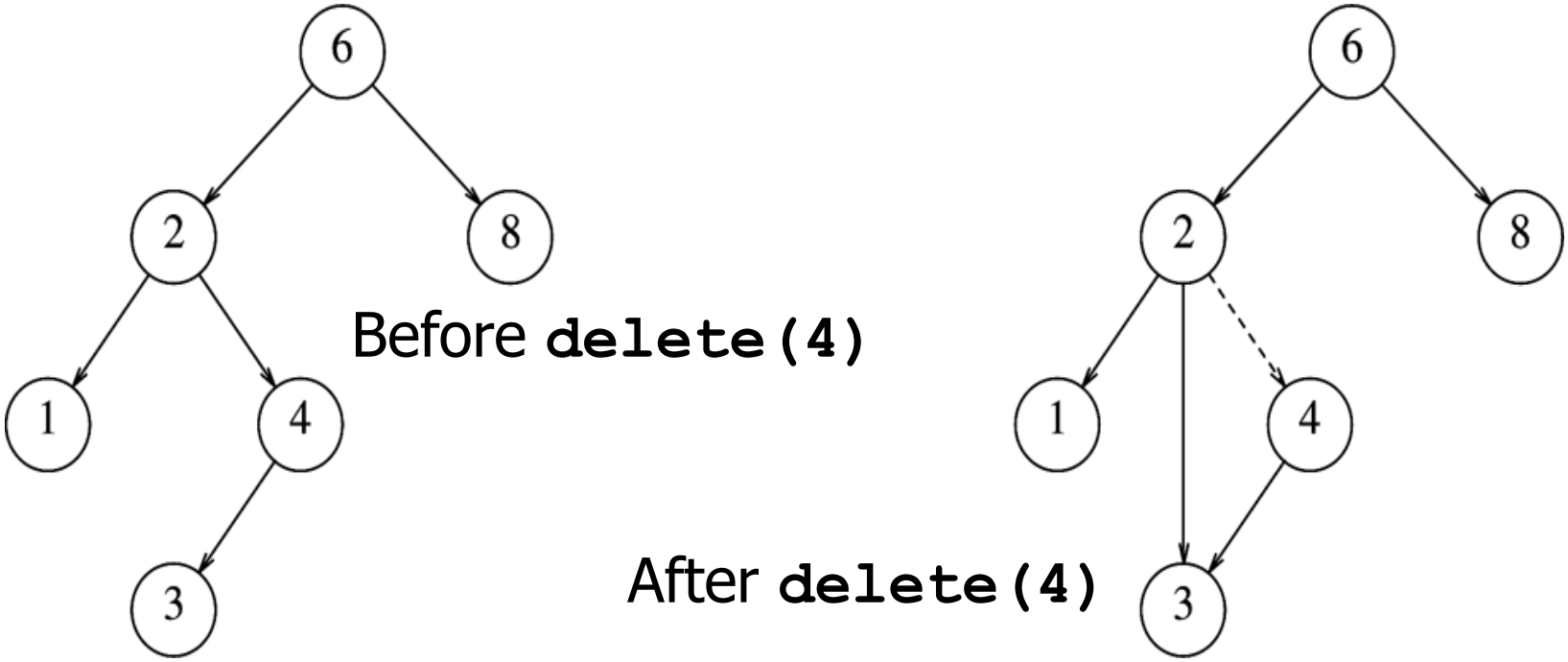
BST



Since node->right=node->left=NULL, delete the node and place NULL in the parent node

BST: Deletion

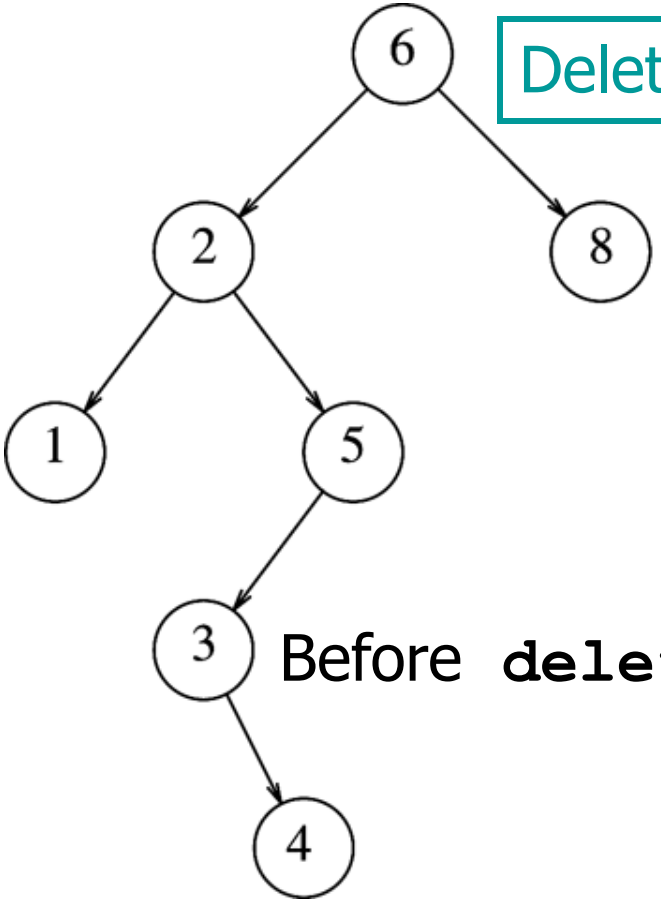
Deleting a node with one child



Deletion Strategy: Bypass the node being deleted

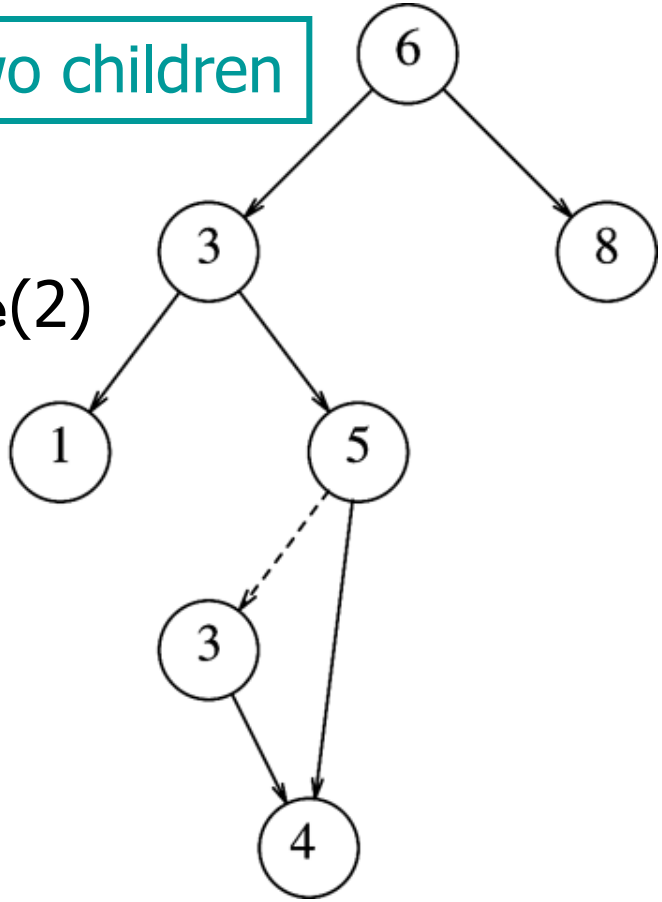
BST: Deletion (contd.)

Deleting a node with two children



Before **`delete(2)`**

After **`delete`**(2)

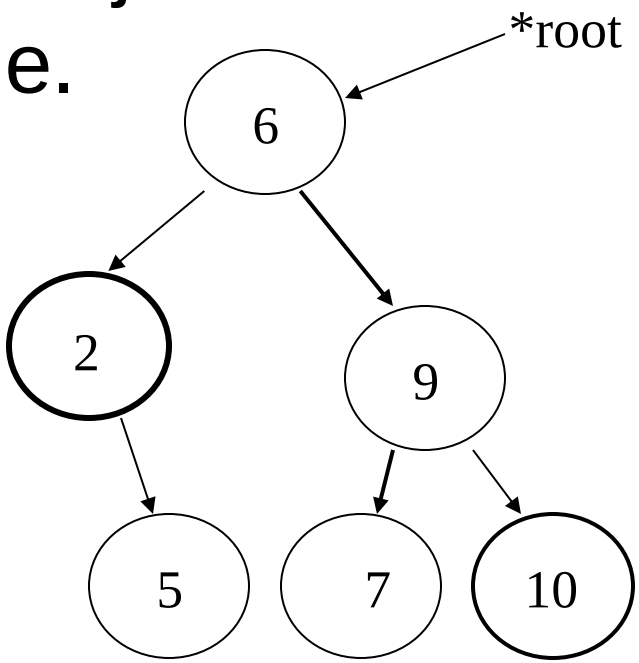


Deletion Strategy: Replace the node with smallest node in the right subtree

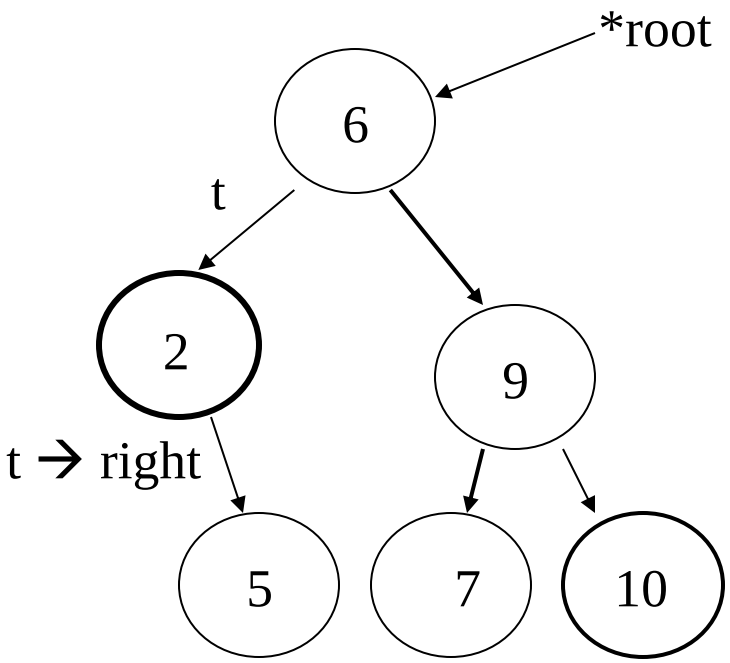
The Removal Operation

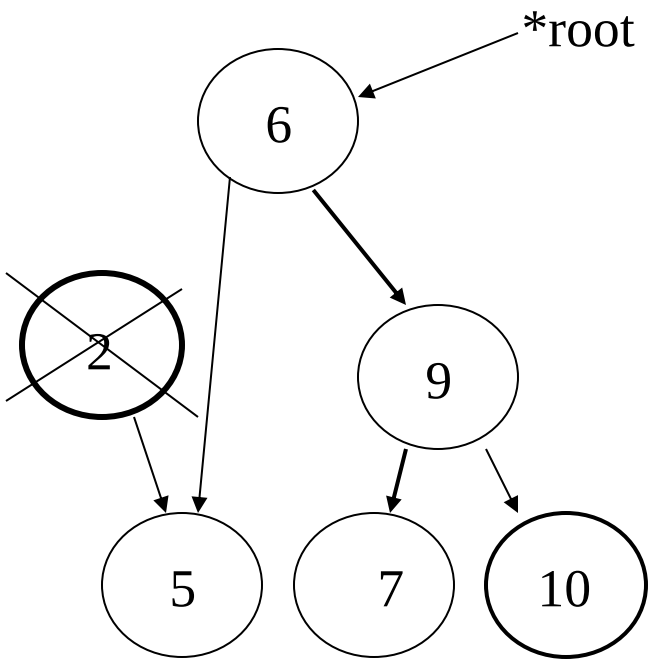
- If the node to be removed is a leaf, it can be deleted immediately.
- If the node has one child, the node can be deleted after its parent adjusts a link to bypass the deleted node.

What if the 2 is deleted?



Removal…





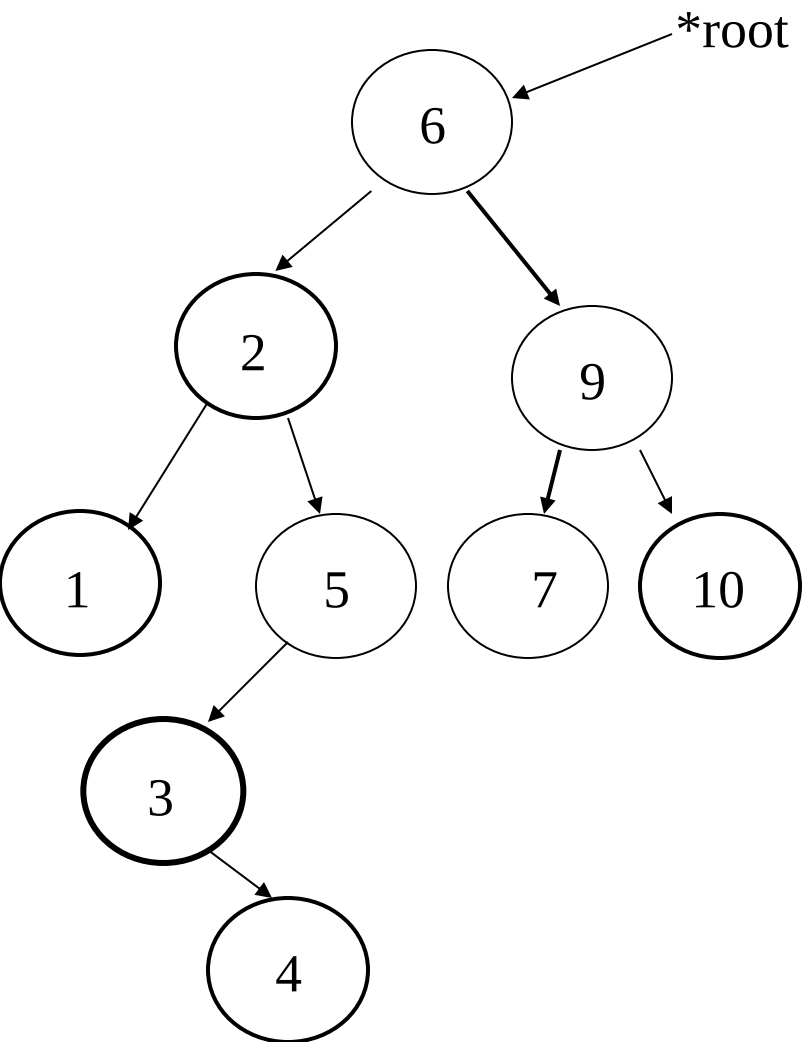
Set t = t → right

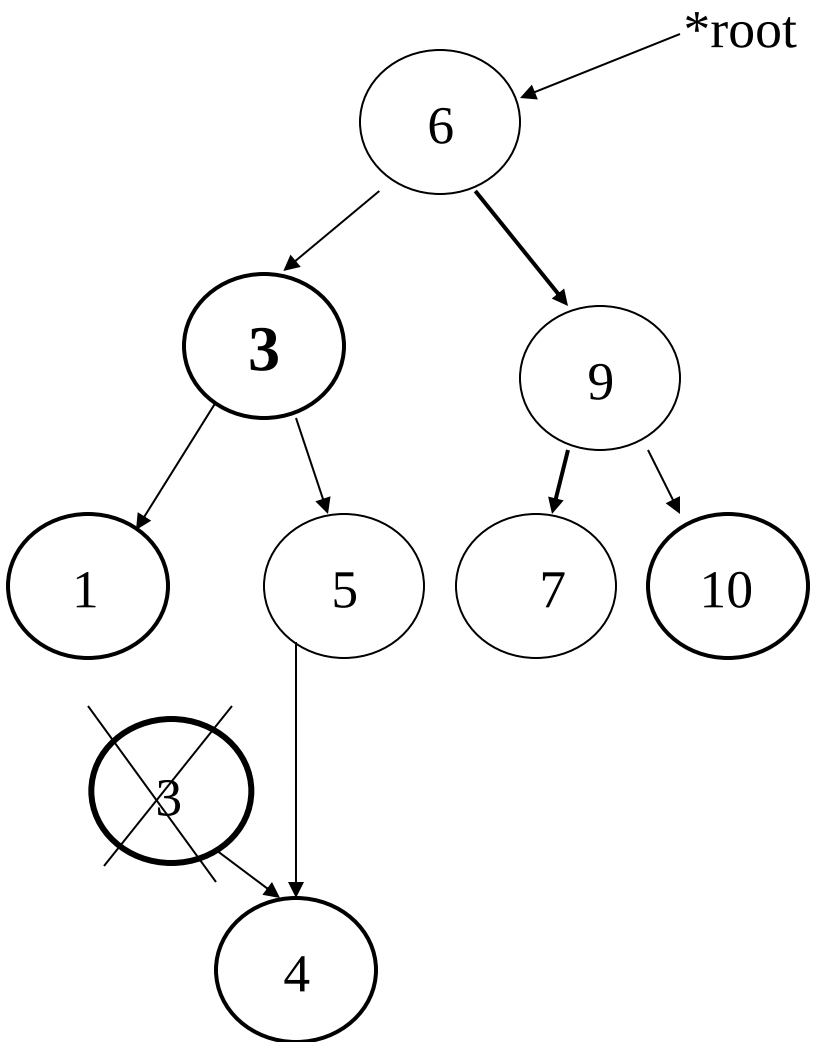
Removal…

- If the node to be removed has two children, the general strategy is to replace the data of this node with the smallest data of the right subtree.
- Then the node with the smallest data is now removed (this case is easy since this node cannot have two children).

Removal…

Remove the 2 again…

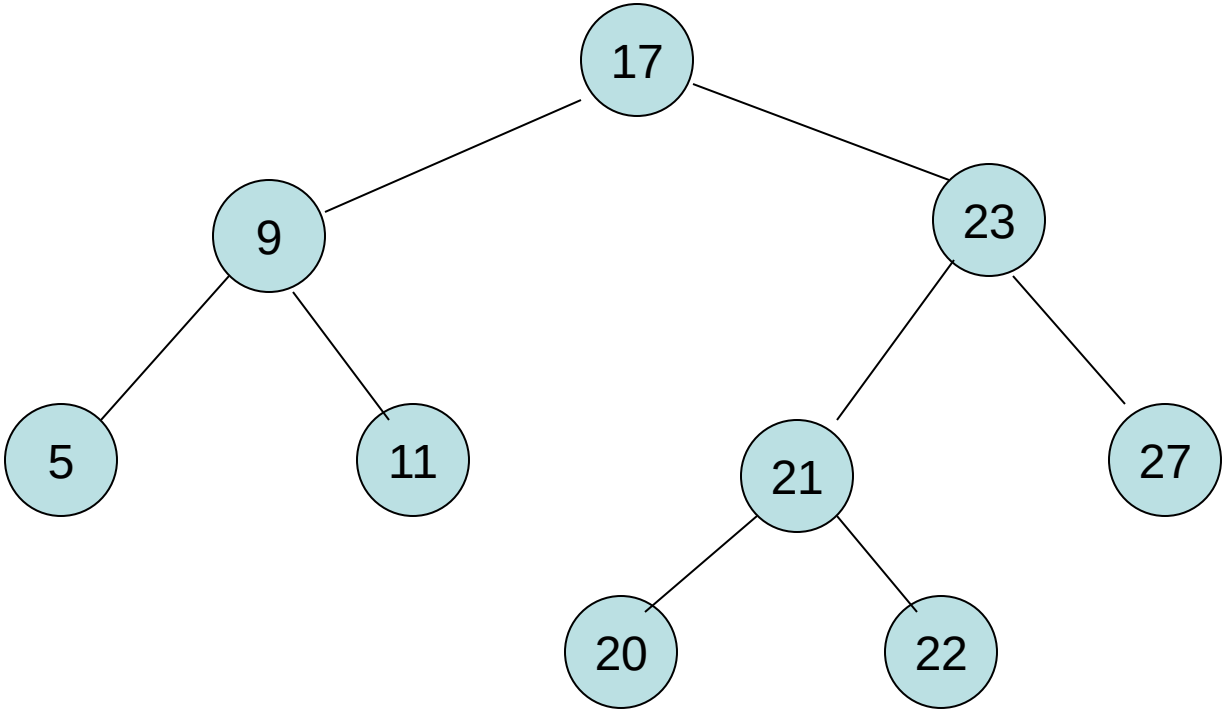




The Removal Operation

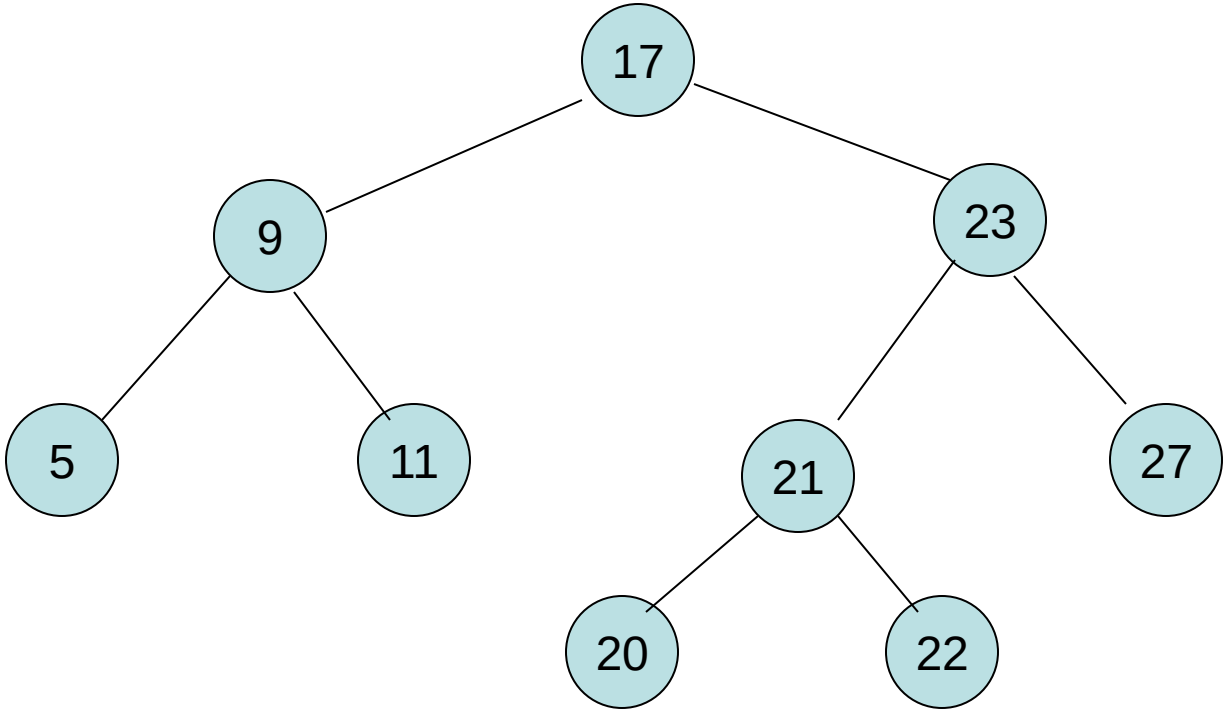
```
delBST(root, delkey)
1.   if(empty tree)
     return false
     endif
2. If (delkey<root)
     rerturn delBST(leftsubtree, delkey)
3. else if(delkey>root)
      rerturn delBST(rightsubtree, delkey)
4. Else
     if(no left subtree)
              make right subtree the root
              return true.
     else if(no right subtree)
              make left subtree the root
              return true.
     else
              save root in deletenode
              set largest to largestBST(left subtree)
              move data in largest to deletenode
              return delBST(left subtree of deletenode, key of the largest)
     endif
     endif
End delBST
```

23 is to be deleted
find delkey

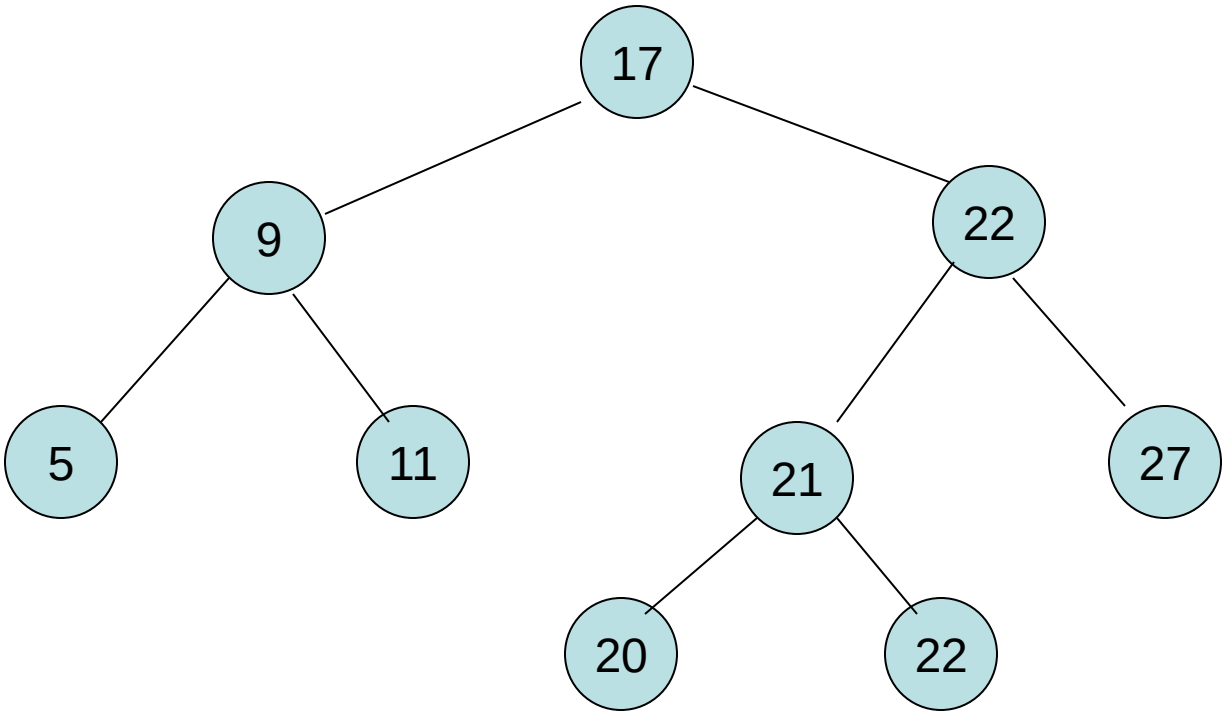


Find largest data

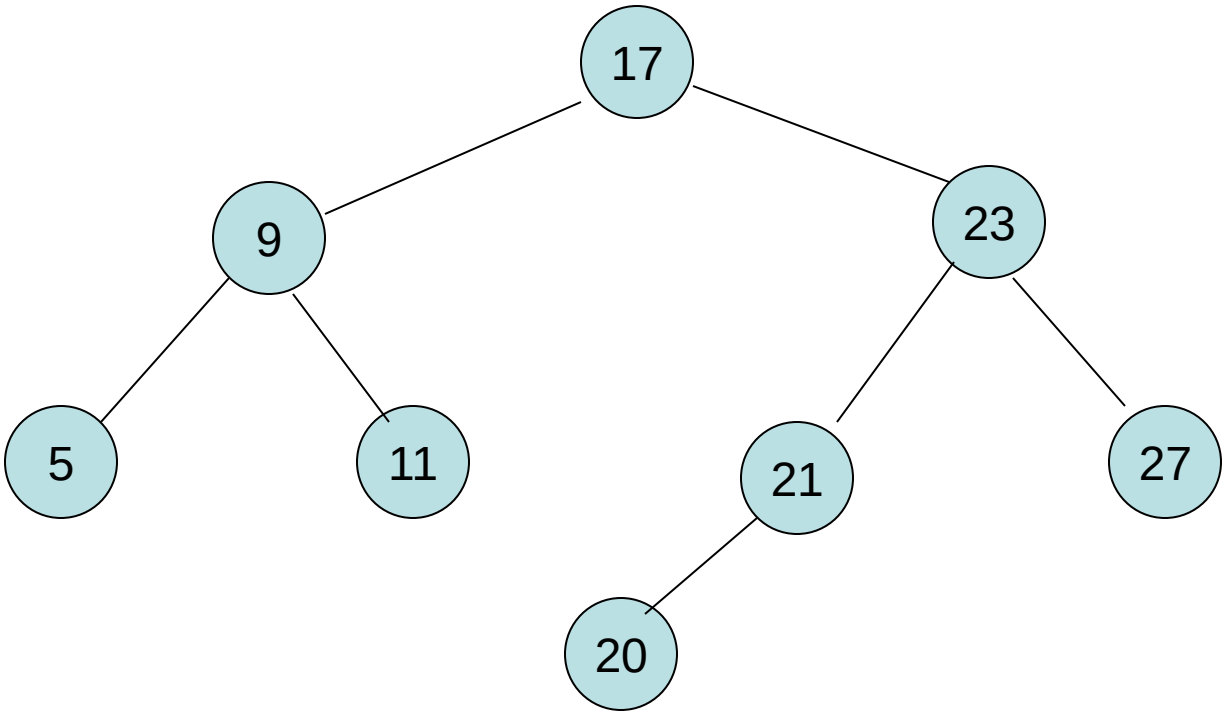
22 is the largest key on the left subtree



Move largest data



Delete largest node



AVL TREES

- We can guarantee $O(\log_2 n)$ performance for each search tree operation by ensuring that the search tree height is always $O(\log_2 n)$.
- Trees with a worst case height of $O(\log_2 n)$ are called balanced trees.
- One of the popular balanced tree is AVL tree, which was introduced by **Adelson-Velskii and Landis.**

AVL TREES

If T is non empty binary tree with T_L and T_R as its left and right sub tree, then T is an AVL tree if and only if:

1. $|h_L - h_R| <= 1$, where h_L and h_R are the heights of T_L and T_R respectively.
2. T_L and T_R are AVL trees

AVL TREES

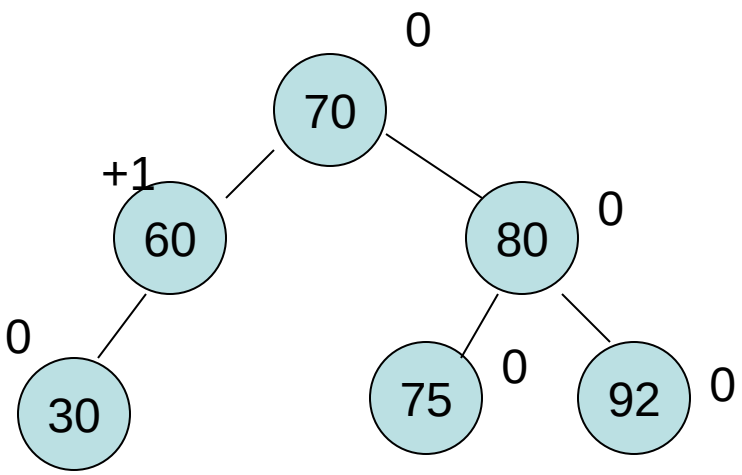


Fig (a)

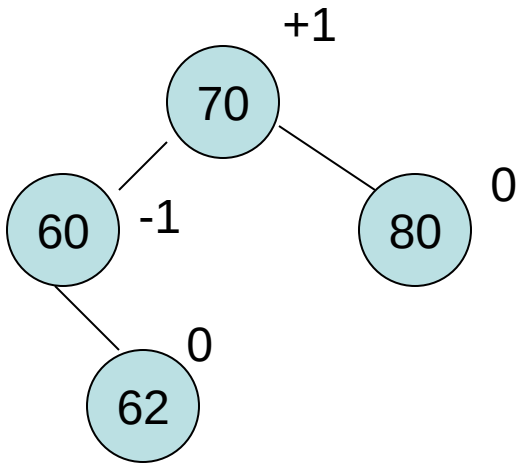


Fig (b)

Representation of AVL trees

The node of the AVL tree is additional field bf (balanced factor) in addition to the structure to the node in binary search tree.

```
typedef struct nodetype
{
        struct nodetype *left;
        int info;
        int bf;
        struct nodetype *right;
}avlnode;
avlnode *root;
```

AVL TREES

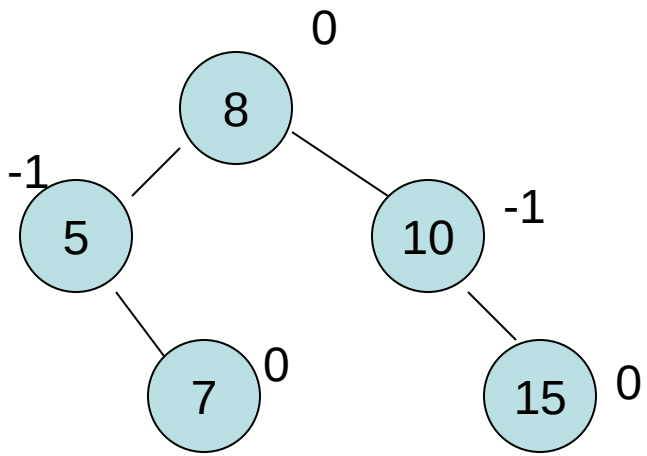
The value of the field bf will be chosen as:

$$bf = \begin{cases} -1 & \text{if} \quad h_L < h_R \\ 0 & \text{if} \quad h_L = h_R \\ +1 & \text{if} \quad h_L > h_R \end{cases}$$

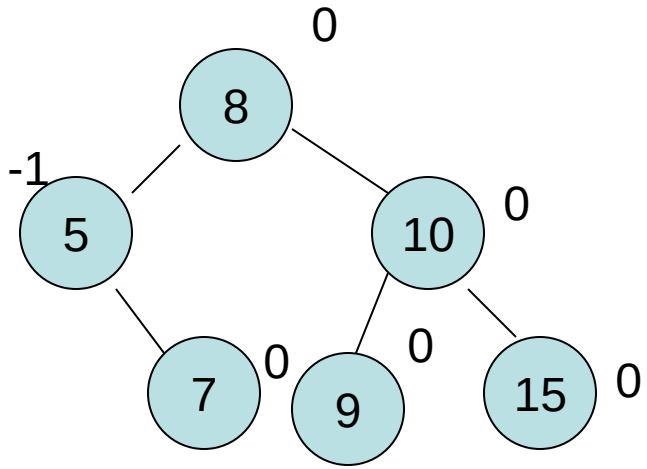
Operation on AVL TREES

- **Insertion of a node:** The new node is inserted using the usual binary search tree insert procedure i.e. comparing the key of the new node with that in the root, and inserting new node into left or right sub tree as appropriate.
- After insertion of new nodes two things can be changed i.e.
  - **Balanced factor**
  - **height**

AVL TREES



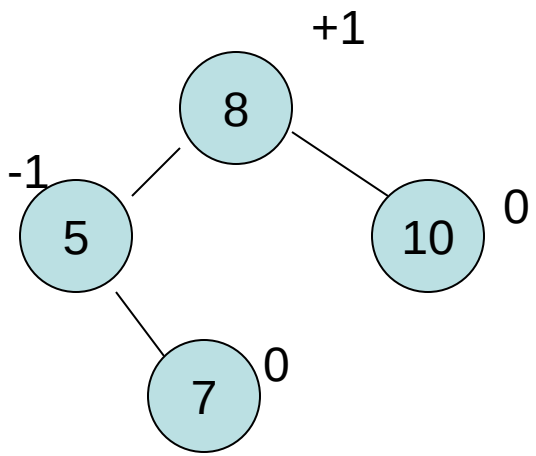
Original AVL Tree



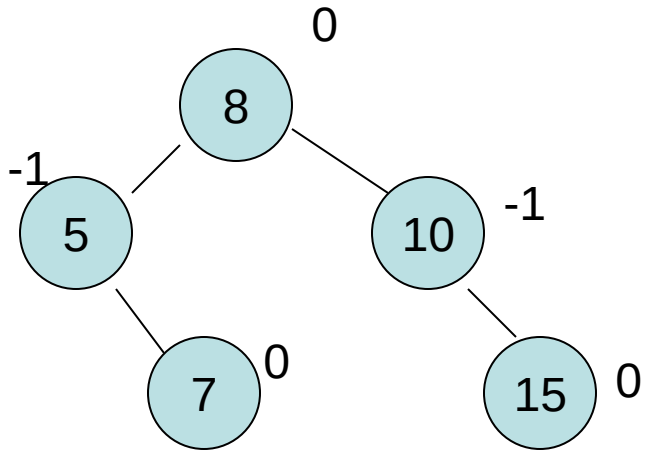
After inserting value 9

Neither the balance factor nor the height of the AVL tree is affected

AVL TREES



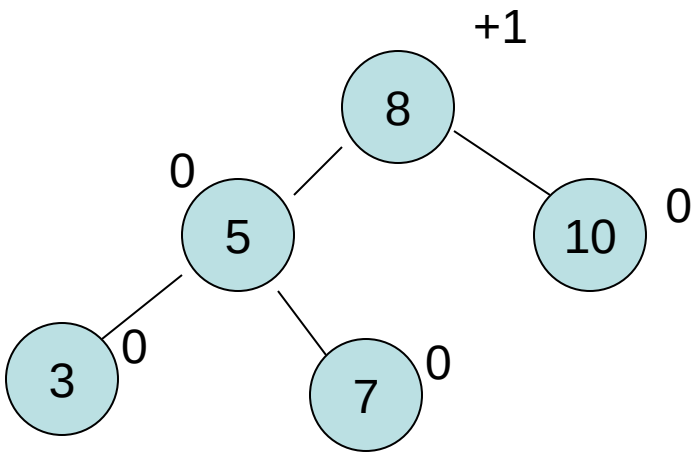
Original AVL Tree

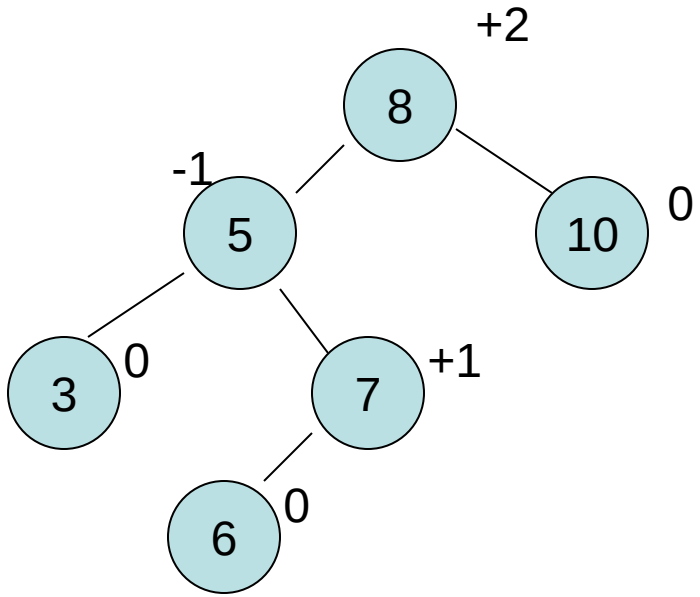


After inserting value 15

Height remains unchanged but the balance factor of the root gets changed

AVL TREES





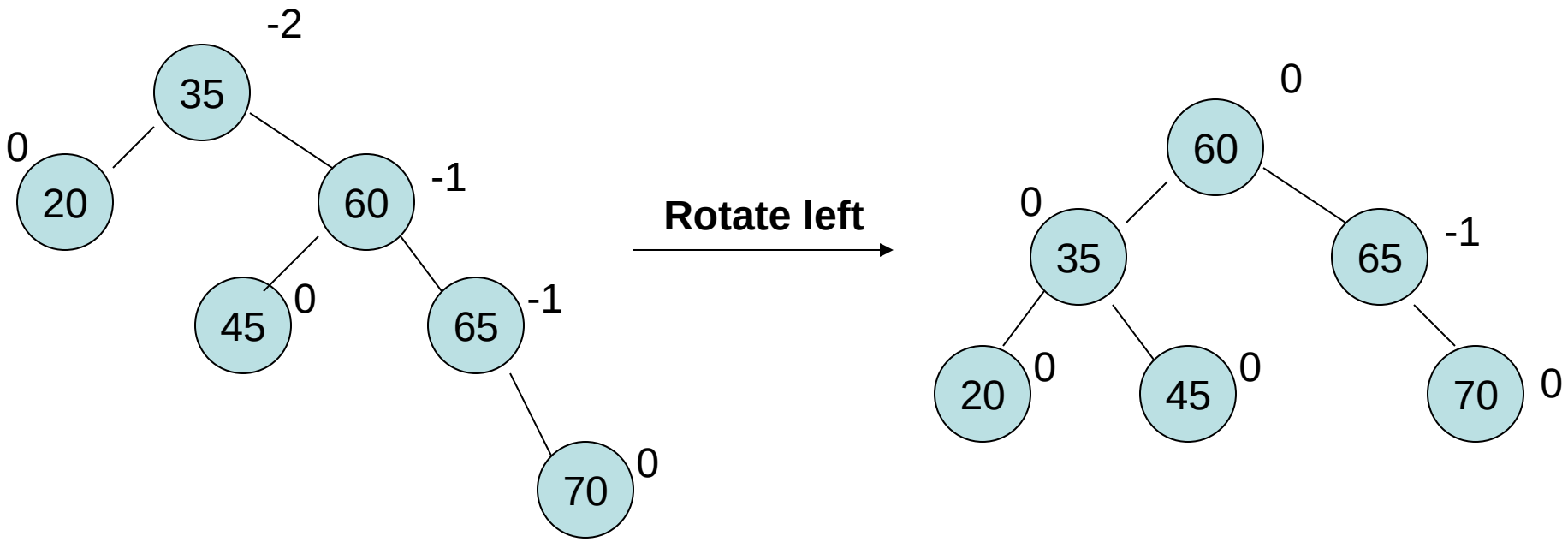
Original AVL Tree

After inserting value 6

Height as well as balanced factor gets changed. It needs rearranging about root node

•In order to restore the balance property, we use the tree rotations

AVL TREES

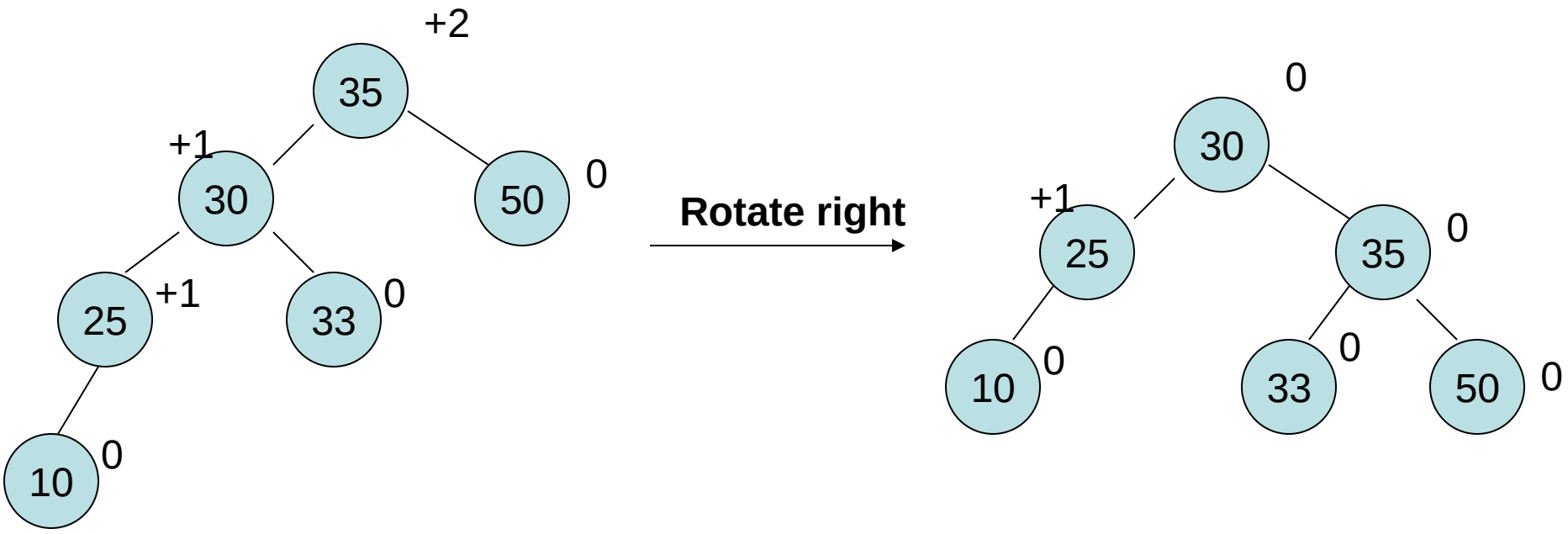


Total height 4

Total height 3

Restoring balance by left rotation

AVL TREES

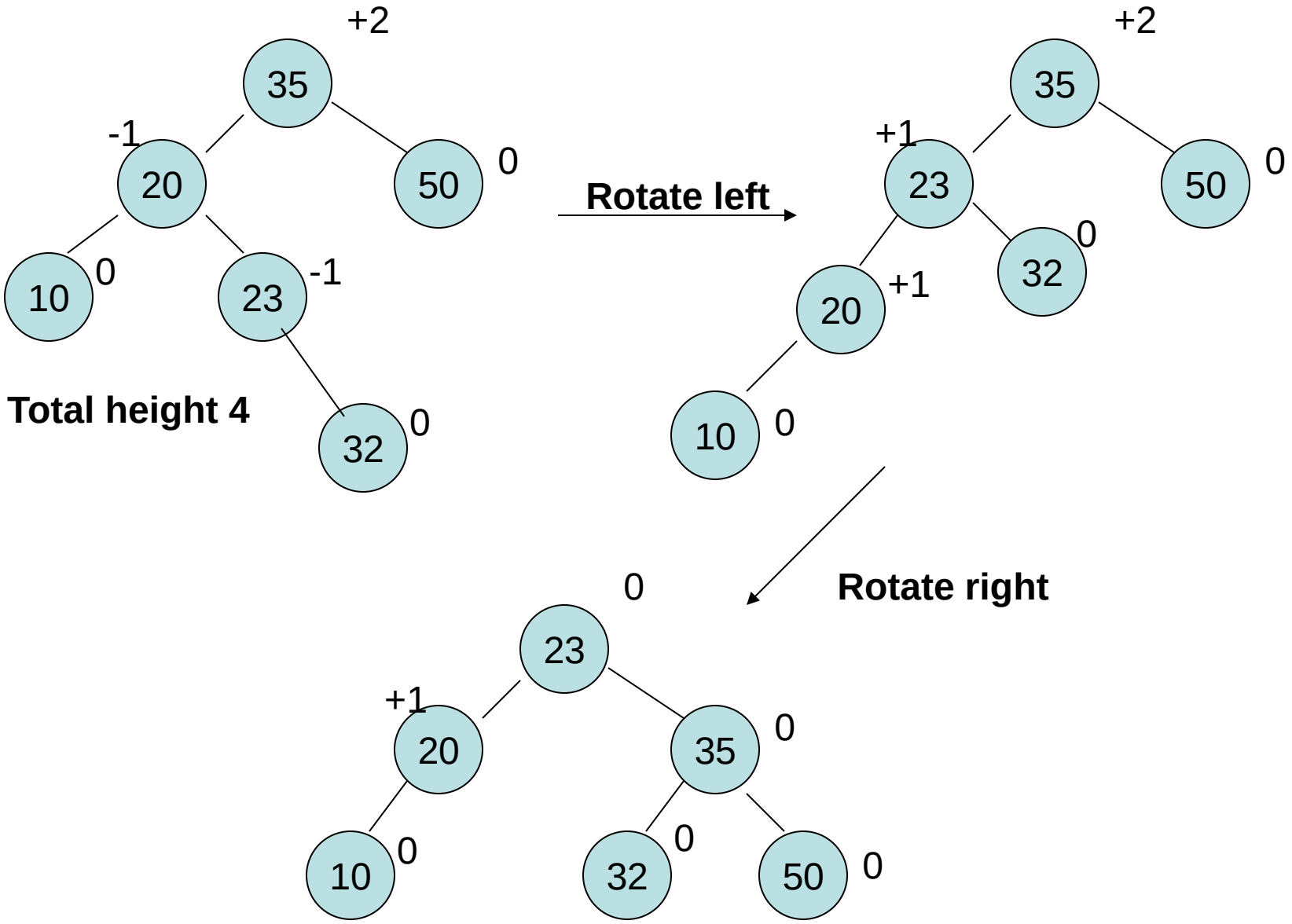


Total height 4

Total height 3

Restoring balance by right rotation

AVL TREES



Restoring balance by double rotation

Threaded Binary trees

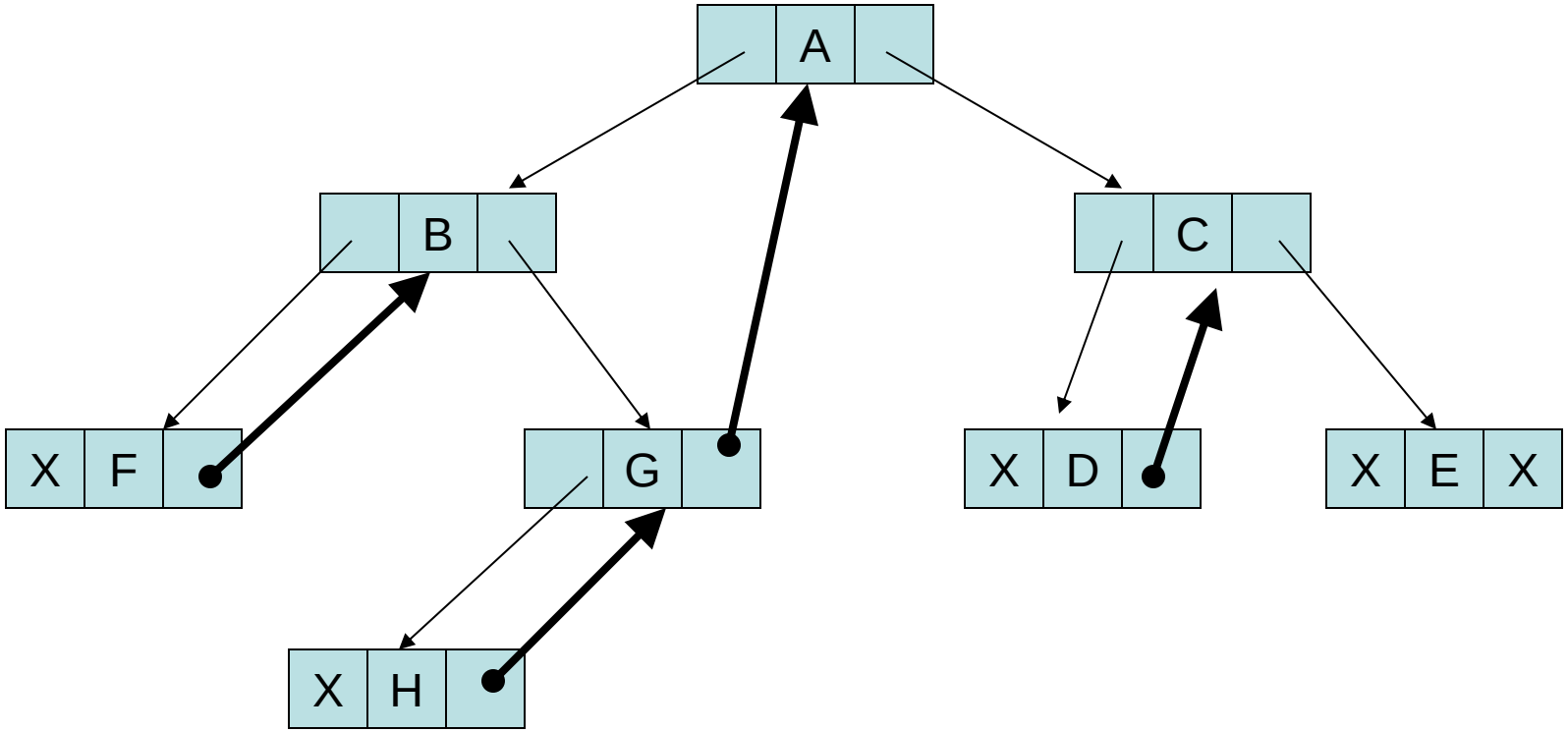
- On carefully examining the linked representation of a binary tree T , we will find that approximately half of the pointer fields contains NULL entries.
- The space occupied by these NULL entries can be utilized to store some kind of valuable information.
- One way to utilize this space is that we can store special pointer that points to nodes higher in the tree, i.e. ancestors.
- These special pointer are called **threads**, and the binary tree having such pointers is called a **threaded binary tree.**
- In computer memory, an extra field, called **tag or flag** is used to distinguish thread from a normal pointer.

Threaded Binary trees

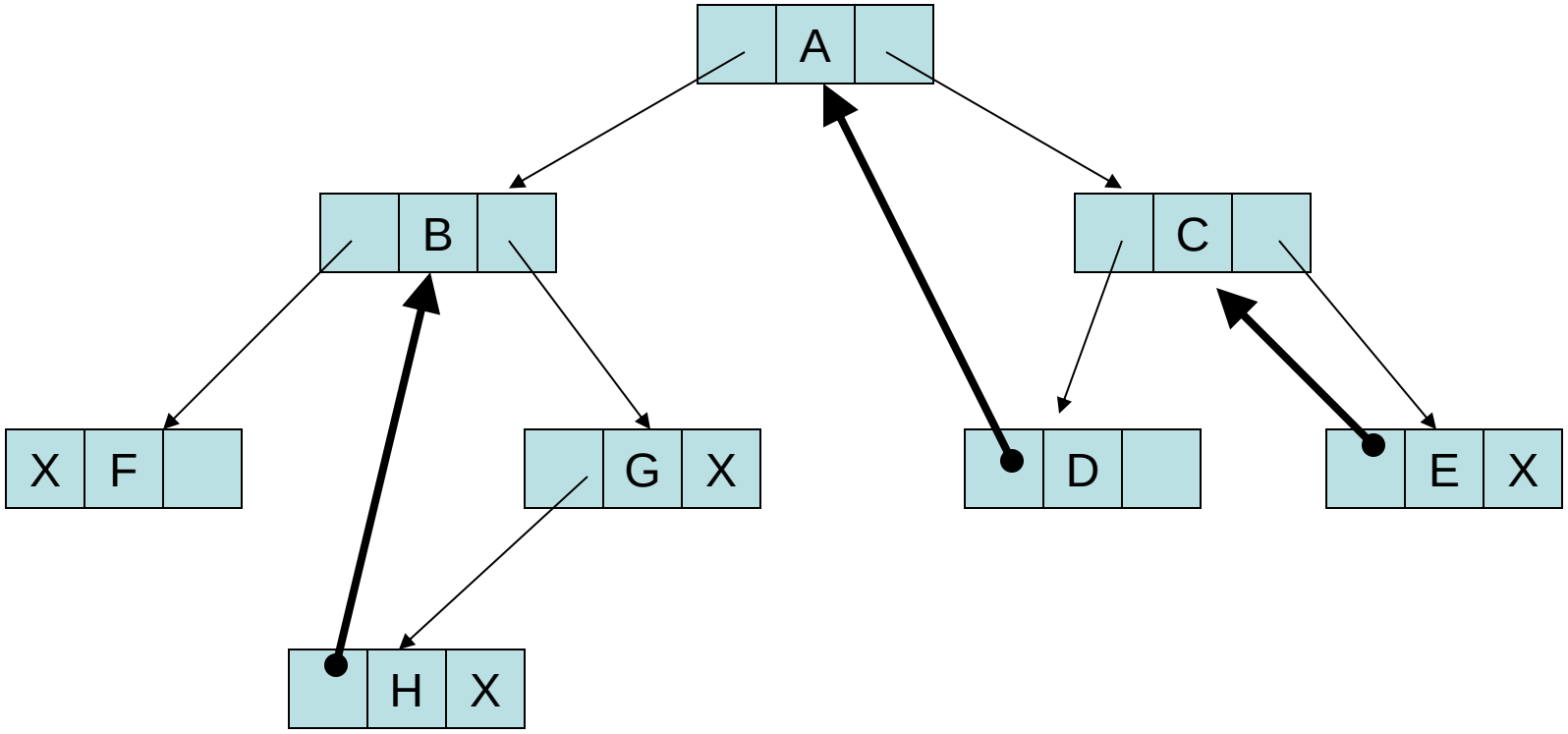
There are many ways to thread a binary tree:

- The right NULL pointer of each node can be replaced by a thread to the successor of the node under in-order traversal called a right thread, and the tree will called a **right threaded tree.**
- The left NULL pointer of each node can be replaced by a thread to the predecessor of the node under in-order traversal called a left thread, and the tree will called a **left threaded tree.**
- Both left and right NULL pointers can be used to point to predecessor and successor of that node respectively, under in-order traversal. Such a tree is called a **fully threaded tree.**
- **A threaded binary tree where only one thread is used is known as one way threaded tree and where both thread are used called two way threaded tree.**

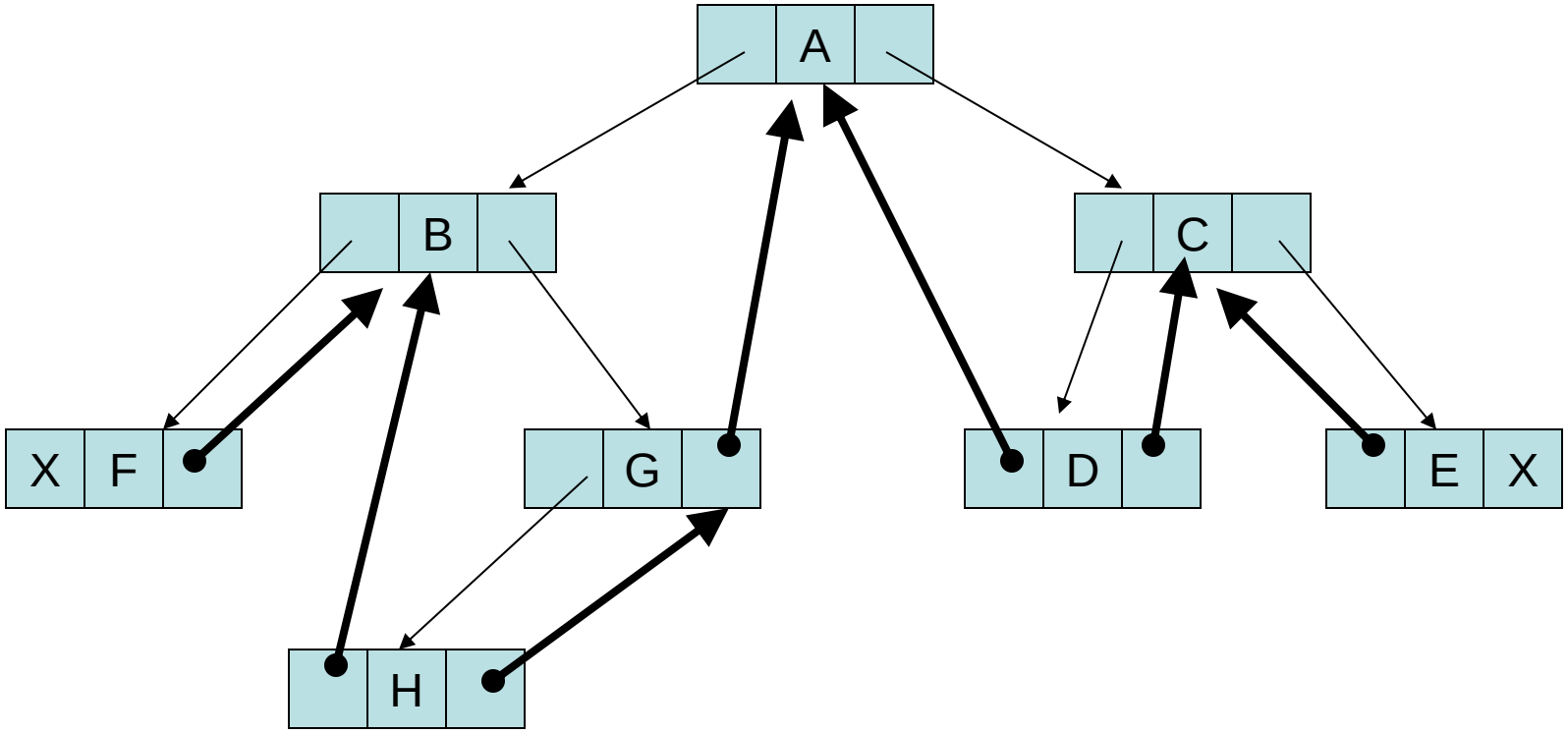
Right threaded binary tree (one way threading)



Left threaded binary tree (one way threading)



Fully threaded binary tree (two way threading)



Representation of threaded binary tree in memory

```
typedef struct nodetype
        {
                struct nodetype *left;
                int info;
                char thread;
                struct nodetype *right;
        }TBST;
TBST *root;
```

In this representation, we have used char field thread as a tag. The character ‘0’ will used for normal right pointer and character ‘1’ will be used for thread.