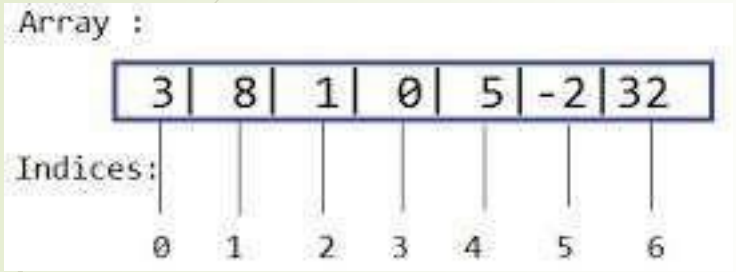
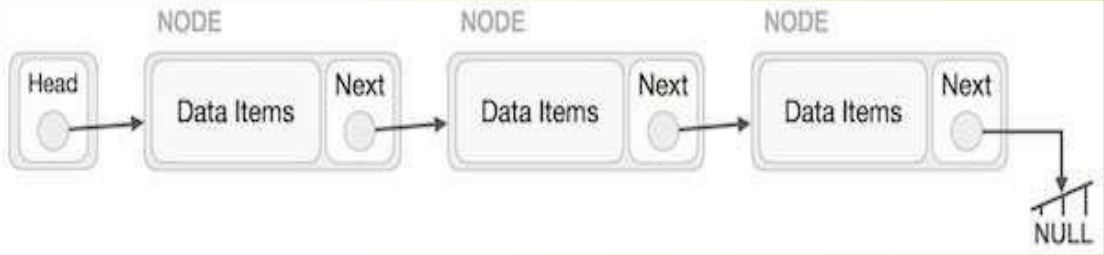
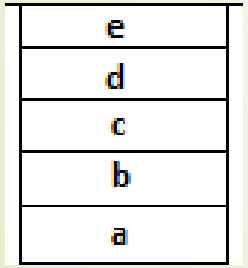
Tree

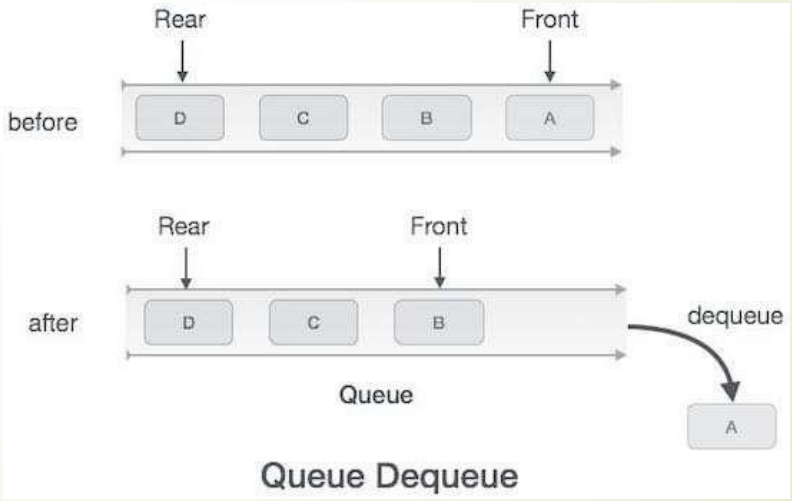
So far we discussed Linear data structures like







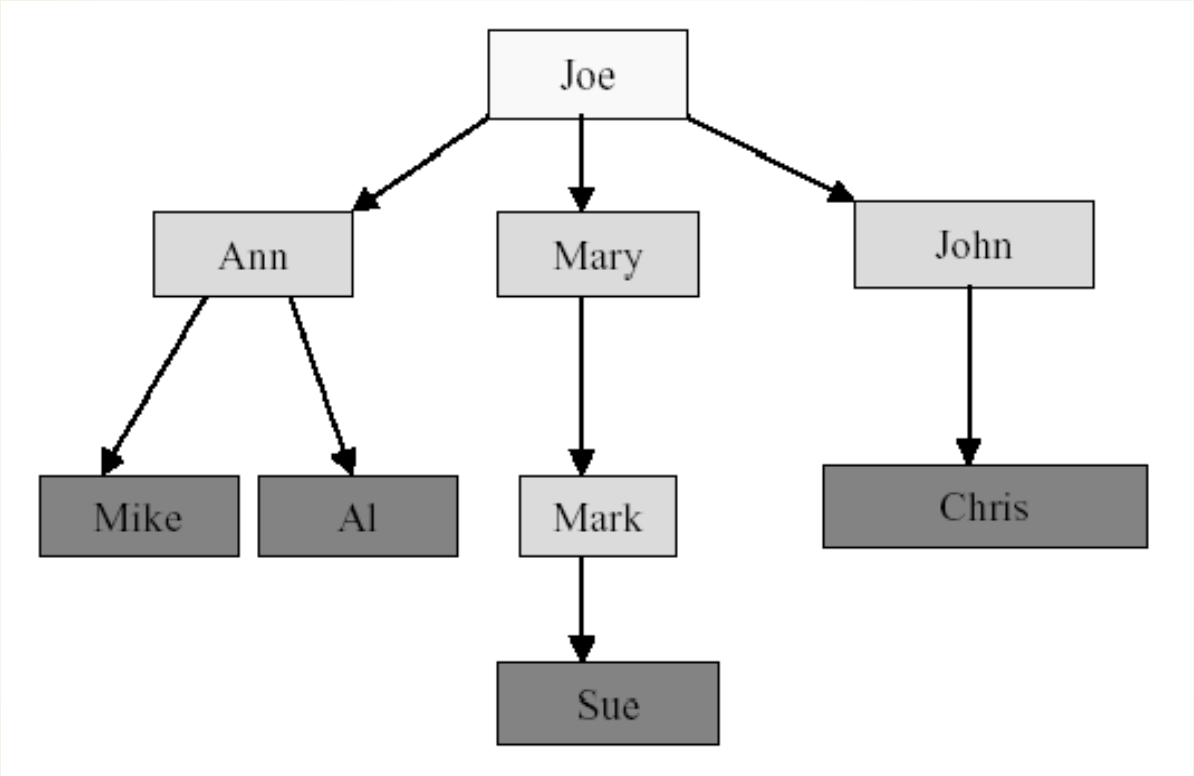
stack



Introduction to trees

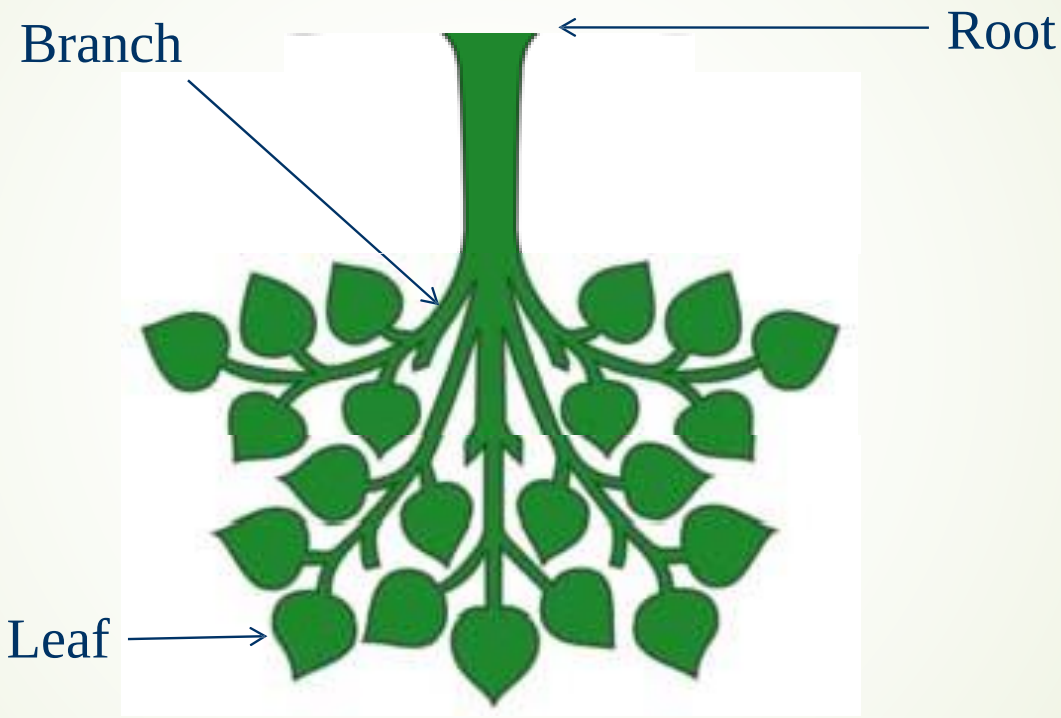
- Now we will discuss a non-linear data structure called tree.
- Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents, sub-structures, sub-divisions.
- Consider a parent-child relationship

Joe's Descendants

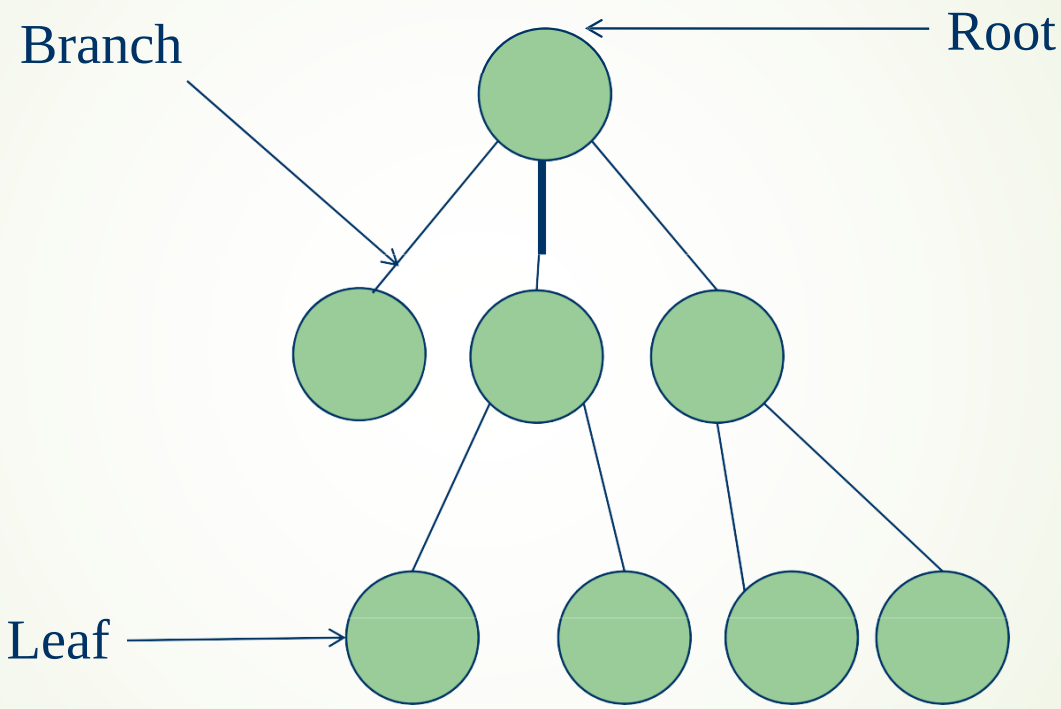


Real Tree vs Tree Data Structure

Real Tree vs Tree Data Structure

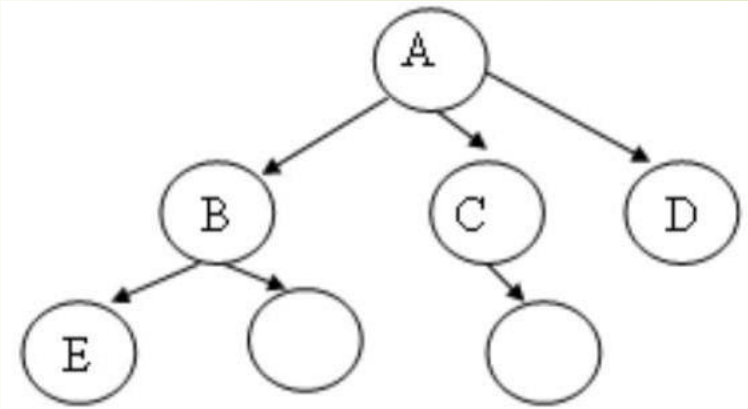


Real Tree vs Tree Data Structure



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

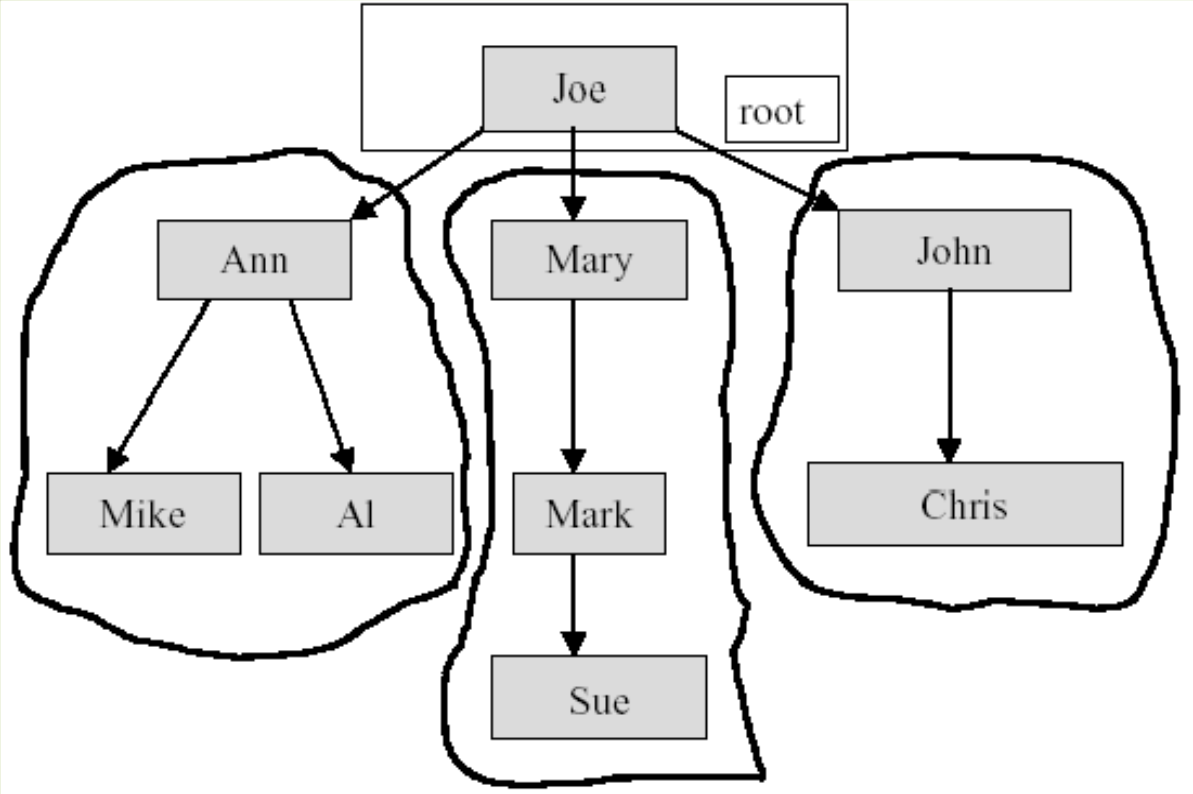


A has 3 children, B, C, D
A is parent of B

Definition of Tree

- A tree *t* is a finite nonempty set of elements
- One of these elements is called the root
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of *t*.

Subtrees

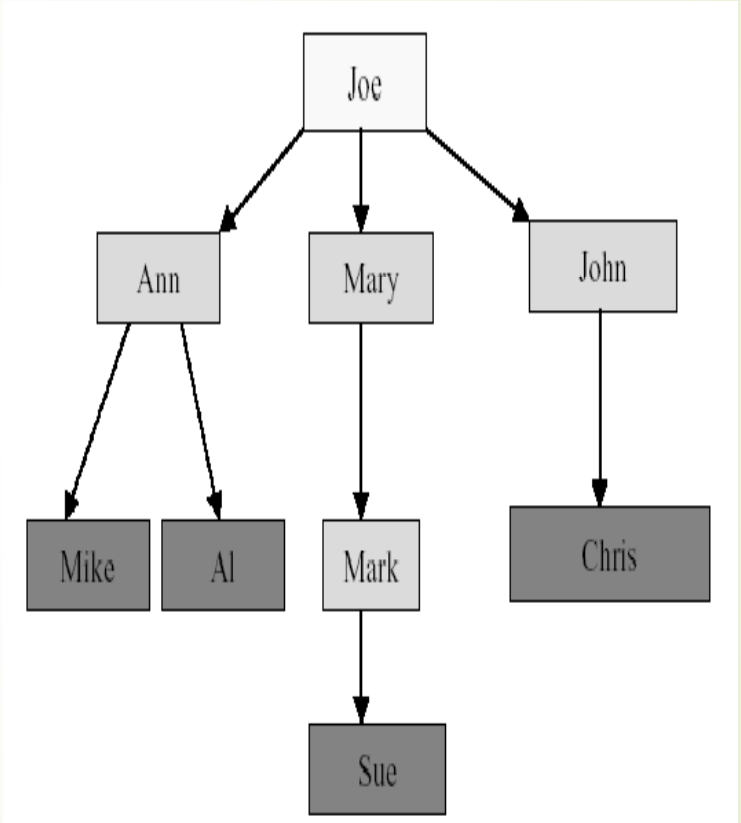


Some Key Terms

- 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.

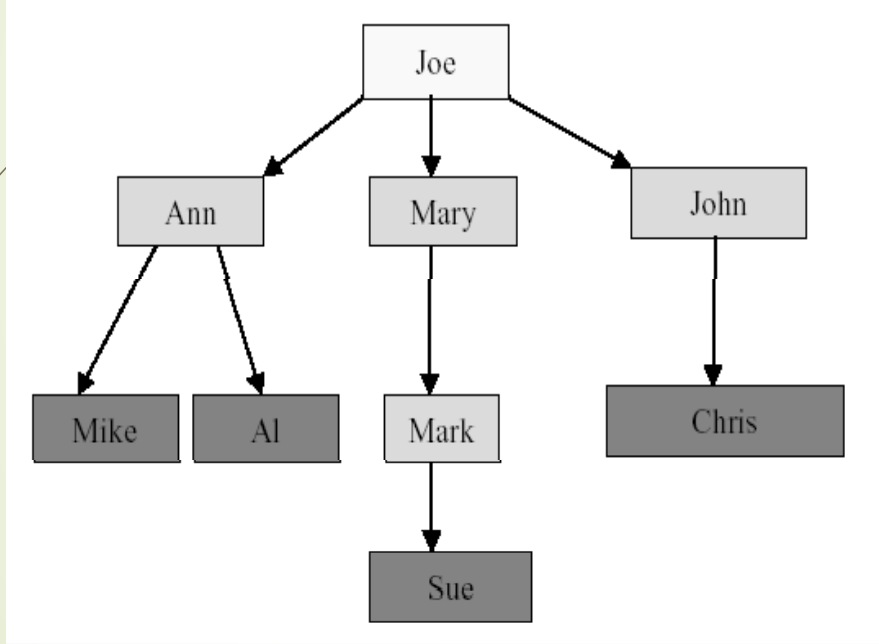
Tree Terminology

- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the root, and so on.
- Elements at the lowest level of the hierarchy are the **leaves**.



Other Definitions

- Leaves, Parent, Grandparent, Siblings, Ancestors, Descendents



Leaves = {Mike,Al,Sue,Chris}

Parent(Mary) = Joe

Grandparent(Sue) = Mary

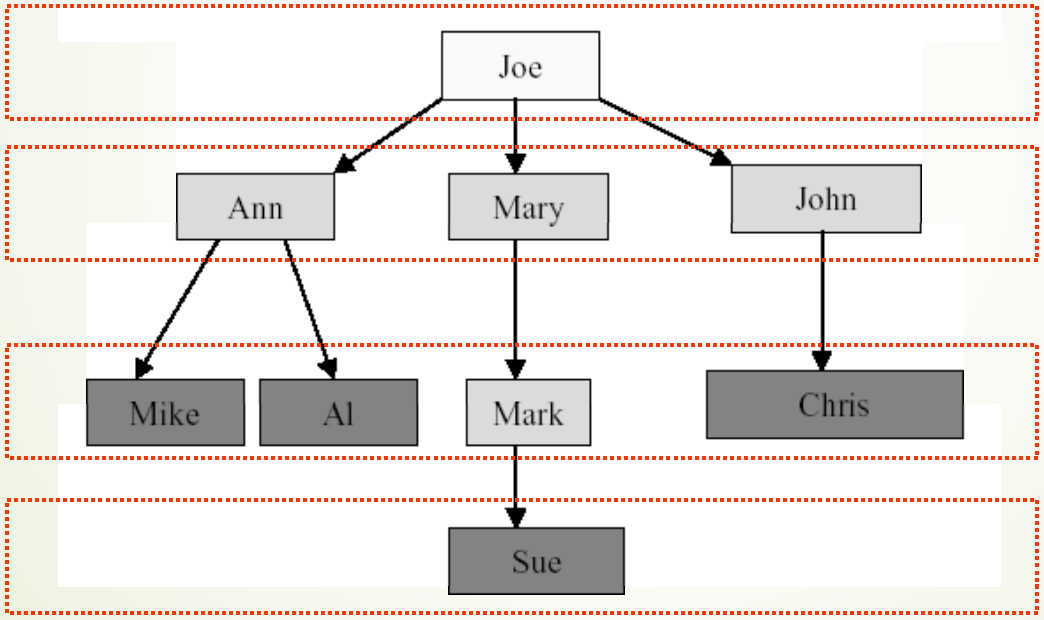
Siblings(Mary) = {Ann,John}

Ancestors(Mike) = {Ann,Joe}

Descendents(Mary)={Mark,Sue}

Levels

- Root is at level 0 and its children are at level 1.

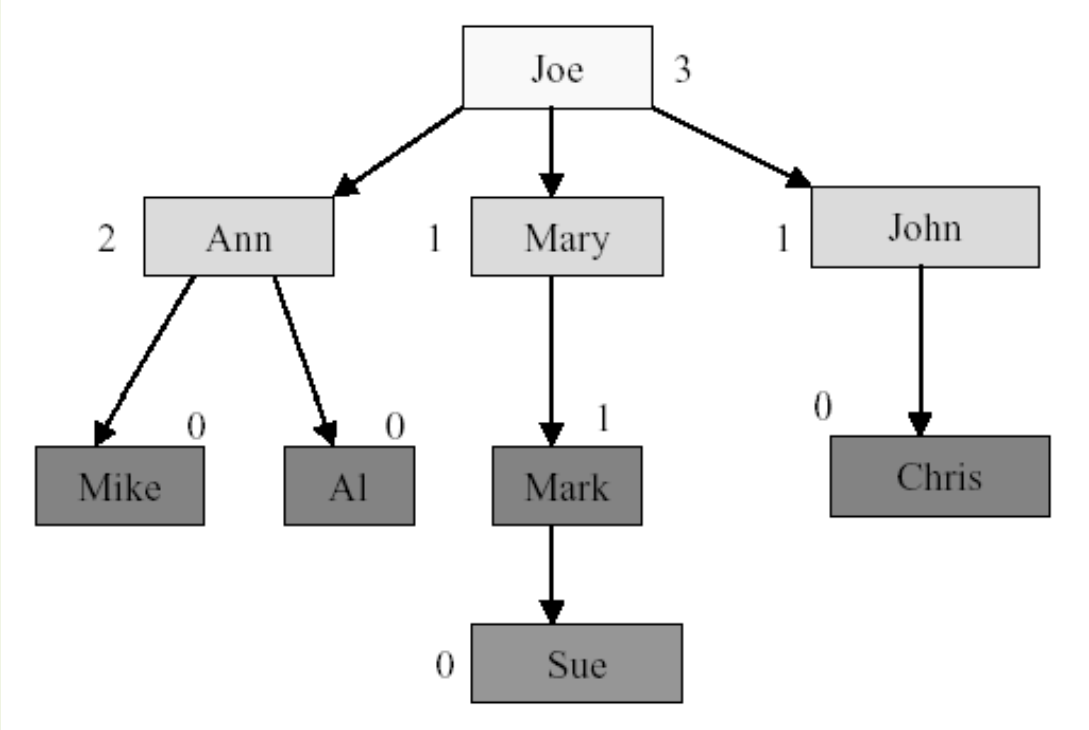


Some Key Terms

- 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.

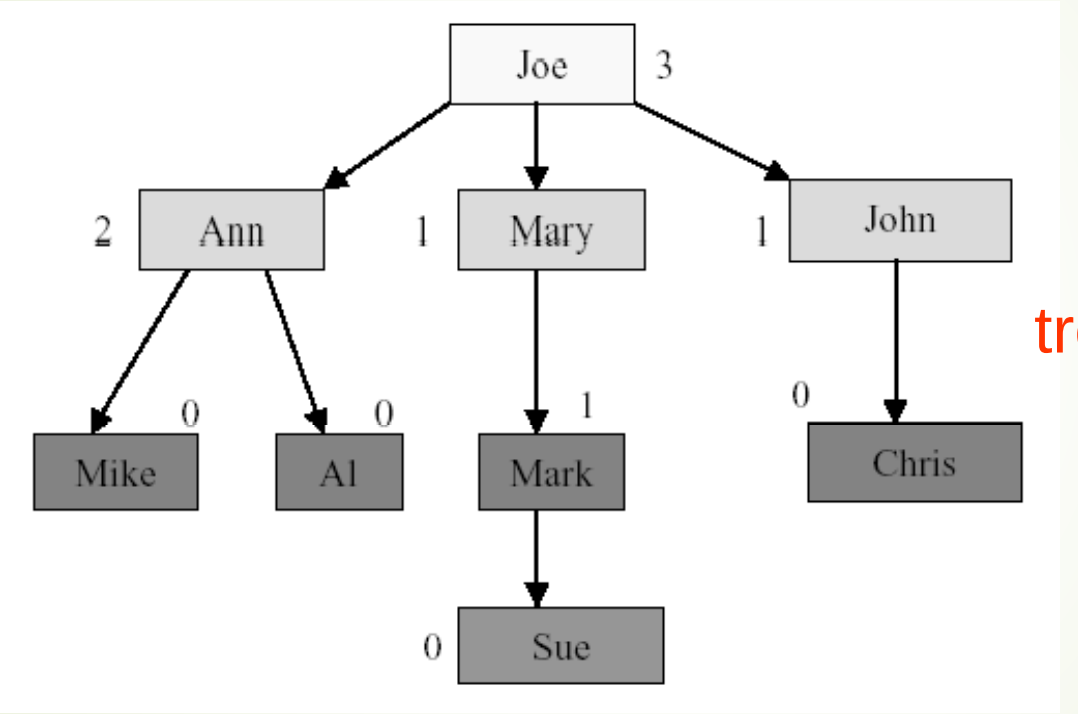
Node Degree

- Node degree is the number of children it has

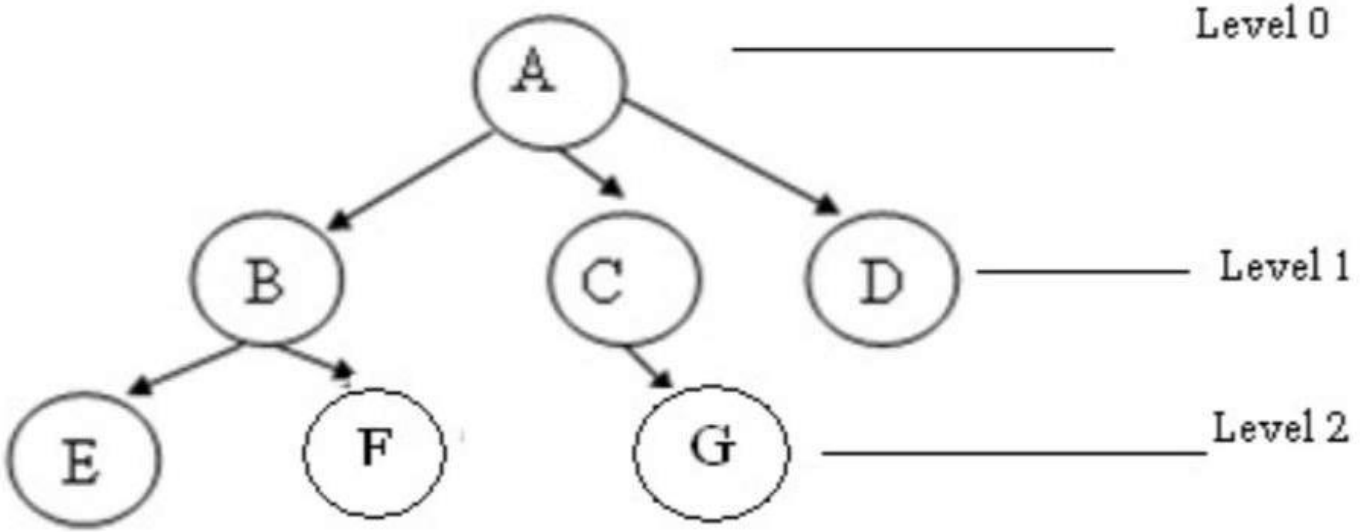


Tree Degree

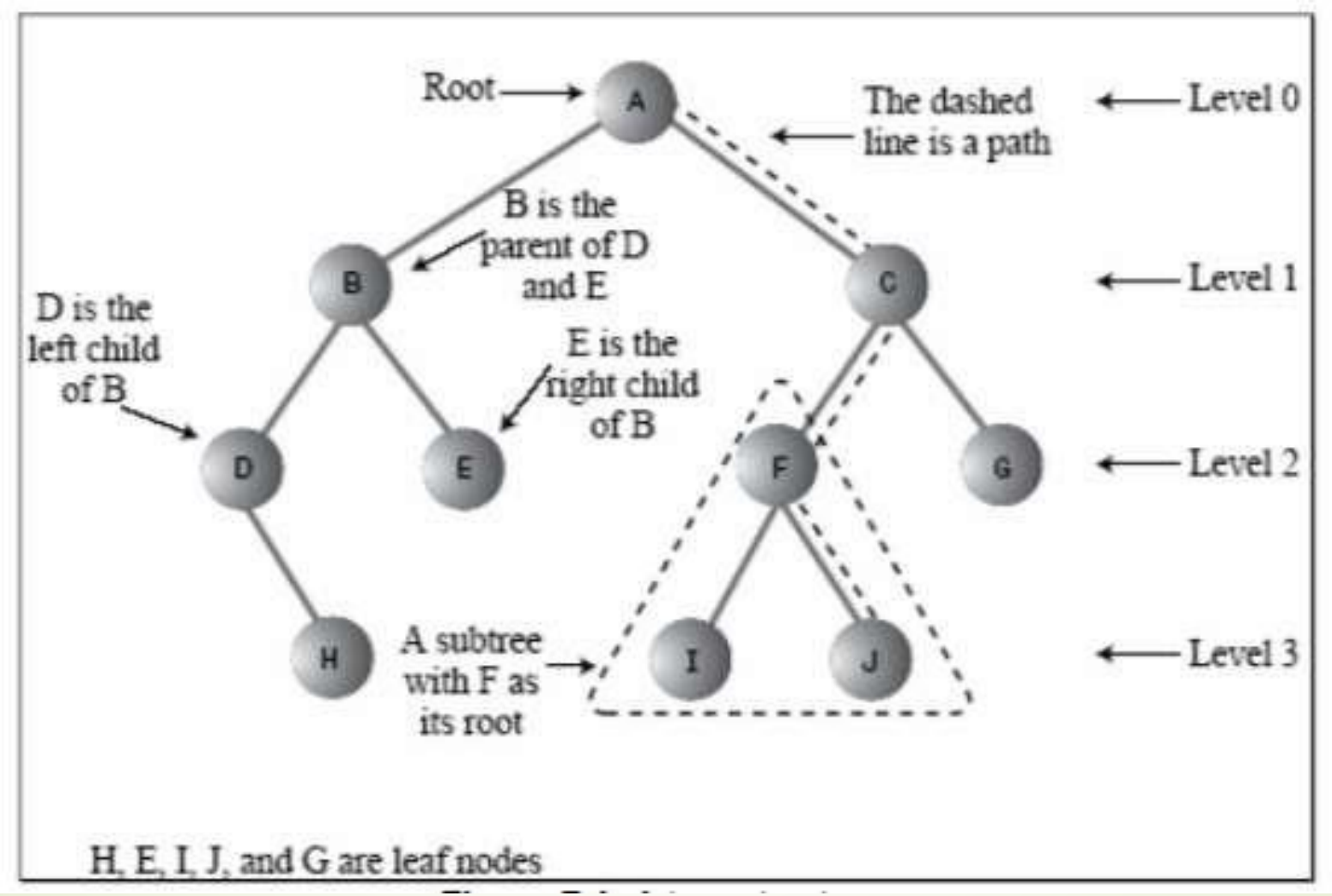
- Tree degree is the maximum of node degrees



tree degree = 3



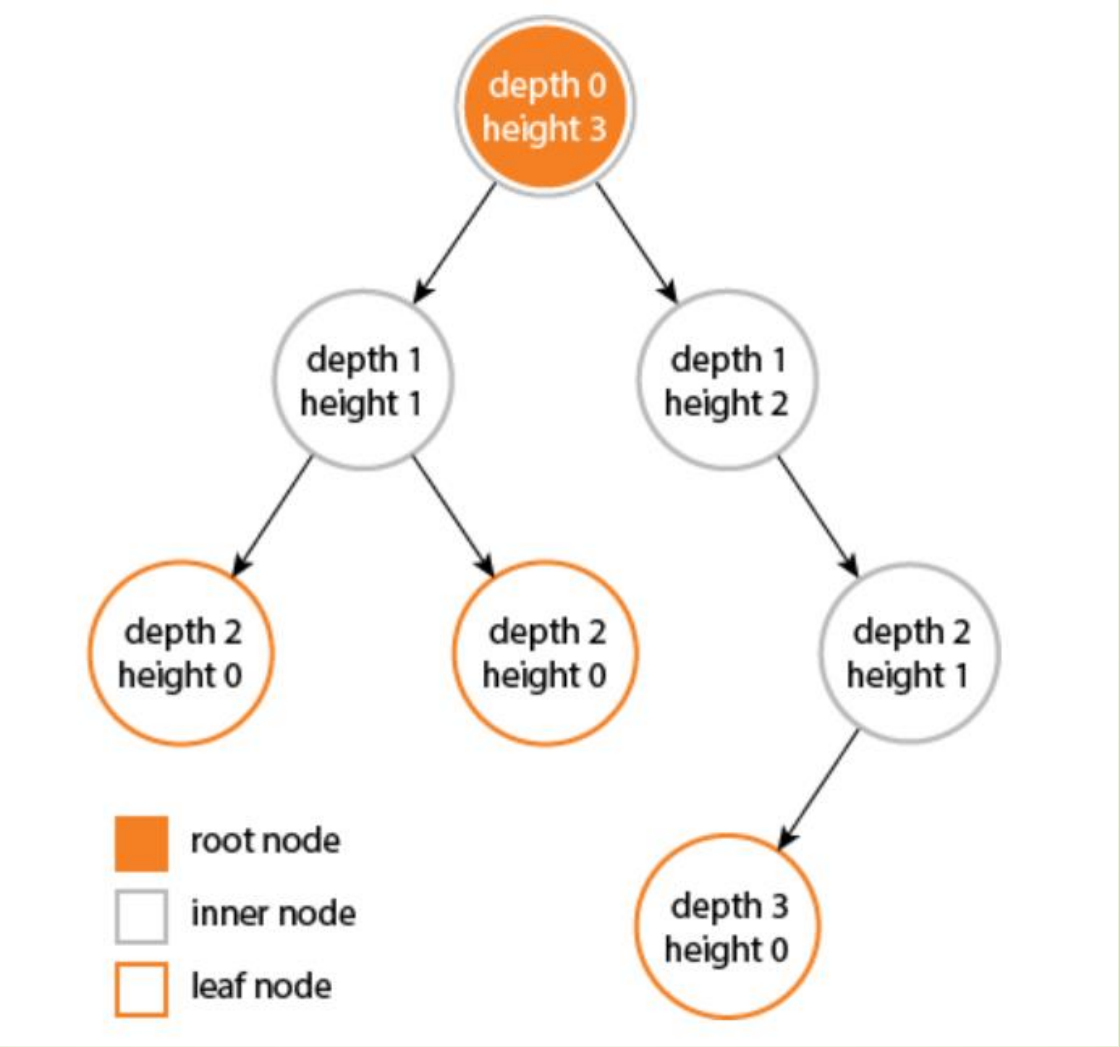
- ✔ A is the root node
- ✔ B is the parent of E and F
- ✔ D is the sibling of B and C
- ✔ E and F are children of B
- ✔ E, F, G, D are external nodes or leaves
- ✔ A, B, C are internal nodes
- ✔ Depth of F is 2
- ✔ the height of tree is 2
- ✔ the degree of node A is 3
- ✔ The degree of tree is 3


Root
A
The dashed line is a path
Level 0
B is the parent of D and E
B
C
Level 1
D is the left child of B
E is the right child of B
D
E
F
G
Level 2
H
A subtree with F as its root
I
J
Level 3
H, E, I, J, and G are leaf nodes

Some Key Terms

- 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 node:
  - the number of edges from the node to the tree's root node (level of node)
- Depth of a tree:
  - Depth of a tree is the max level of any leaf in the tree

Height and Depth



Application

- 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.

Introduction To Binary Trees

- A binary tree, is a tree in which no node can have more than two children.
- Consider a binary tree T, here ‘A’ is the root node of the binary tree T.
- ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’
  - i.e A is a father of B and C.
  - The node B and C are called siblings.
- Nodes D,H,I,F,J are leaf node

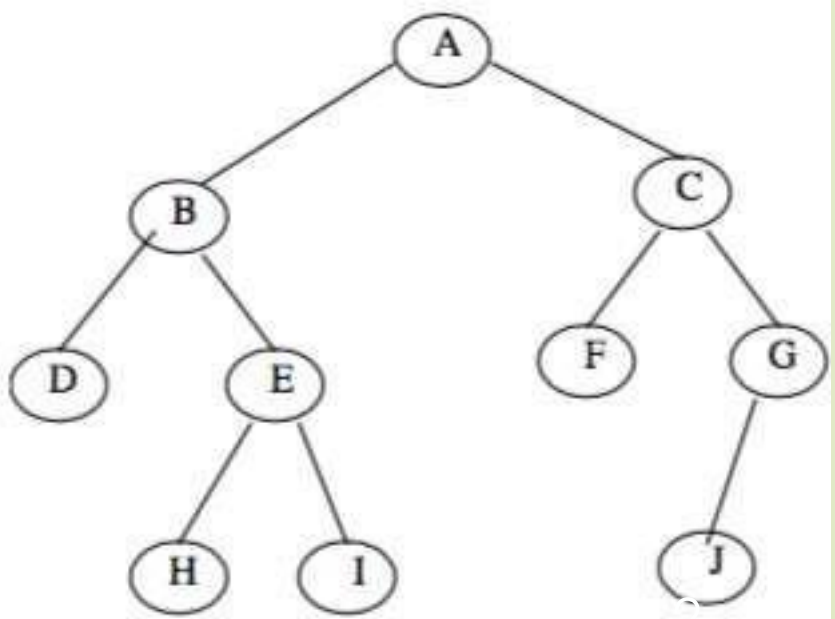
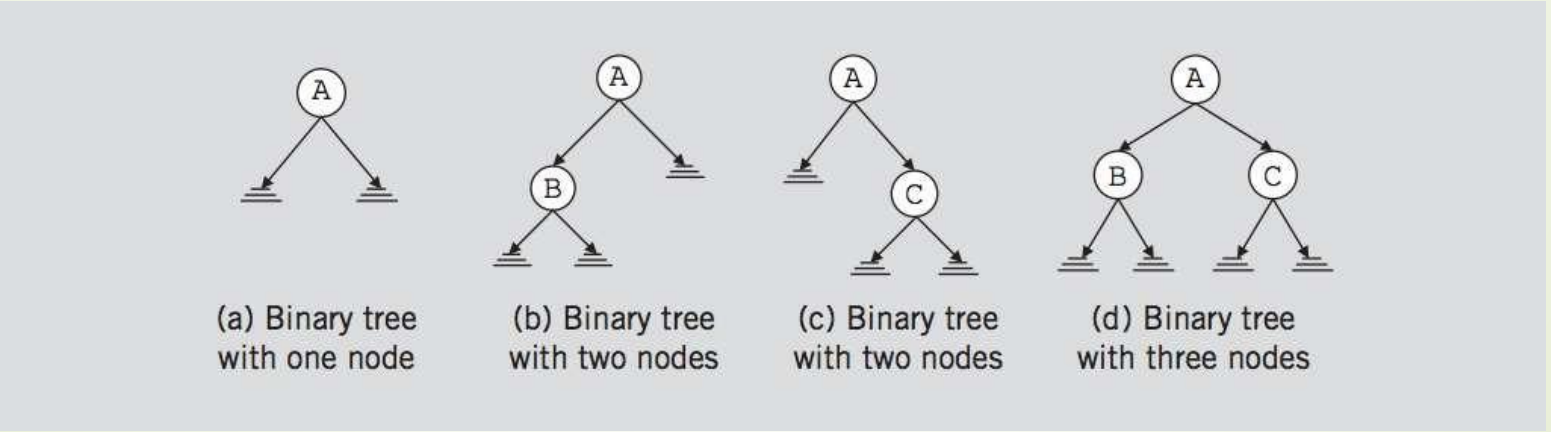


Fig. 8.3. Binary tree

Binary Trees

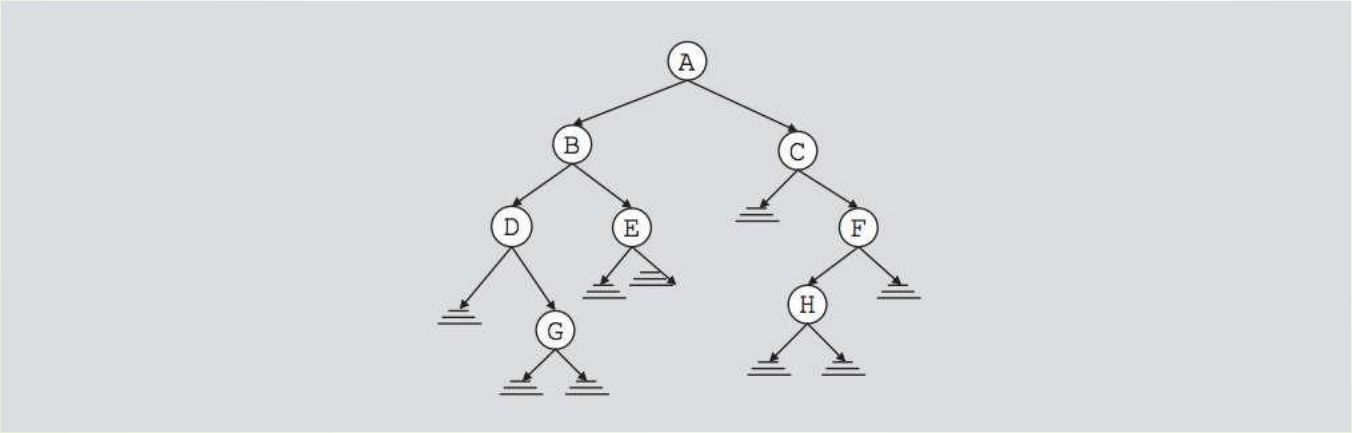
- A binary tree, *T*, is either empty or such that
  - *I.* *T* has a special node called the root node
  - *II.* *T* has two sets of nodes L_T and R_T, called the left subtree and right subtree of *T*, respectively.
  - *III.* L_T and R_T are binary trees.



Binary Tree

- The root node of this binary tree is A.
- The left sub tree of the root node, which we denoted by L_A, is the set L_A = {B,D,E,G} and the right sub tree of the root node, R_A is the set R_A={C,F,H}

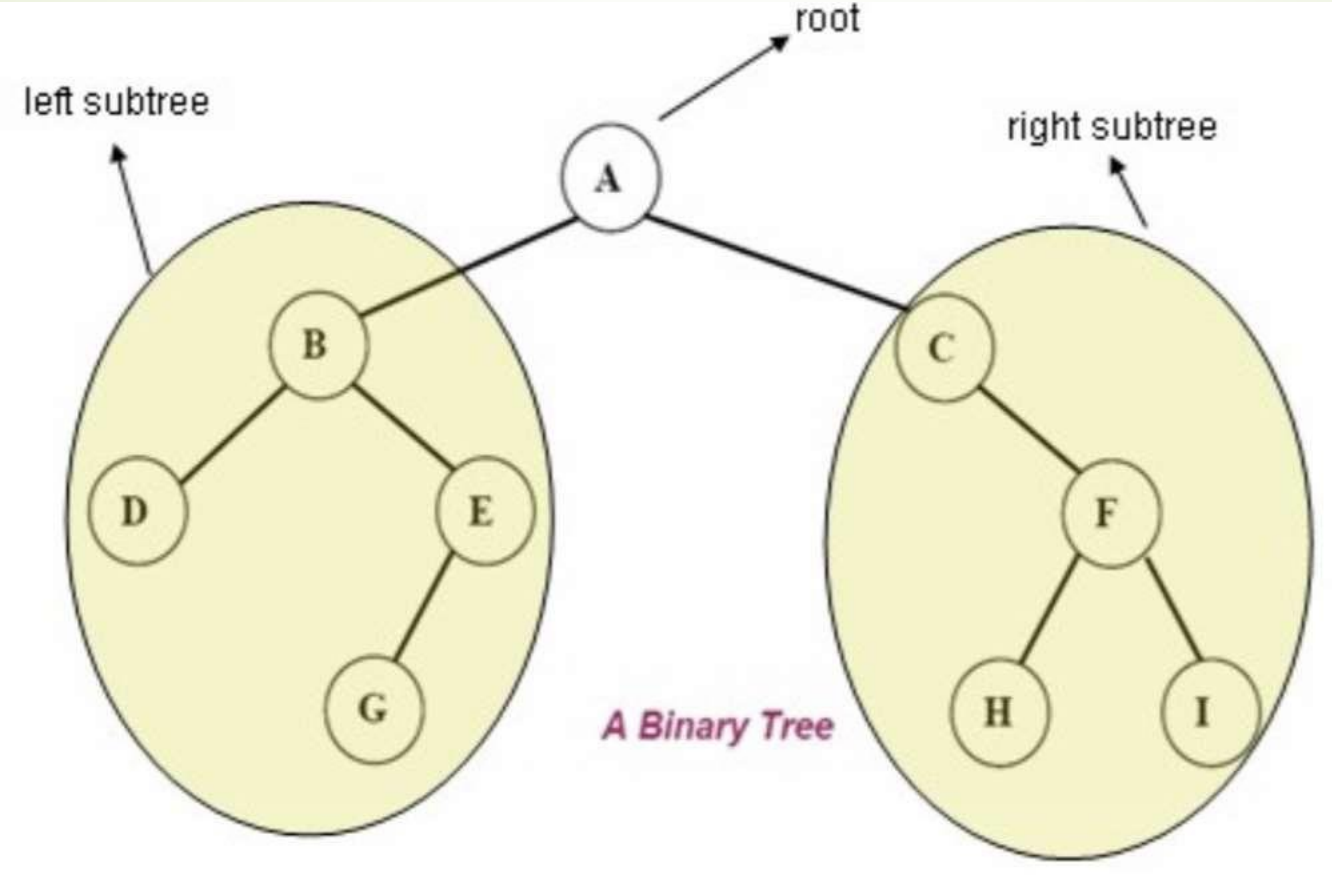
- The root node of L_A is node B, the root node of R_A is C and so on



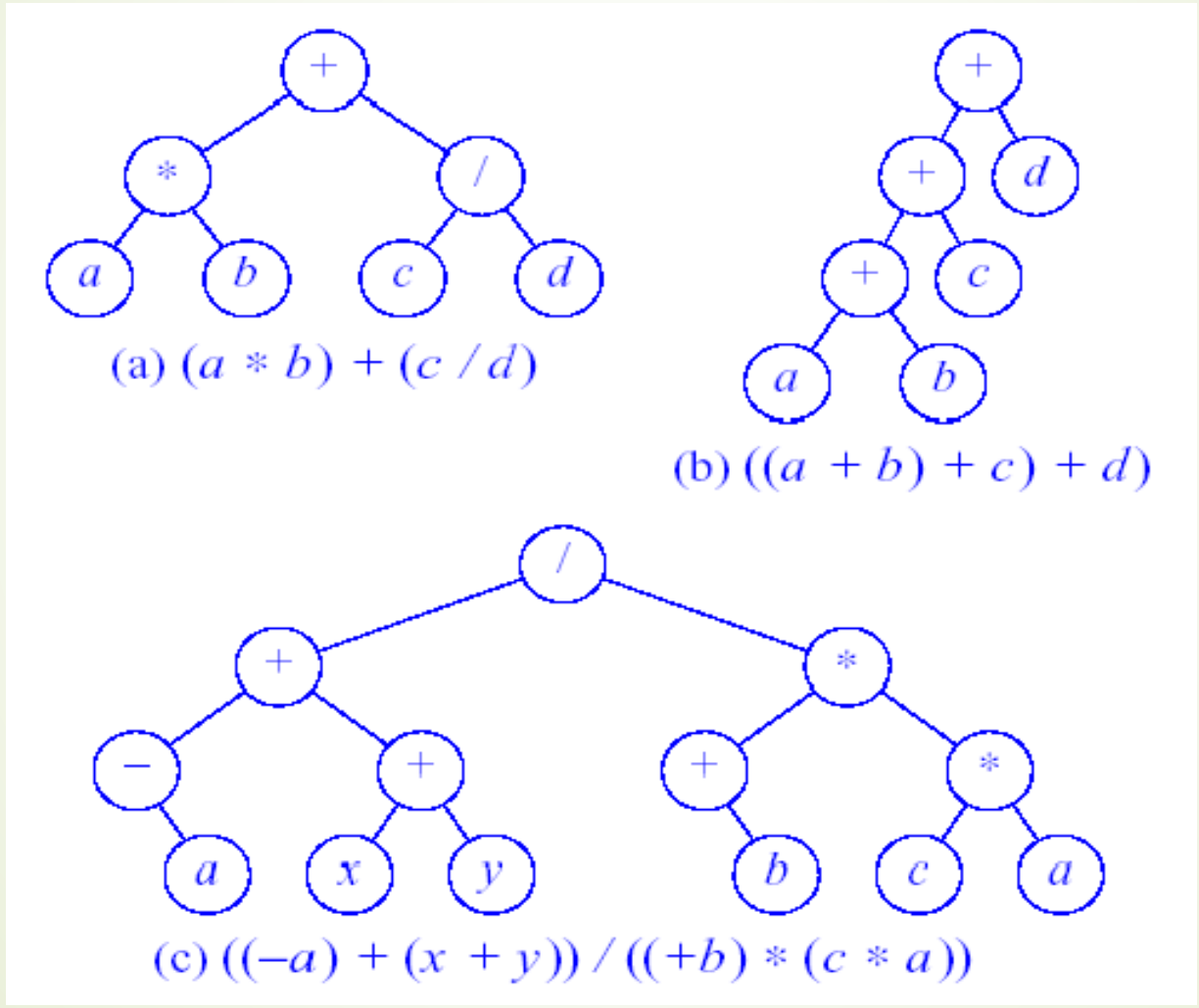
Binary Tree

- A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets.
- The first subset contains a single element called the root of the tree.
- The other two subsets are themselves binary trees called the left and right sub-trees of the original tree.
- A left or right sub-tree can be empty.
- All the nodes in a binary tree have 0, 1 or 2 child/children.
- Each element of a binary tree is called a node of the tree.

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



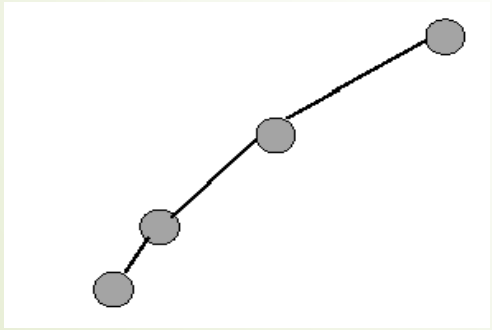
Binary Tree for Algebraic Expressions



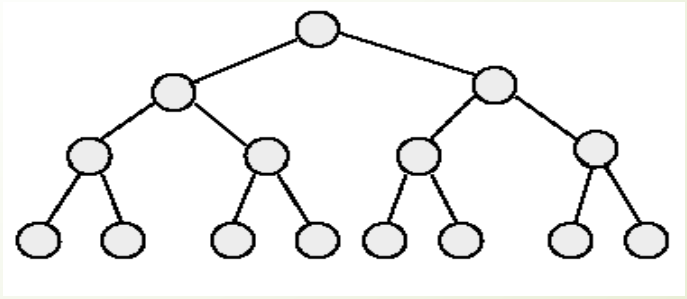
Binary Tree Properties

1. Every binary tree with n nodes, $n > 0$, has exactly n-1 edges.

2. A binary tree of height h, $h >= 0$, has at least $h+1$ and at most 2^{h+1}-1 nodes in it.



minimum number of nodes



maximum number of nodes

Binary Tree Properties

3. The height of a binary tree that contains n nodes, $n >= 0$, is at least $\lceil (log_2(n+1)) \rceil - 1$ and at most $n-1$.

4. For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof:

$n = n0 + n1 + n2$, $B + 1 = n$, $B = n1 + 2n2$ ==>

$n1 + 2n2 + 1 = n$,

$n1 + 2n2 + 1 = n0 + n1 + n2$ ==> $n0 = n2 + 1$

Binary Tree Properties

- If a binary tree contains m nodes at level L, it contains at most 2m nodes at level L+1

- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most 2^L nodes at level L.

Types of Binary Tree

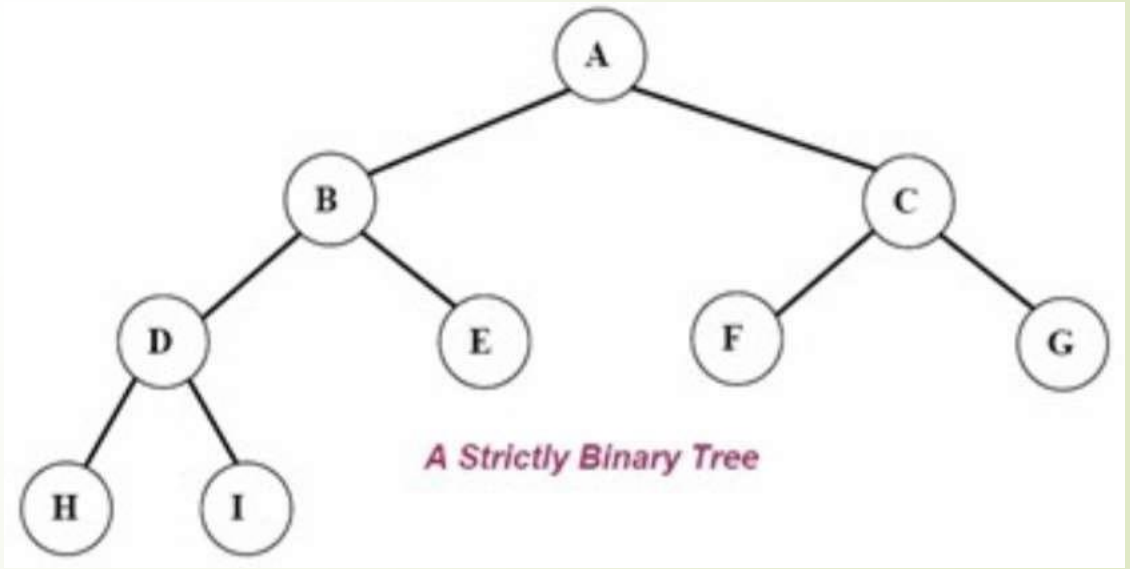
- Strictly binary tree (2-tree or extended binary tree)
- Complete (or full) binary tree
- Almost complete binary tree

Strictly binary tree (2-tree or extended 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.

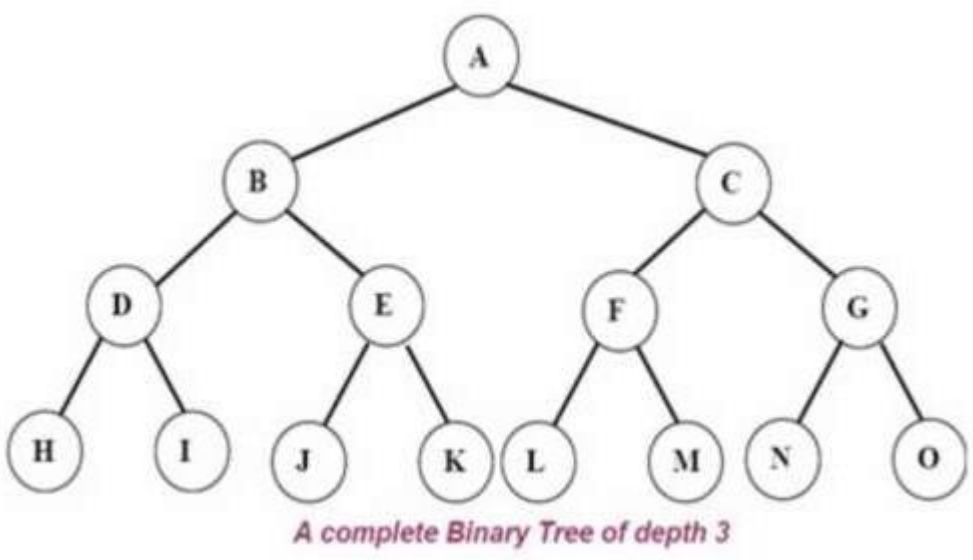
- Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.

- A strictly binary tree with
  N leaves always contains 2N – 1 nodes.
  Proof:
  $B= 2n_2$, $B+1=n$, $n=n_0+n_2$
  $\Rightarrow 2n_2+1=n_0+n_2$
  $\Rightarrow n_2=n_0-1$
  $\Rightarrow n=2n_0-1$



Complete (or full) binary tree

- At each level, there are maximum number of nodes.
- A complete binary tree has 2^L nodes at every level L.
- There are 2^L -1 non leaf nodes in a complete binary tree of depth L.



A complete Binary Tree of depth 3

Almost complete binary tree

- An almost complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes at last level are as far left as possible.

- For a leaf right child, there is always a left sibling, but for a leaf left child there may not be a right child.

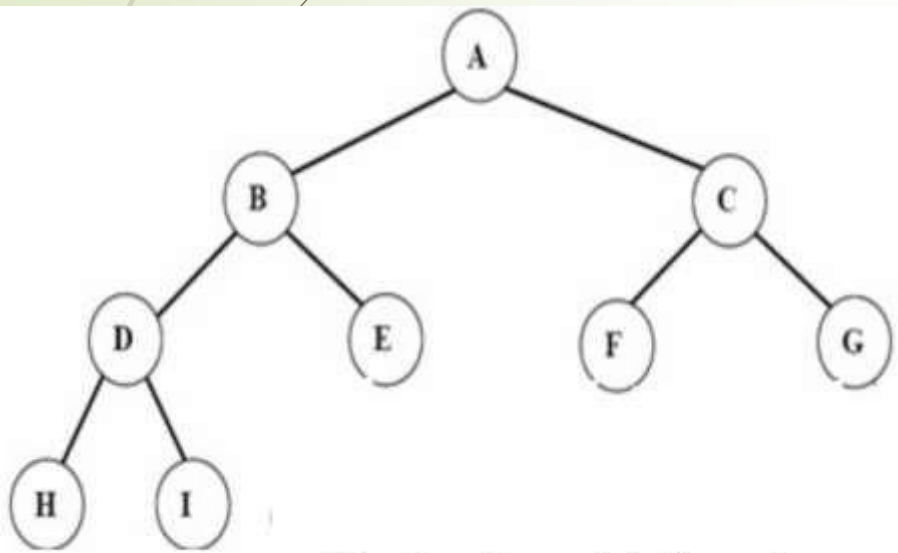


Fig Almost complete binary tree.

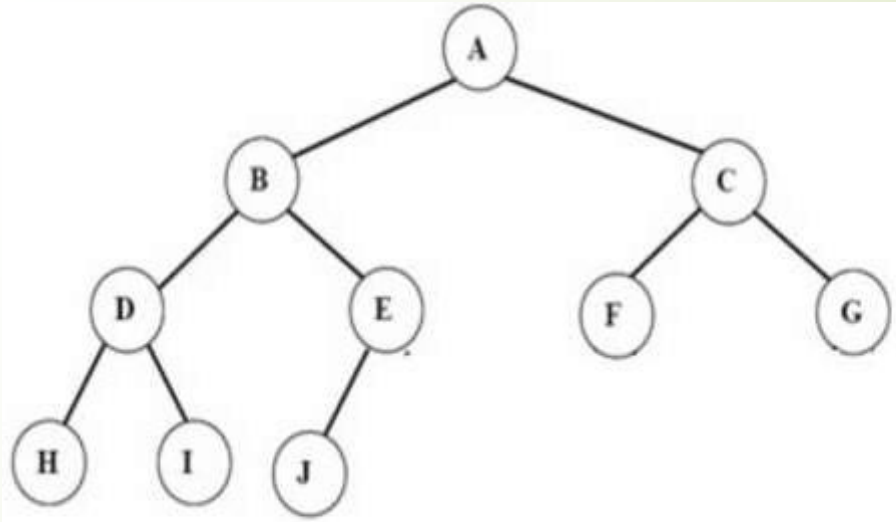


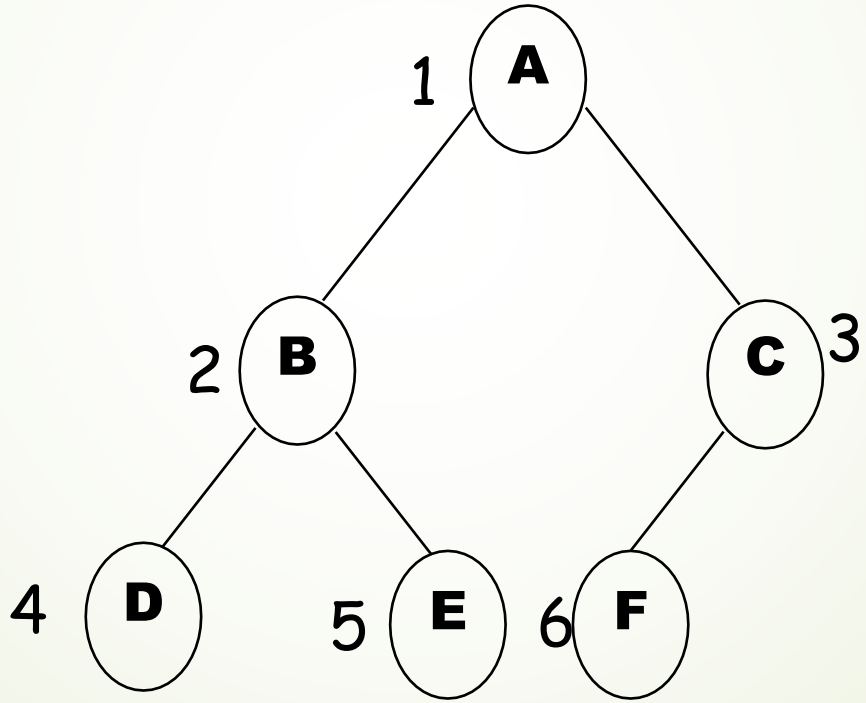
Fig Almost complete binary tree but not strictly binary tree.
Since node E has a left son but not a right son.

A (almost) complete binary tree obeys the following properties with regard to its node numbering:

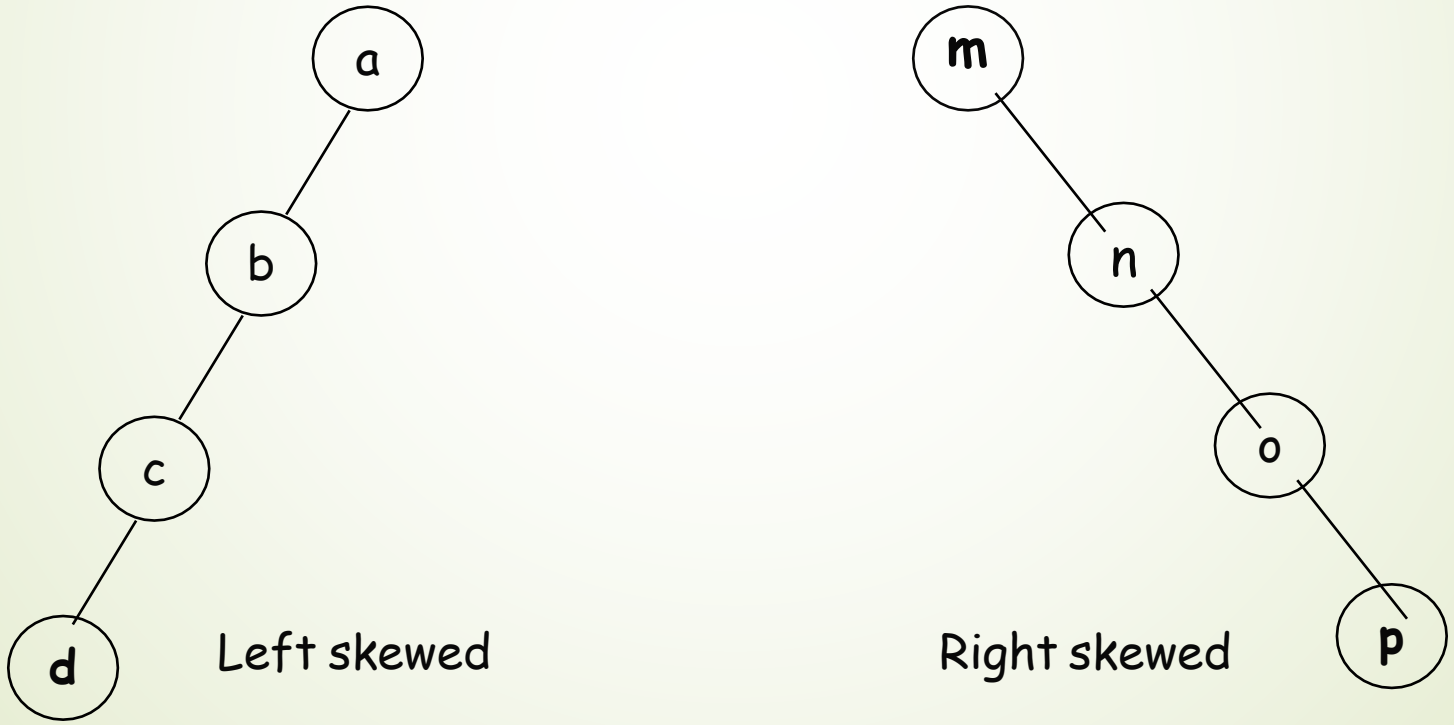
a. If a parent node has a number i then its left child has the number 2i ($2i \leq n$). If $2i > n$ then i has no left child.

b. If a parent node has a number i, then its right child has the number 2i+1 ($2i + 1 \leq n$). If $2i + 1 > n$ then i has no right child.

[c] If a child node (left or right) has a number i then the parent node has the number $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$ then i is the root and hence has no parent.



A binary tree which is dominated solely by left child nodes or right child nodes is called a **skewed binary tree** or more specifically **left skewed binary tree** or **right skewed binary tree** respectively.

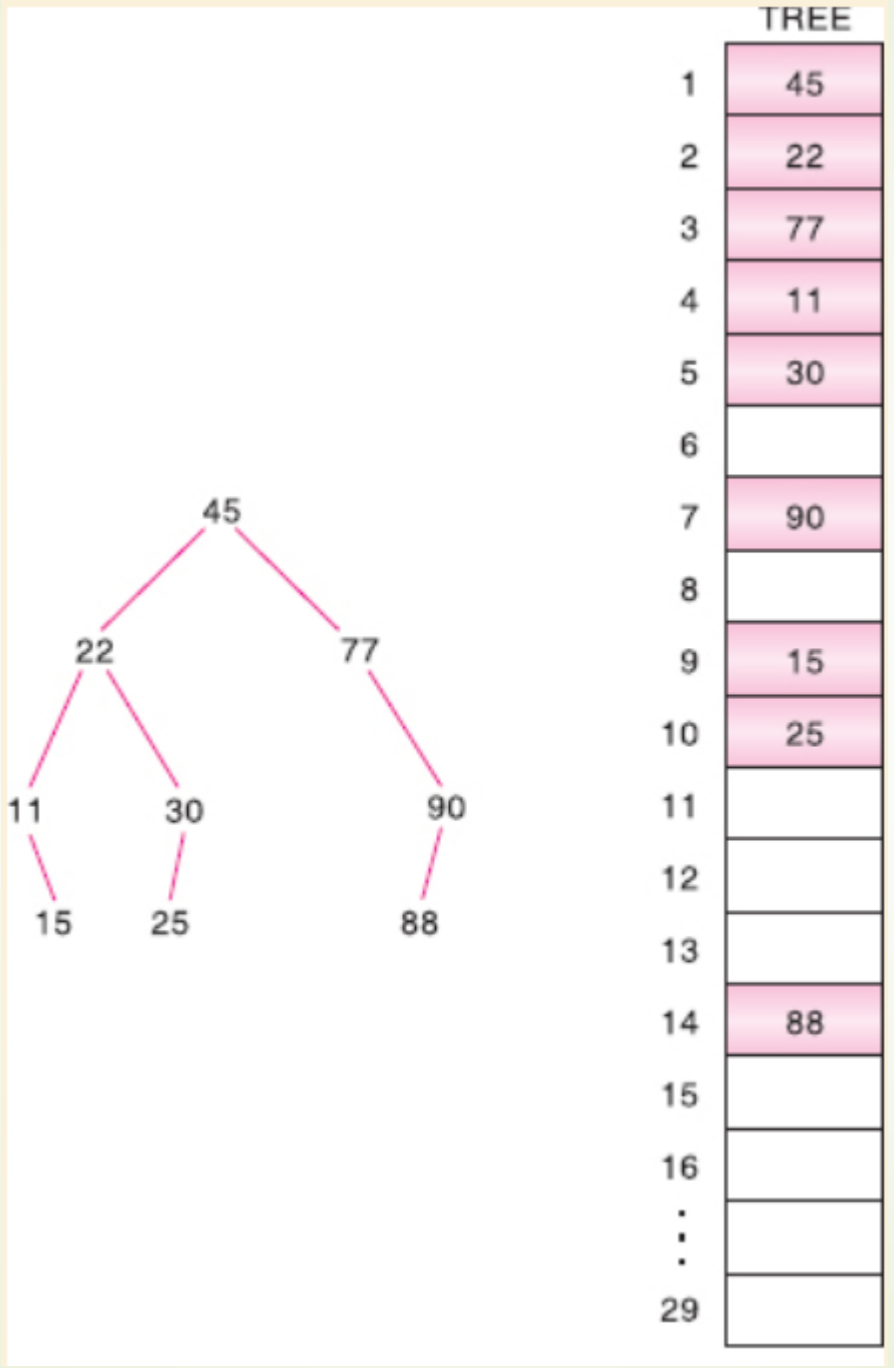


Operations on Binary tree:

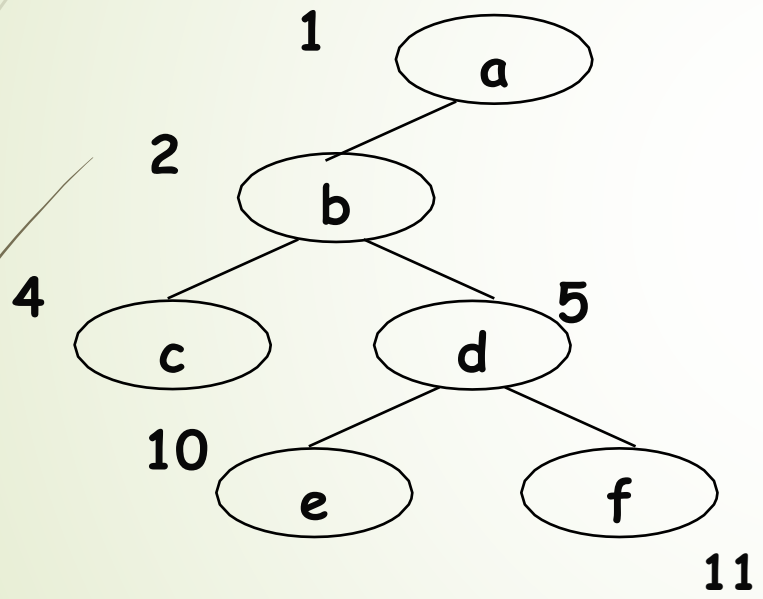
- **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild(n,T)**:Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- **Sibling(n,T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.
- **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- **Size(T):** Return the number of nodes in tree T
- **MakeEmpty(T):** Create an empty tree T
- **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder(T):** Traverses all the nodes of tree T in preorder.
- **postorder(T):** Traverses all the nodes of tree T in postorder
- **Inorder(T):** Traverses all the nodes of tree T in inorder.

Sequential Representations

- If a complete binary tree with n nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index i, $1 <= i <= n$, we have:
  - *parent*(i) is at $\lfloor i/2 \rfloor$ if $i!=1$. If $i=1$, i is at the root and
    has no parent.
  - *left_child*(i) ia at $2i$ if $2i <= n$. If $2i>n$, then i has no left child.
  - *right_child*(i) ia at $2i+1$ if $2i+1 <= n$. If $2i+1 > n$, then i has no right child.


45
22
77
11
30
90
15
25
88
TREE
1 45
2 22
3 77
4 11
5 30
6
7 90
8
9 15
10 25
11
12
13
14 88
15
16
29

Sequential Representation

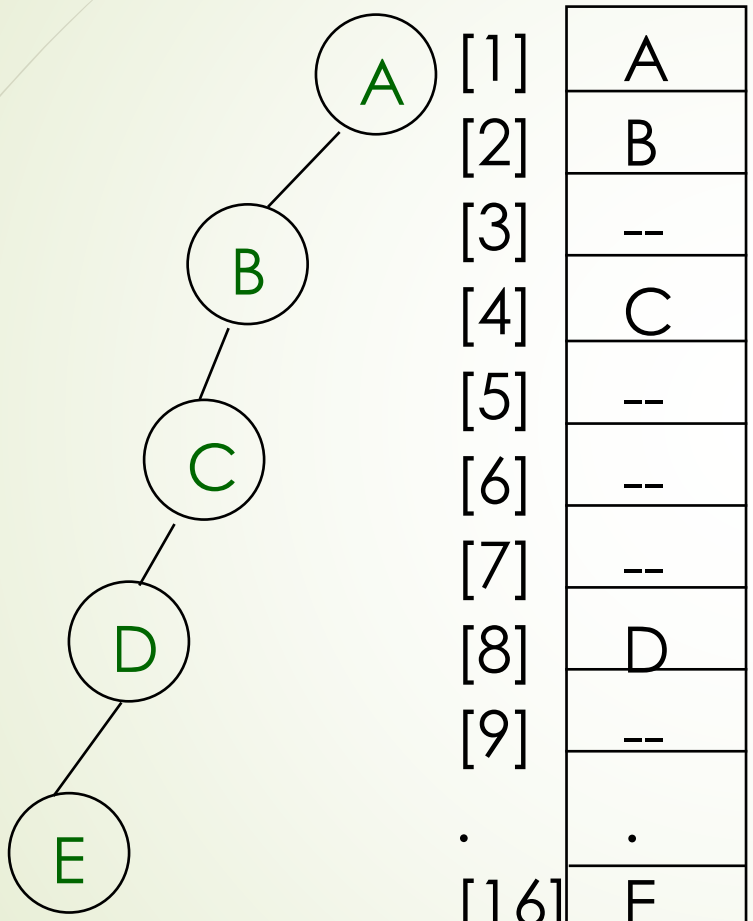


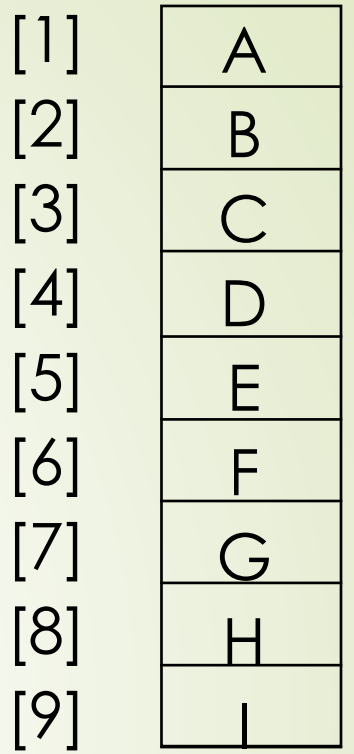
- Sequential representation of a tree with depth **d** will require an array with **$2^{(d+1)}-1$ cells**

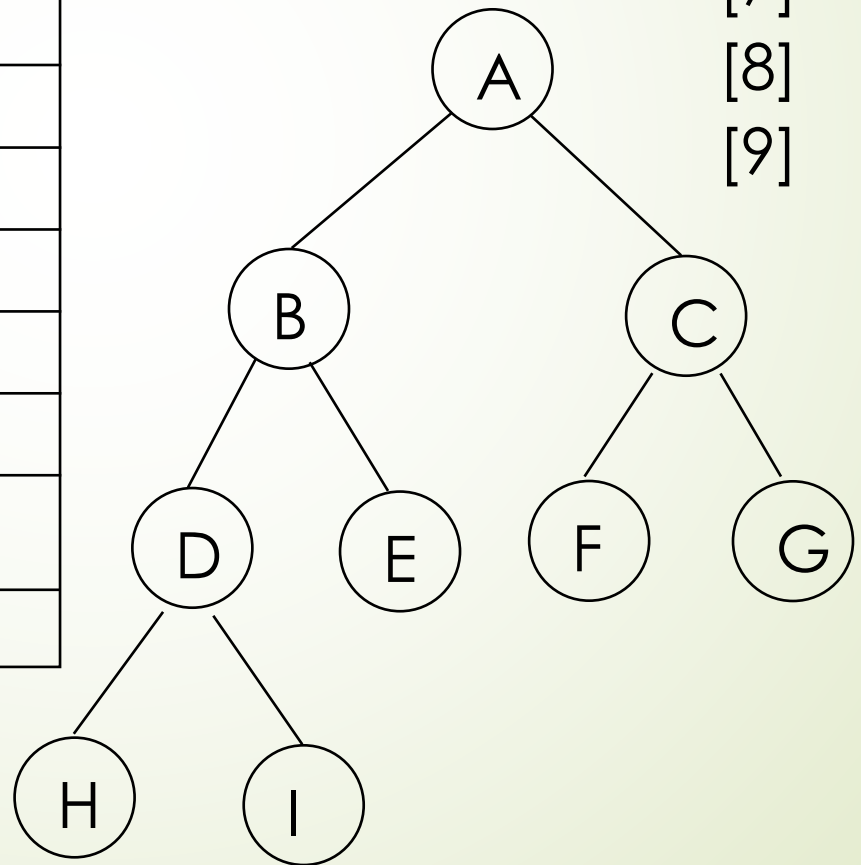
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b		c	d					e	f				

Sequential Representation

(1) waste space





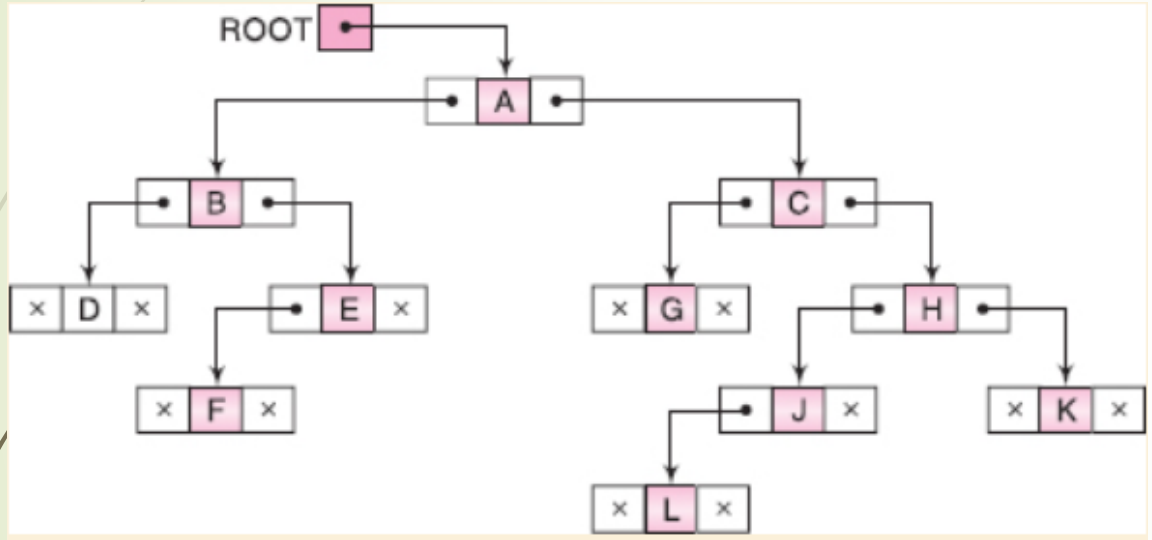


Linked Representation

- Three arrays INFO, LEFT and RIGHT
- Each node N in tree corresponds to a location K

INFO[K] contains the data at the node N.
LEFT[K] contains the location of the left child of node N.
RIGHT[K] contains the location of the right child of node N.

- ROOT points to the root of binary tree



ROOT → 5

AVAIL → 8

	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

C representation for Binary tree:

```
struct bnode
{
        int info;
        struct bnode  *left;
        struct bnode  *right;
};
struct bnode *root=NUl
```

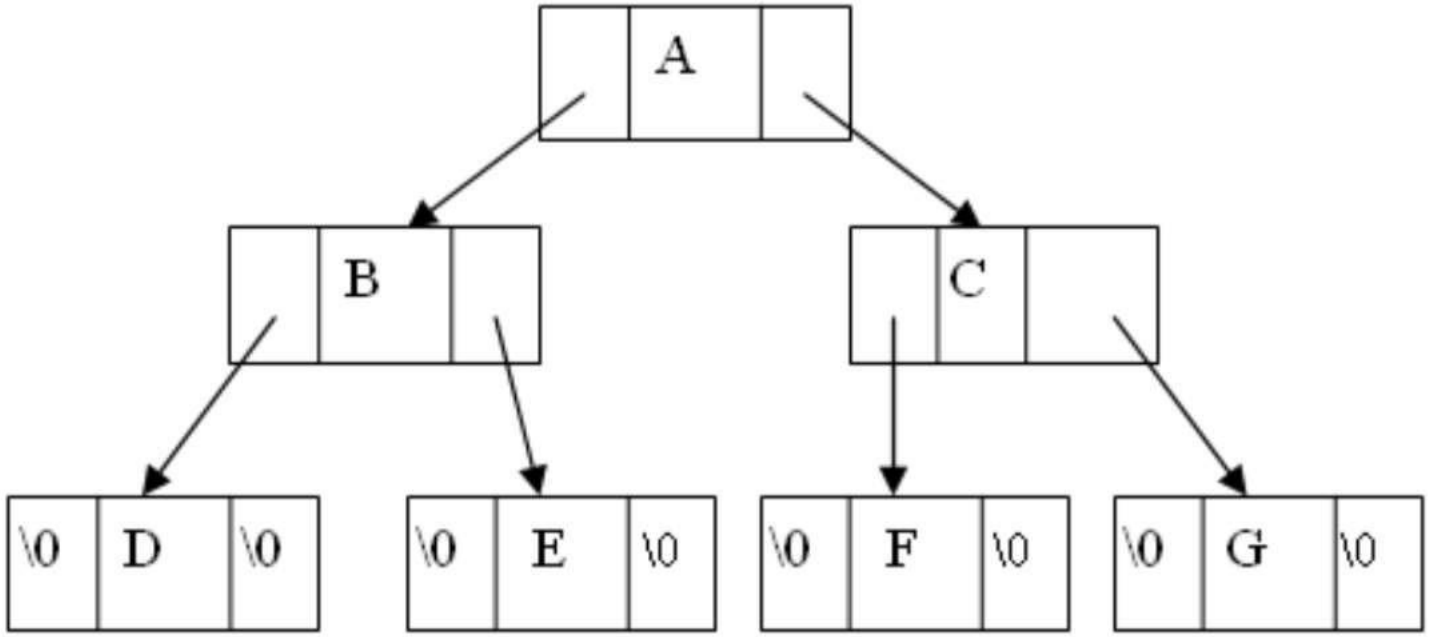


Fig: Structure of Binary tree

Tree traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- All nodes are connected via edges (links)
- There are three ways which we use to traverse a tree
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
- Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

Pre-order, In-order, Post-order

- Pre-order (NLR)

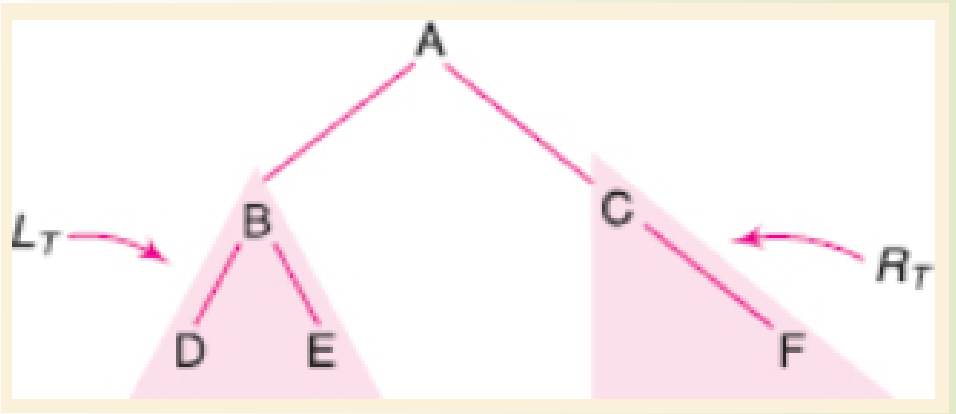
<root><left><right>

- In-order (LNR)

<left><root><right>

- Post-order (LRN)

<left><right><root>



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

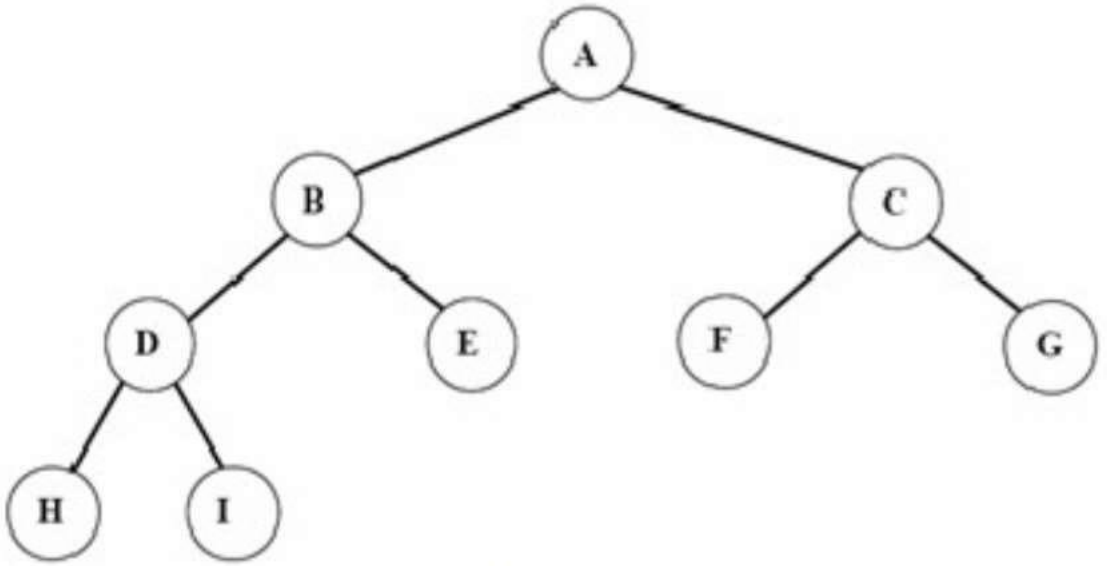


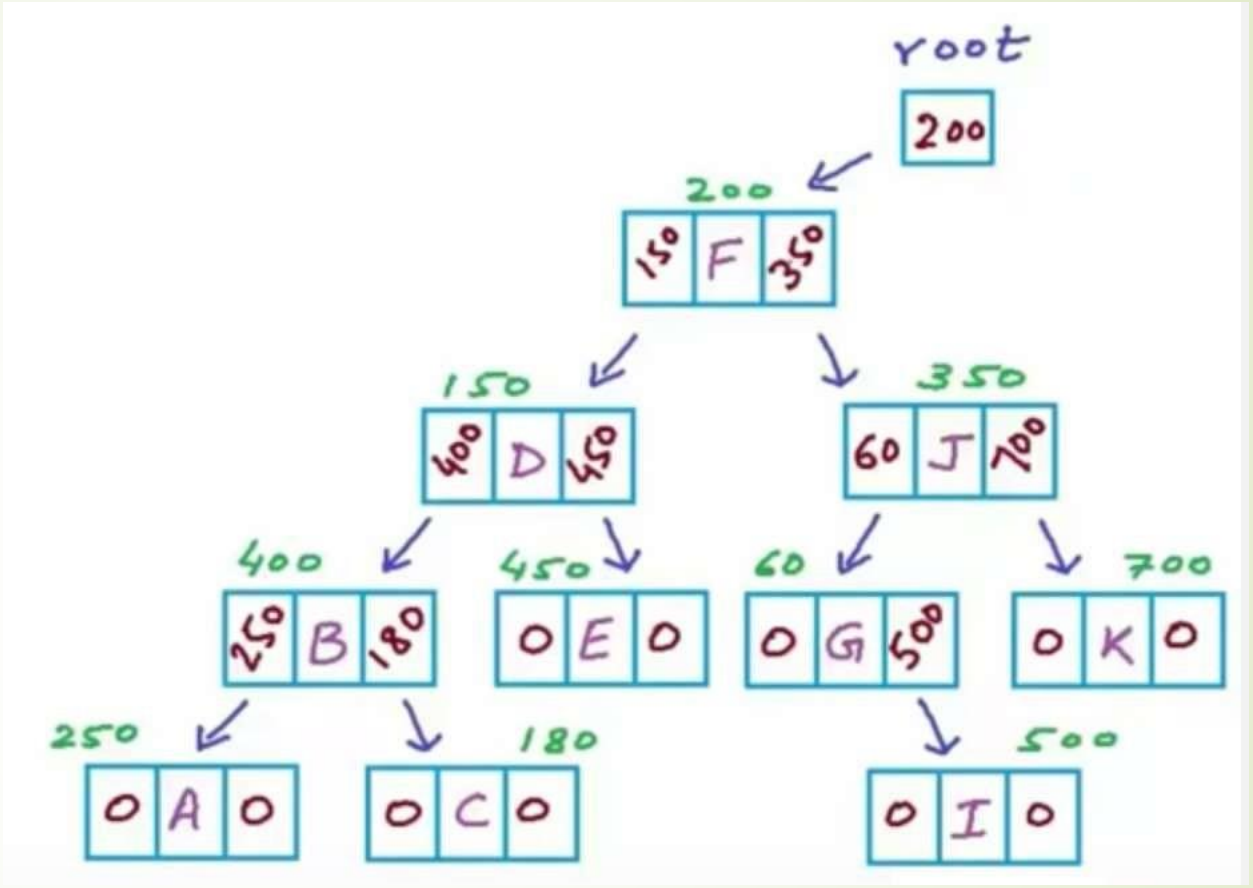
fig Binary tree

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

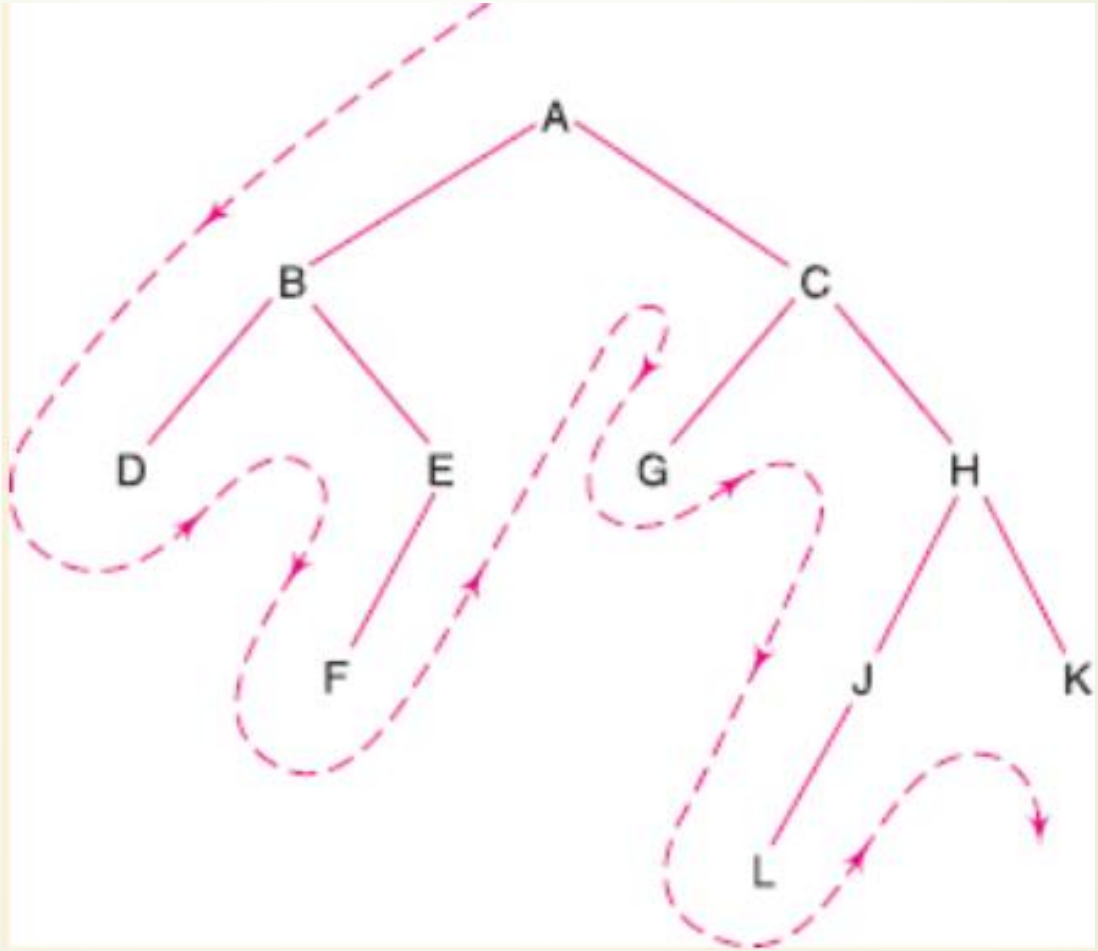
The preorder is also known as depth first order.

Pre-order Pseudocode

```
struct Node{
    char data;
    Node *left;
    Node *right;
}
void Preorder(Node *root)
{
    if (root==NULL) return;
    printf (“%c”, root->data);
    Preorder(root->left);
    Preorder(root->right);
}
```



Pre-order Trick



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

- The in-order traversal outpu
  of the given tree is
  H D I B E A F C G

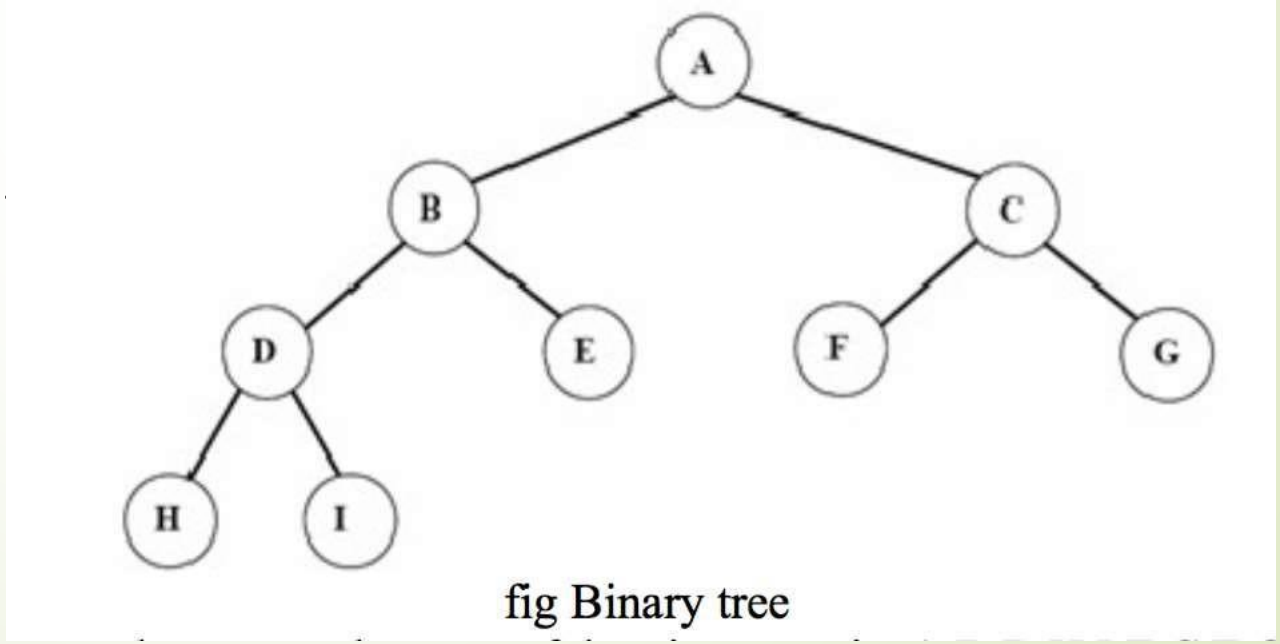
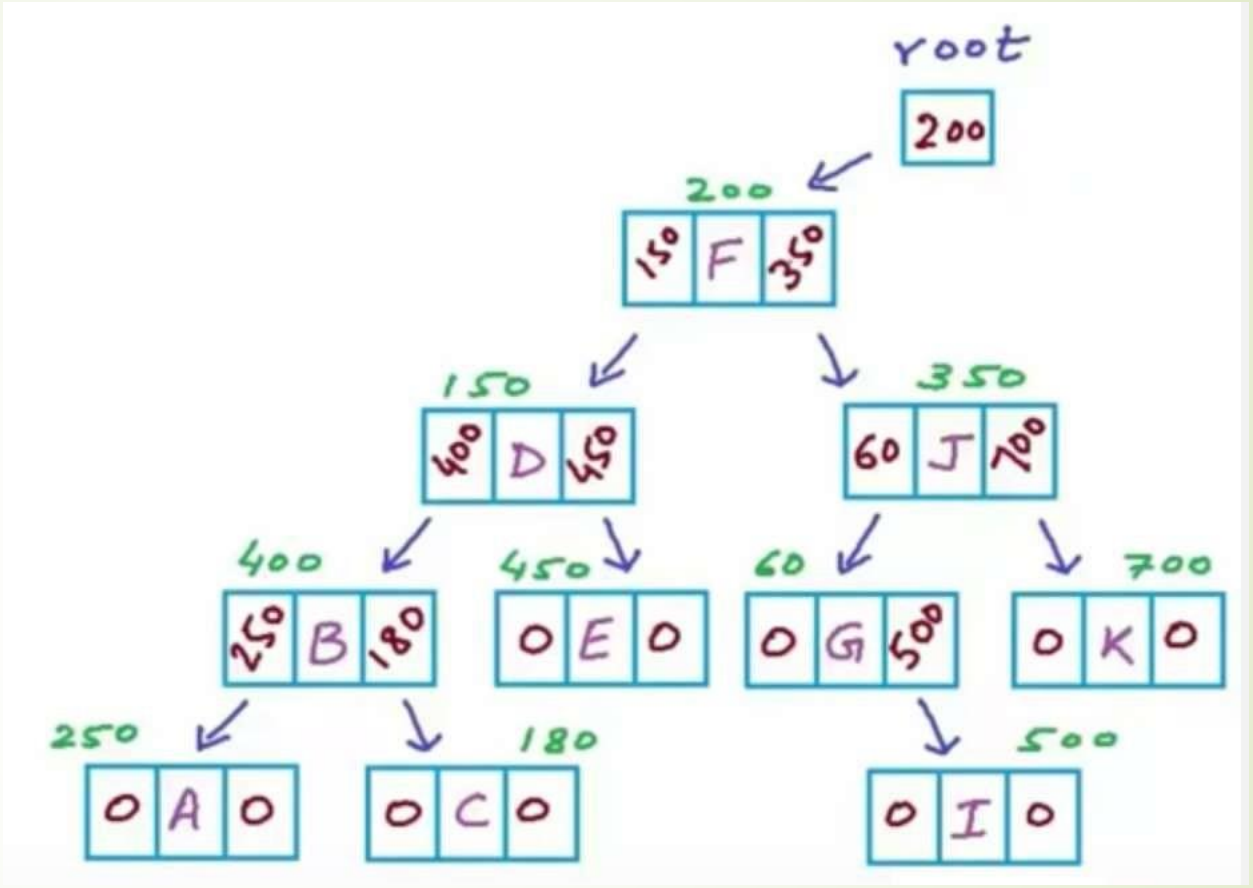


fig Binary tree

In-order Pseudocode

```
struct Node{
    char data;
    Node *left;
    Node *right;
}
void Inorder(Node *root)
{
    if (root==NULL) return;
    Inorder(root->left);
    printf ("%c", root->data);
    Inorder(root->right);
}
```



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

- The in-order traversal outpu
  of the given tree is
  H I D E B F G C A

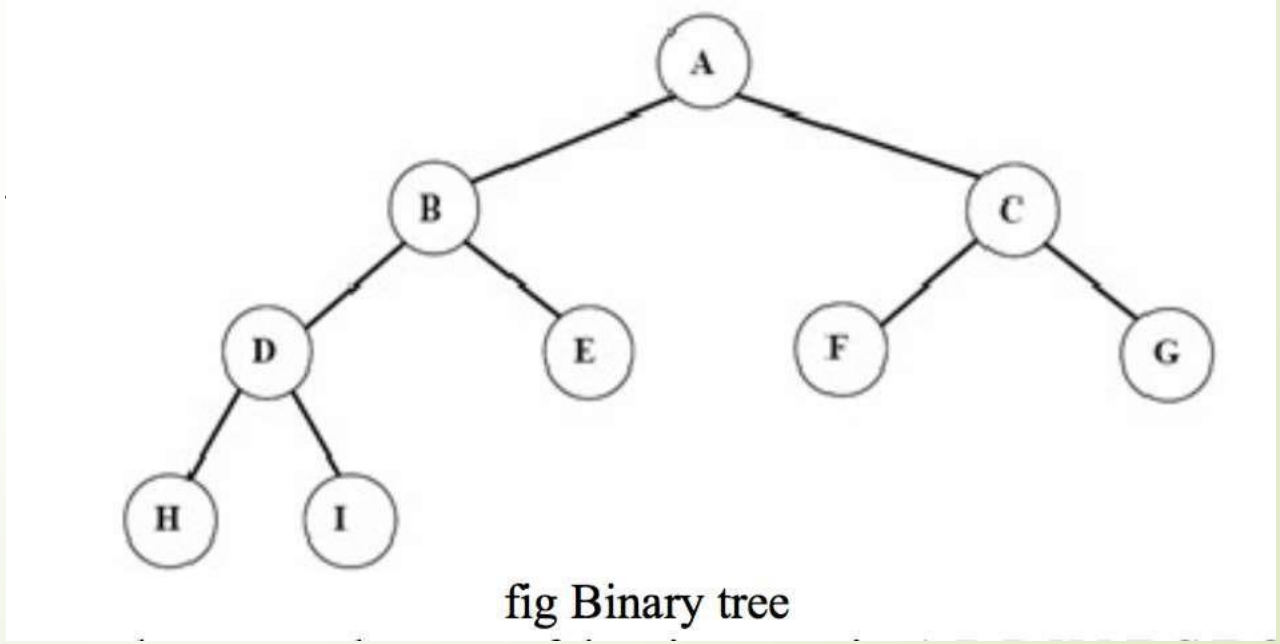
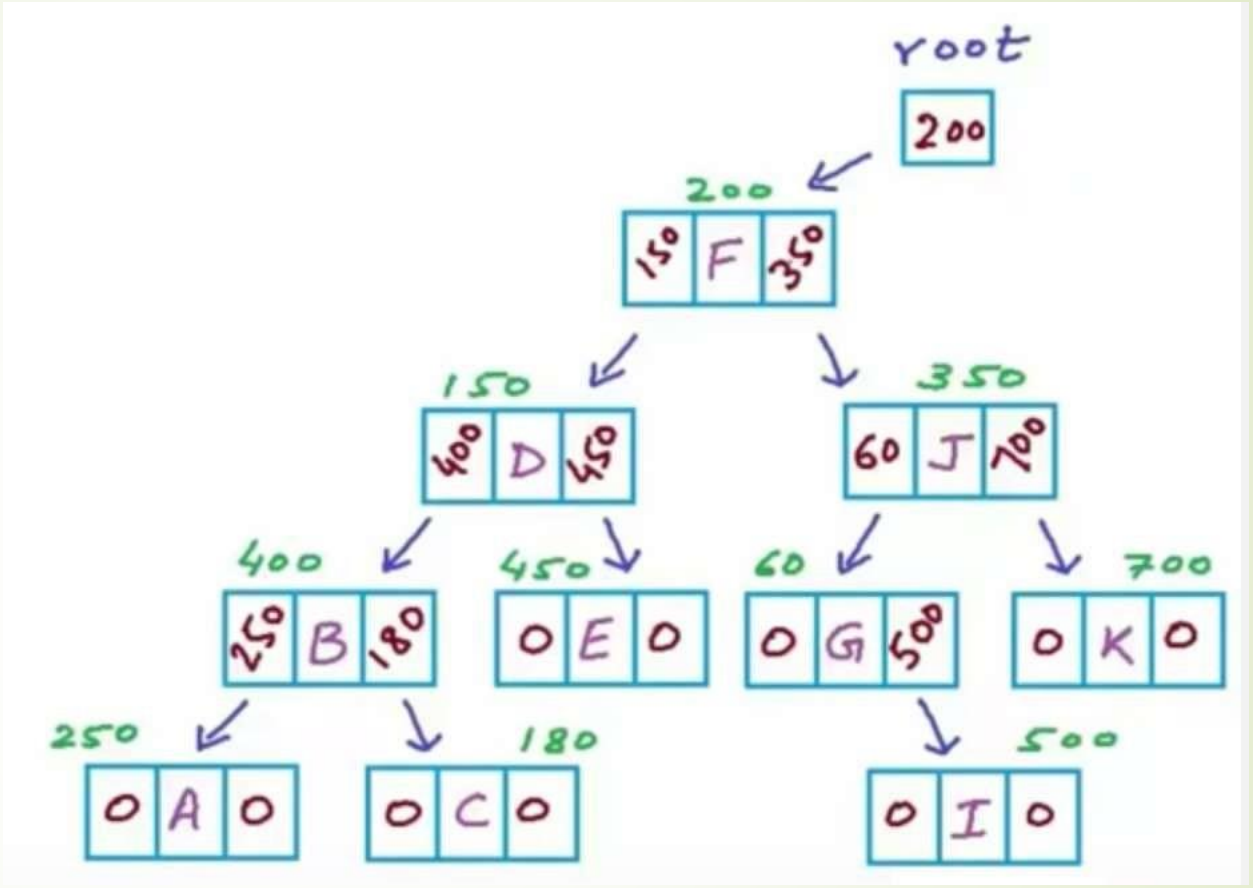


fig Binary tree

Post-order Pseudocode

```
struct Node{
    char data;
    Node *left;
    Node *right;
}
void Postorder(Node *root)
{
    if (root==NULL) return;
    Postorder(root->left);
    Postorder(root->right);
    printf ("%c", root->data);
}
```

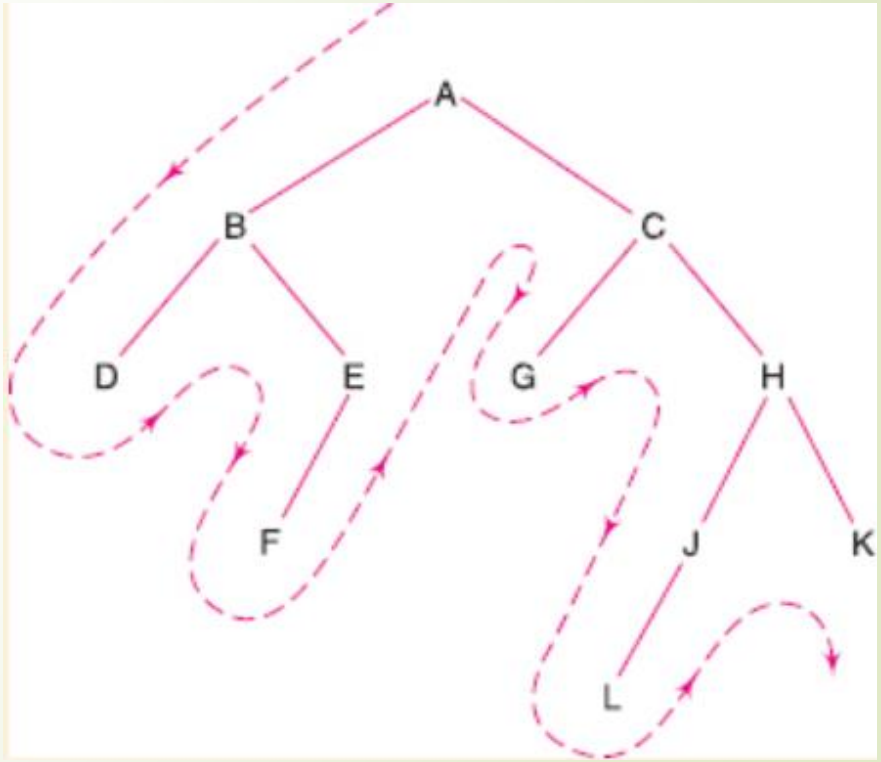


Order of leaf nodes

- Order of leaf nodes are same in all three traversals

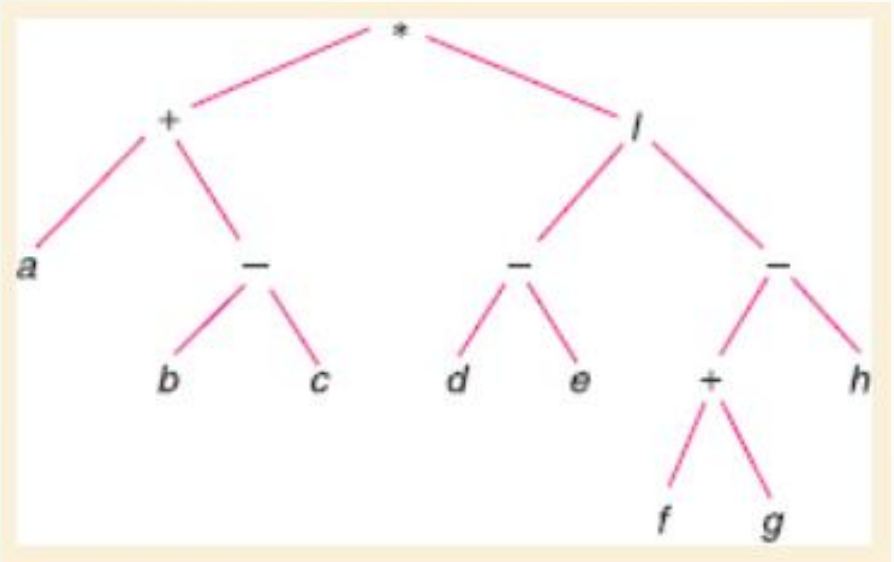
Pre-order: A B D E F C G H J L K

(Inorder)	D	B	F	E	A	G	C	L	J	H	K
(Postorder)	D	F	E	B	G	L	J	K	H	C	A



Algebraic Expressions

$$[a + (b - c)] * [(d - e)/(f + g - h)]$$

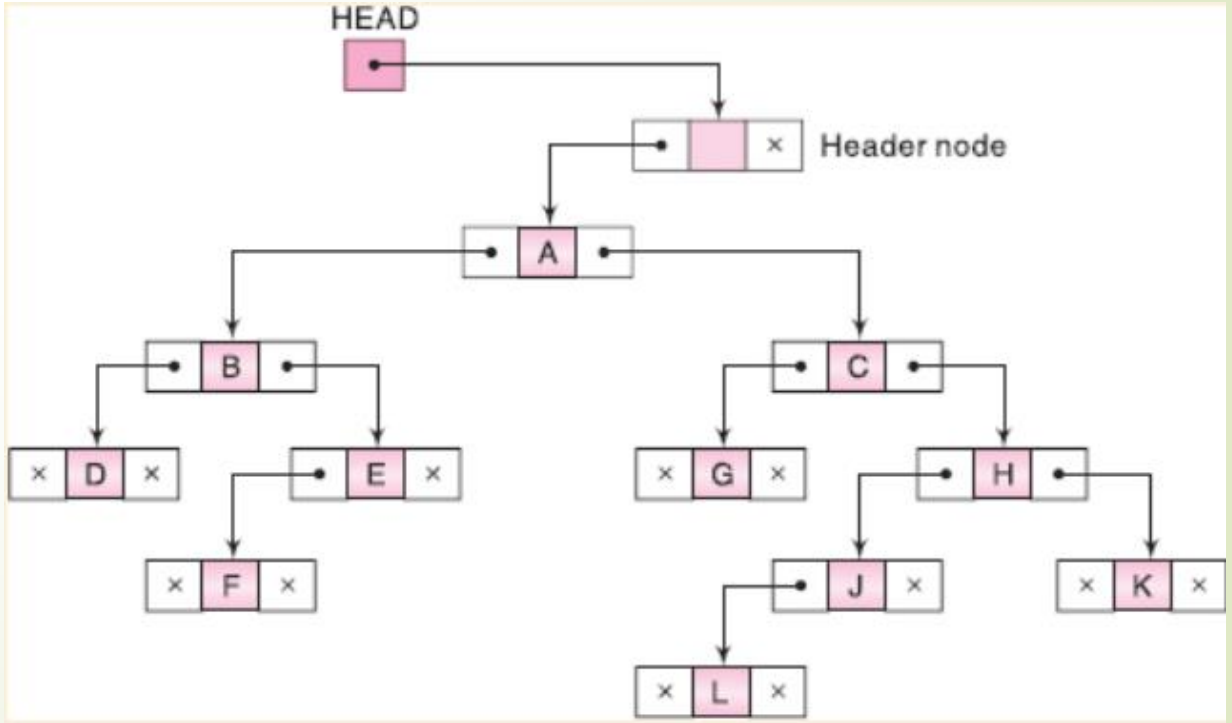


Inorder: Infix
Preorder: Prefix
Postorder: Postfix

(Preorder) * + a − b c / − d e − + f g h
(Postorder) a b c − + d e − f g + h − / *

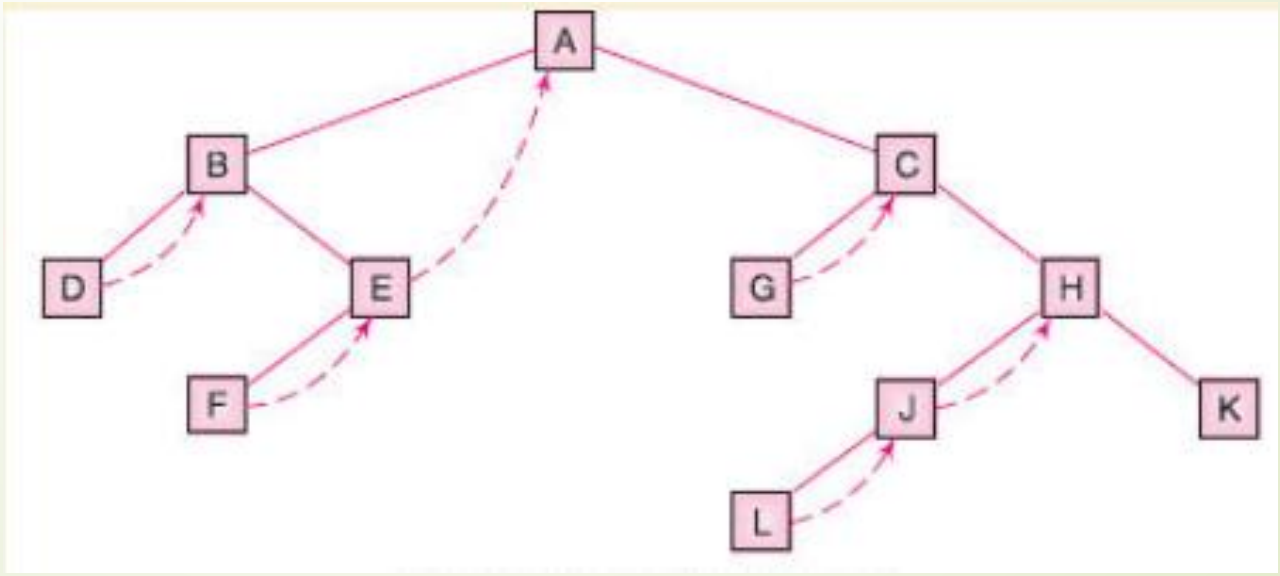
Header Node

- Pointer HEAD point to the header node
- Left pointer of the header node points to the root of the binary tree
- Right pointer points to NULL
- INFO part may store some summary information like number of nodes in the binary tree
- Header node as sentinel
  - If subtree is empty, it points to header node instead of NULL
- If binary tree is empty, there is still one node, the header node



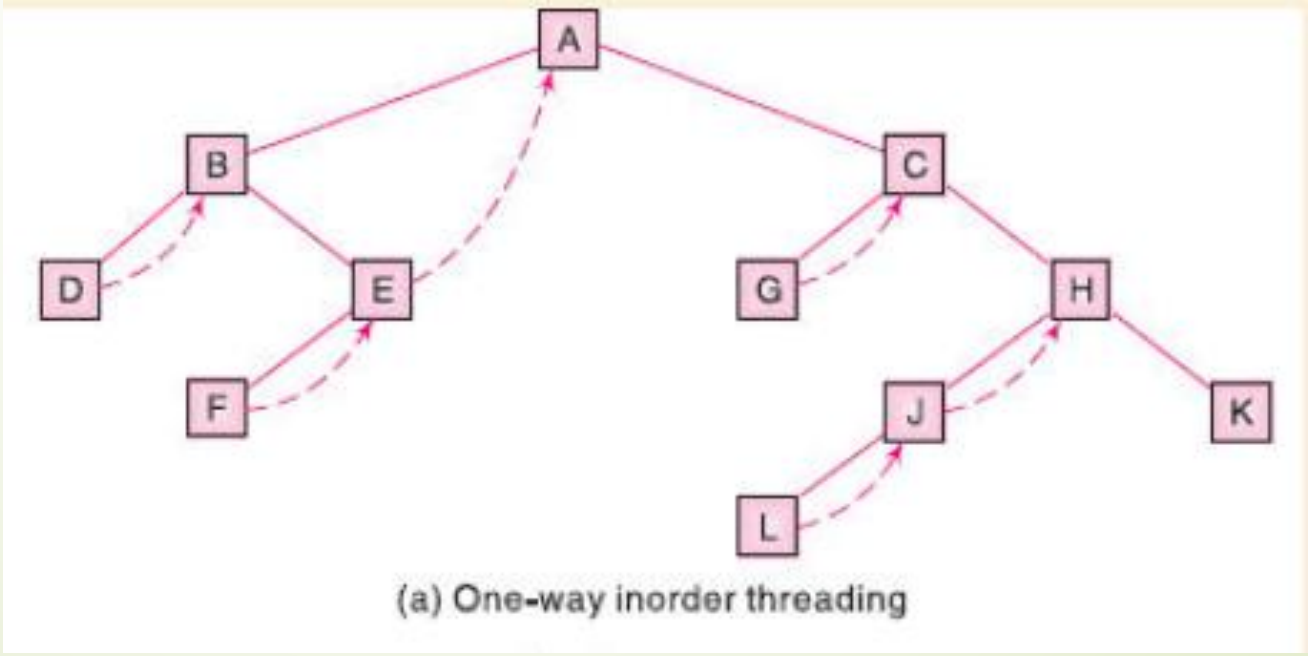
Threads

- NULL left and right pointers may be used for storing some useful information
- These pointers are called threads and such binary trees are called threaded binary trees
- Threads must be distinguished from normal pointers
  - Figures: use of dotted lines
  - Memory: use of one bit



Inorder threads

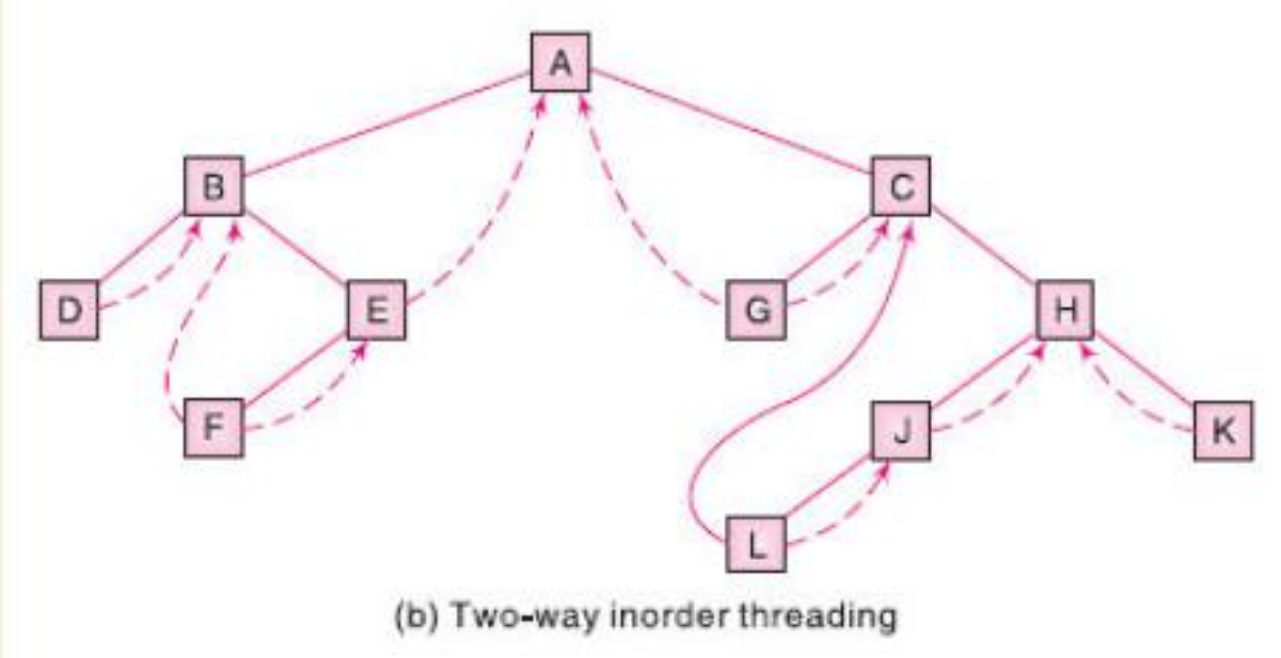
- One way threading
  - Inorder traversal in forward direction becomes easy
  - Right field of a node points to the inorder successor of the node
    - Right in-threaded binary tree
  - Right pointer of last node point to NULL (header node if it exists)



(a) One-way inorder threading

Inorder threads

- Two way threading
  - Inorder traversal in reverse direction becomes easy also
  - left field of a node points to the inorder predecessor of the node
    - left in-threaded binary tree
    - In-threaded binary tree (both kind of threads)
  - Left pointer of first node and right pointer of last node point to NULL (header node if it exists)



(b) Two-way inorder threading

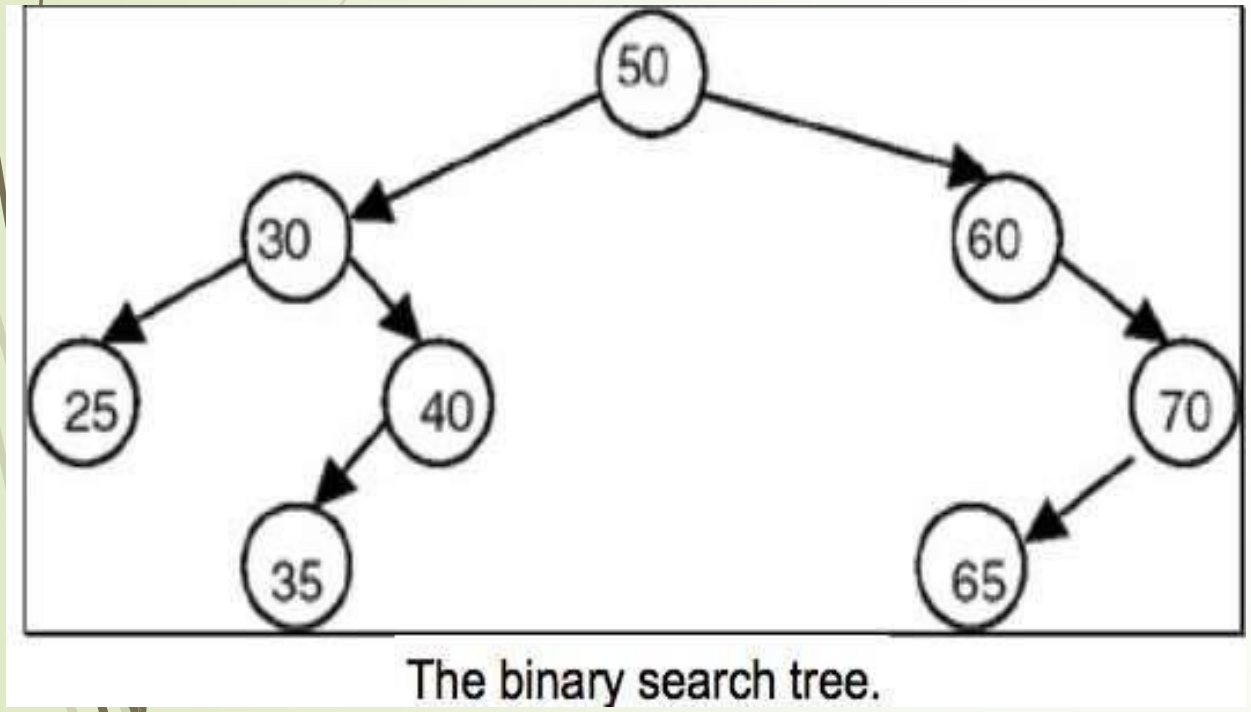
Preorder threads

- Preorder threads can be defined in similar way
- Postorder threads does not exist

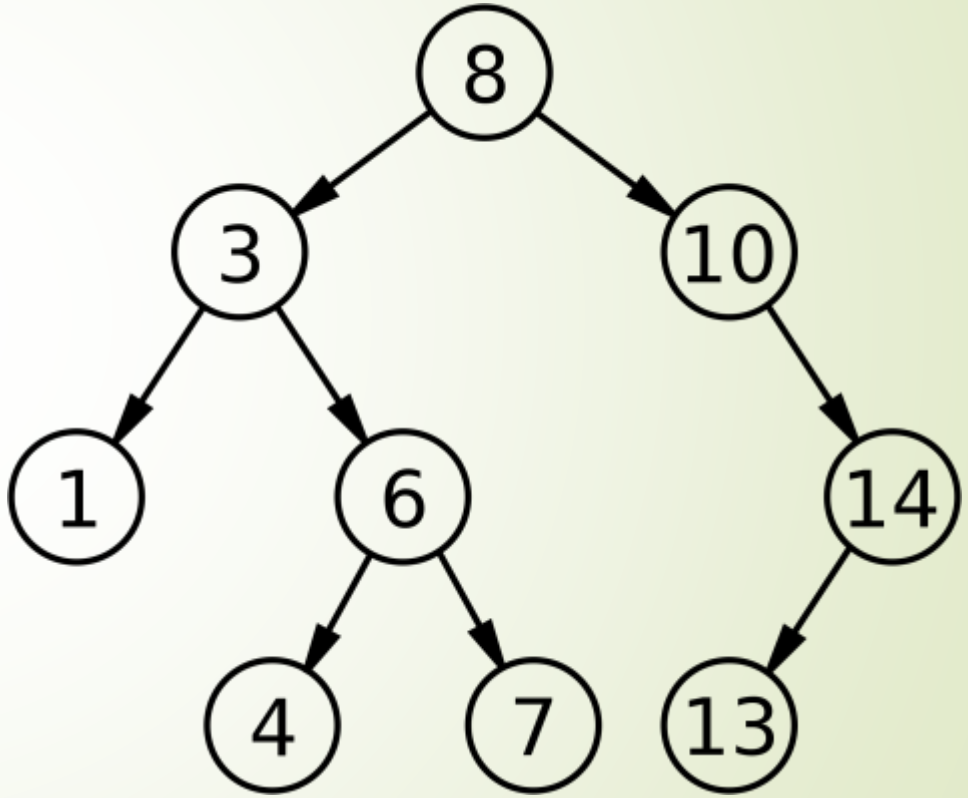
Binary Search Tree(BST)

- 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.



Binary Search Tree(BST)

- A binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder and postorder.

- If we traverse a binary search tree in inorder and print the keys (values) contained in the nodes of the tree, we get a sorted list of identifiers in ascending order.

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 array requires moving many of the entries every time.

Why Binary Search Tree?

- So we 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 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				
	Unsorted Array	**Sorted Array**	**Linked List**	**BST**
Search	O(n)	O(logn)	O(n)	O(logn)
Insert	O(1)	O(n)	O(1)	O(logn)
Remove	O(n)	O(n)	O(n)	O(logn)

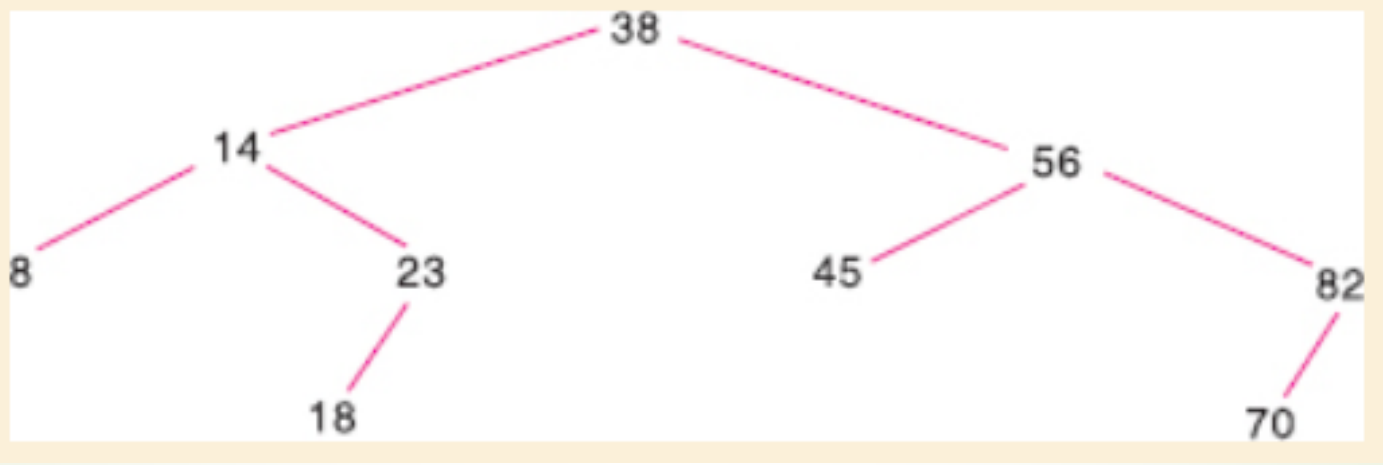
Operations on Binary Search Tree (BST)

- 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.

Searching Through The BST



Search for 18
Search for 75

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

(i) LOC = NULL and PAR = NULL will indicate that the tree is empty.
(ii) LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T.
(iii) LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

1. [Tree empty?]
If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and Return.
2. [ITEM at root?]
If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAB := NULL, and Return.
3. [Initialize pointers PTR and SAVE.]
If ITEM < INFO[ROOT], then:
    Set PTR := LEFT[ROOT] and SAVE := ROOT.
Else:
    Set PTR := RIGHT[ROOT] and SAVE := ROOT.
[End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. [ITEM found?]
    If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6. If ITEM < INFO[PTR], then:
        Set SAVE := PTR and PTR := LEFT[PTR].
    Else:
        Set SAVE := PTR and PTR := RIGHT[PTR].
    [End of If structure.]
    [End of Step 4 loop.]
7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.

C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
        if(root == NULL)
        {
                printf("The number does not exist");
                exit(1);
        }
        else  if (key == root->info)
        {
                printf("The searched item is found"):
        }
        else if(key < root->info)
                return BinSearch(root->left, key);
        else
                return BinSearch(root->right, key);
}
```

Searching Complexity in BST

- We follow a single path from root to leaf node while searching
- Number of comparisons is bounded by depth of BST
- Searching complexity is proportional to the depth of BST

- Best case: depth is minimum : BT is complete: $O(\log_2 n)$
- Worst case: BT is skewed : $O(n)$
- Average case: consider all BTs with n nodes = n!

  average depth = $1.4 \log_2 n$

  Average complexity=$O(\log_2 n)$

Insertion of a node in BST

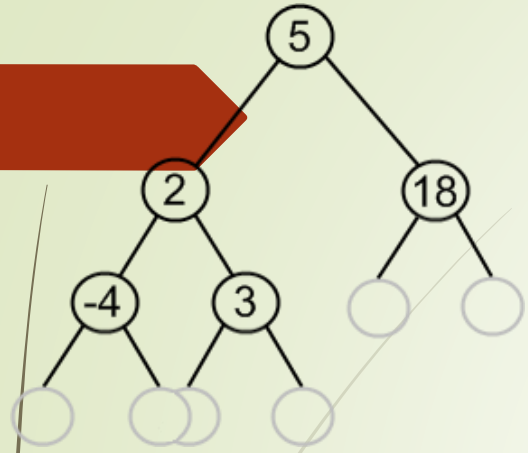
- To insert a new item in a tree, we must first verify that its key is different from those of existing elements.
- If a new value is less, than the current node's value, go to the left subtree, else go to the right subtree.
- Following this simple rule, the algorithm reaches a node, which has no left 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.

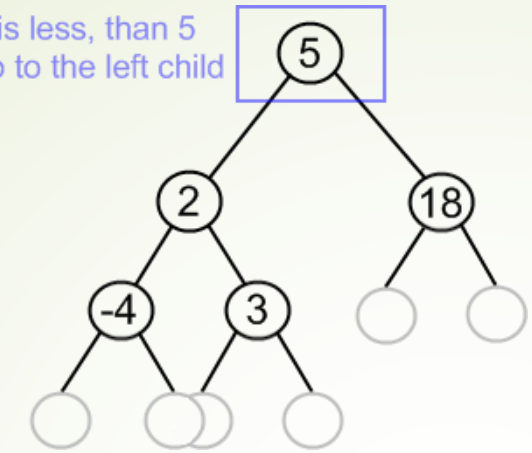
Algorithm for insertion in BST

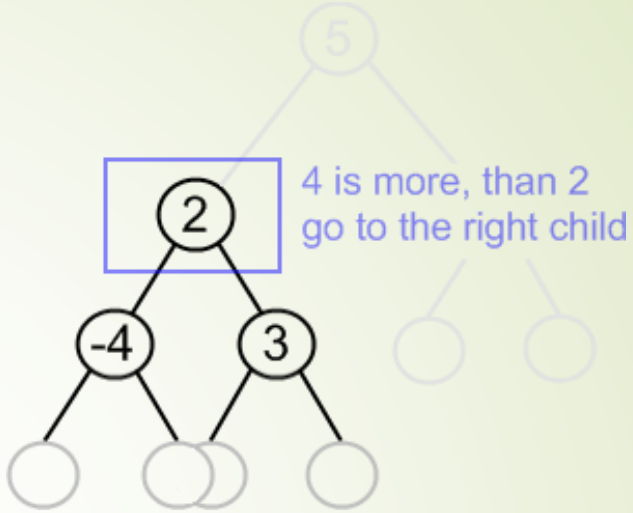
- Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

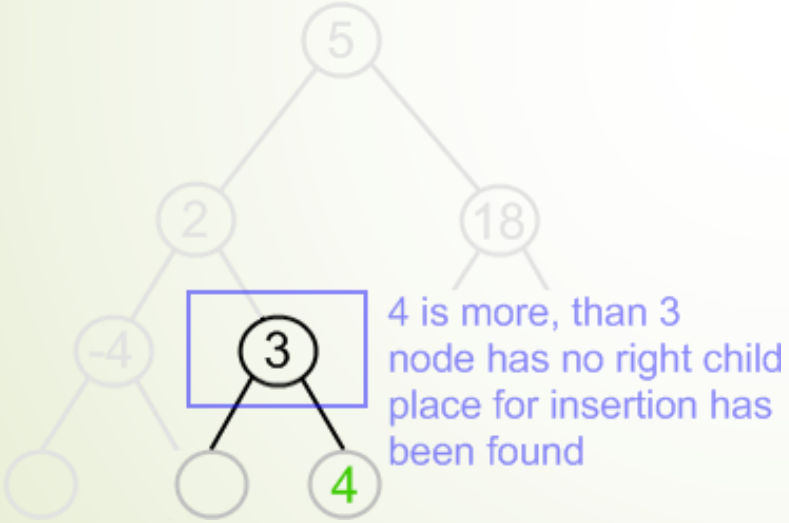
- if a new value is less than the node's value:
  - if a current node has no left child, place for insertion has been found;
  - otherwise, handle the left child with the same algorithm.

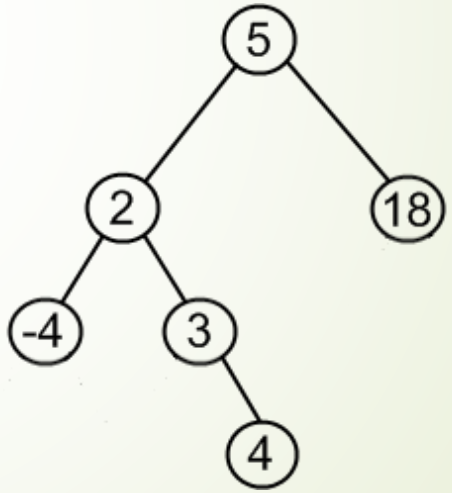
- if a new value is greater, than the node's value:
  - if a current node has no right child, place for insertion has been found;
  - otherwise, handle the right child with the same algorithm.


5
2
18
-4
3

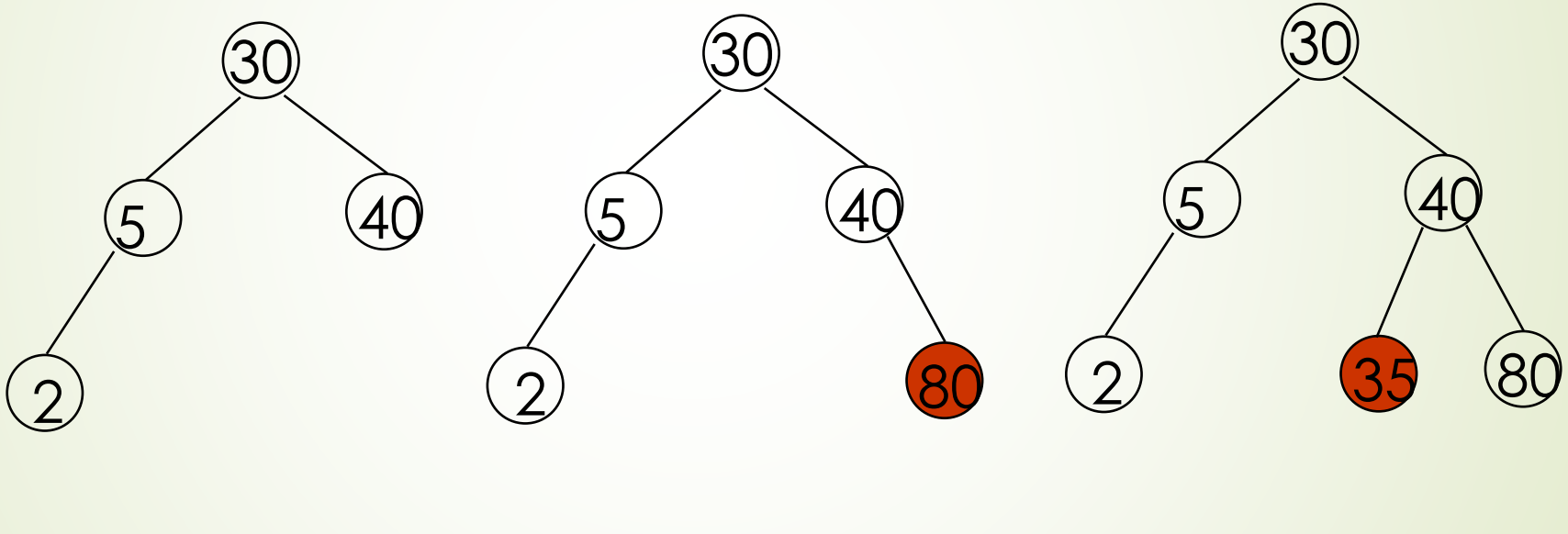

4 is less, than 5
go to the left child
5
2
18
-4
3


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


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


5
2
18
-4
3
4

Insert Node in Binary Search Tree



Insert 80

Insert 35

C function for BST insertion:

```
void insert(struct bnode *root, int item)
{
        if(root=NULL)
        {
                root=(struct bnode*)malloc (sizeof(struct bnode));
                root->left=root->right=NULL;
                root->info=item;
        }
        else
        {
                if(item<root->info)
                        root->left=insert(root->left, item);
                else
                        root->right=insert(root->right, item);
        }
}
```

Insertion Complexity in BST

- We follow a single path from root to leaf node while inserting new node
- Number of comparisons is bounded by depth of BST
- Insertion complexity is proportional to the depth of BST

- Best case: depth is minimum : BT is complete: $O(\log_2 n)$
- Worst case: BT is skewed : $O(n)$
- Average case: consider all BTs with n nodes = n!

  average depth = $1.4 \log_2 n$

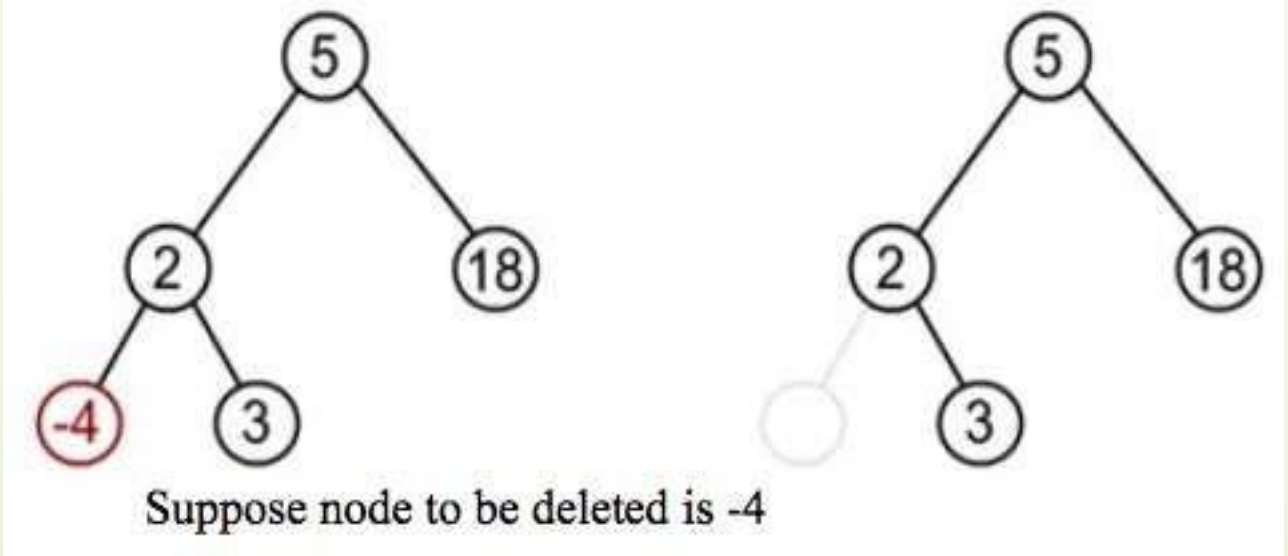
  Average complexity=$O(\log_2 n)$

if you have to create a BST with n elements, complexity = $O(n\log_2 n)$

Deleting a node from the BST

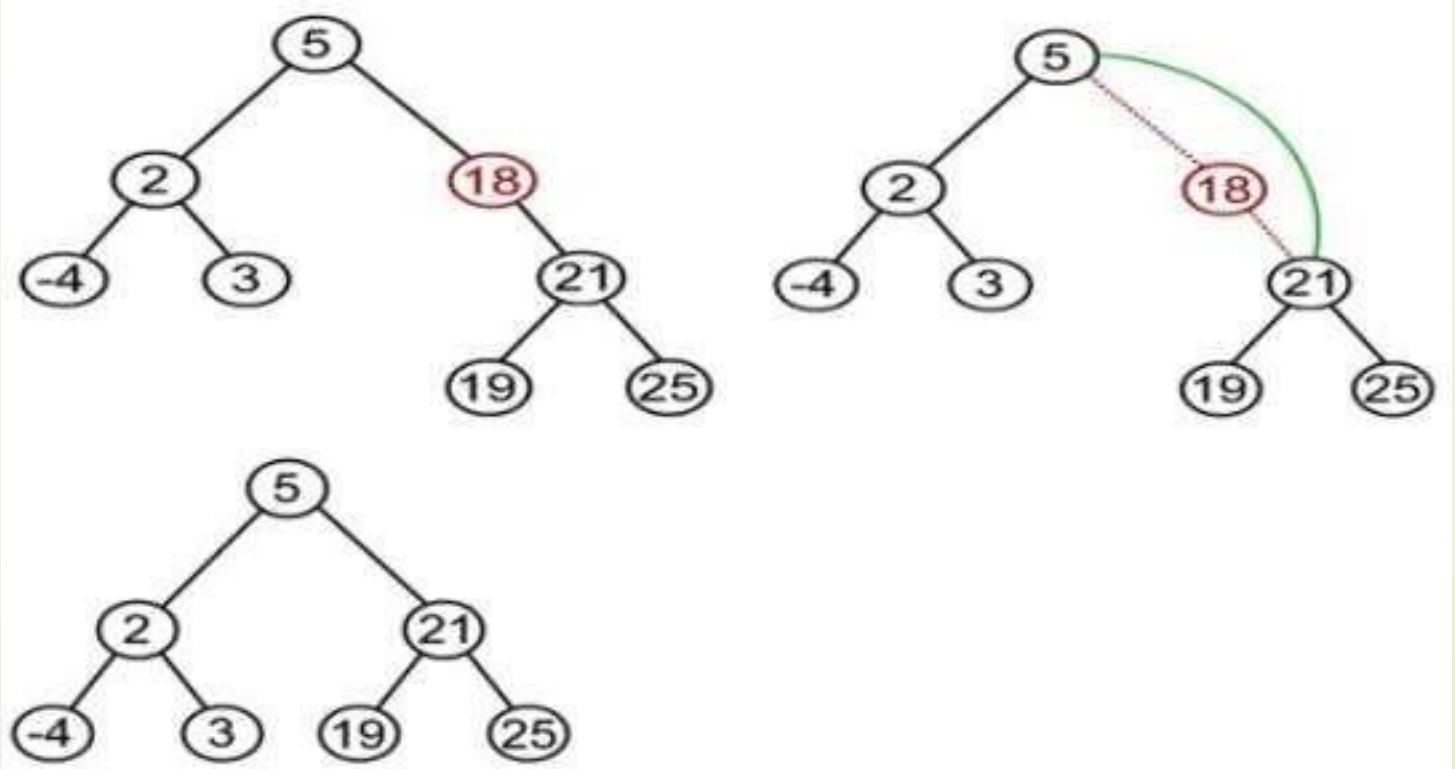
- While deleting a node from BST, there may be three cases:

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

2. 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



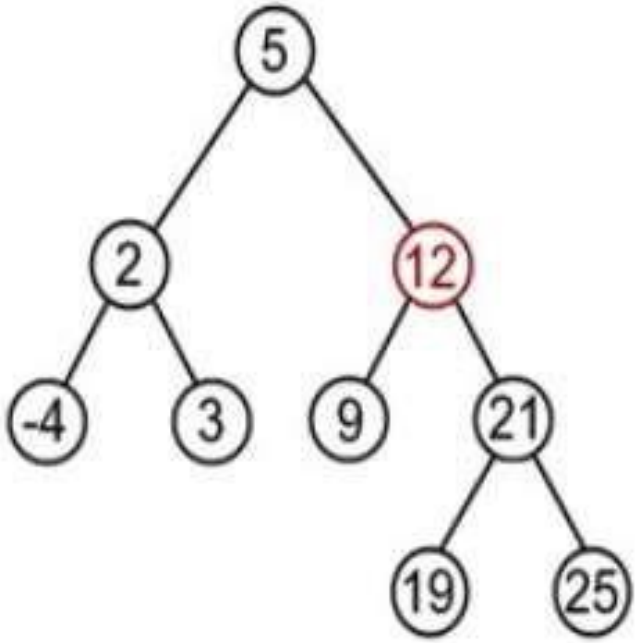
Deleting a node from the BST

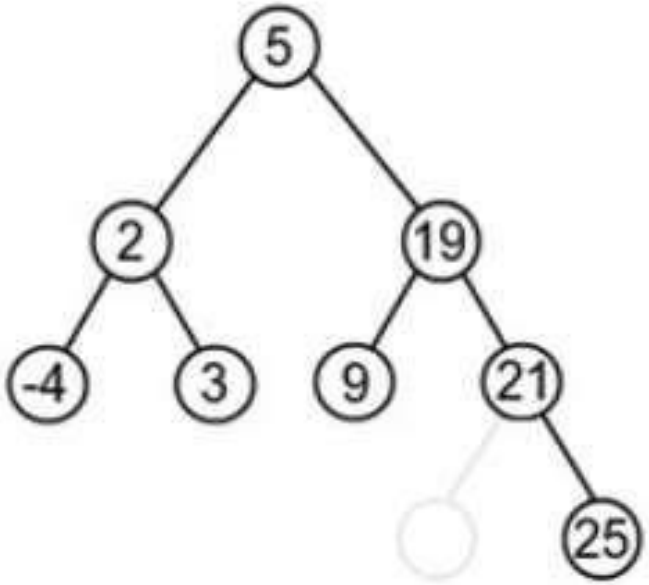
3. the node to be deleted has two children:

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.



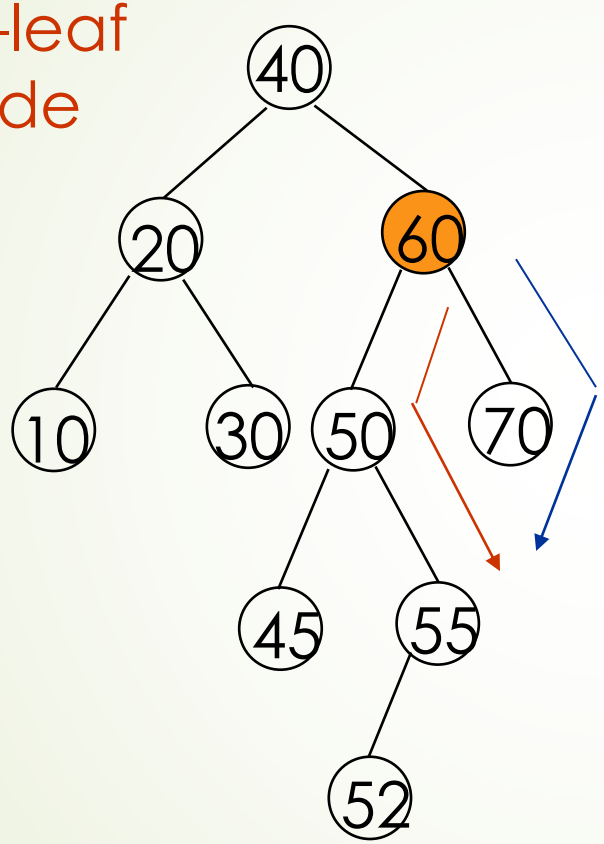


Suppose node to deleted is 12

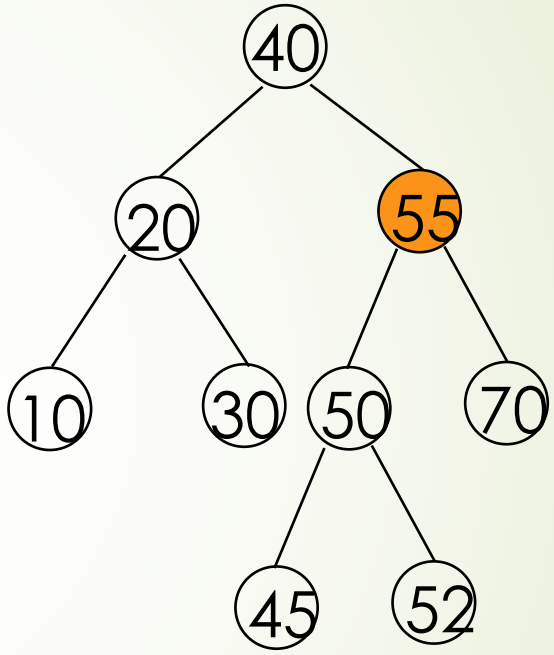
Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

Deletion for A Binary Search Tree

non-leaf node



Before deleting 60



After deleting 60

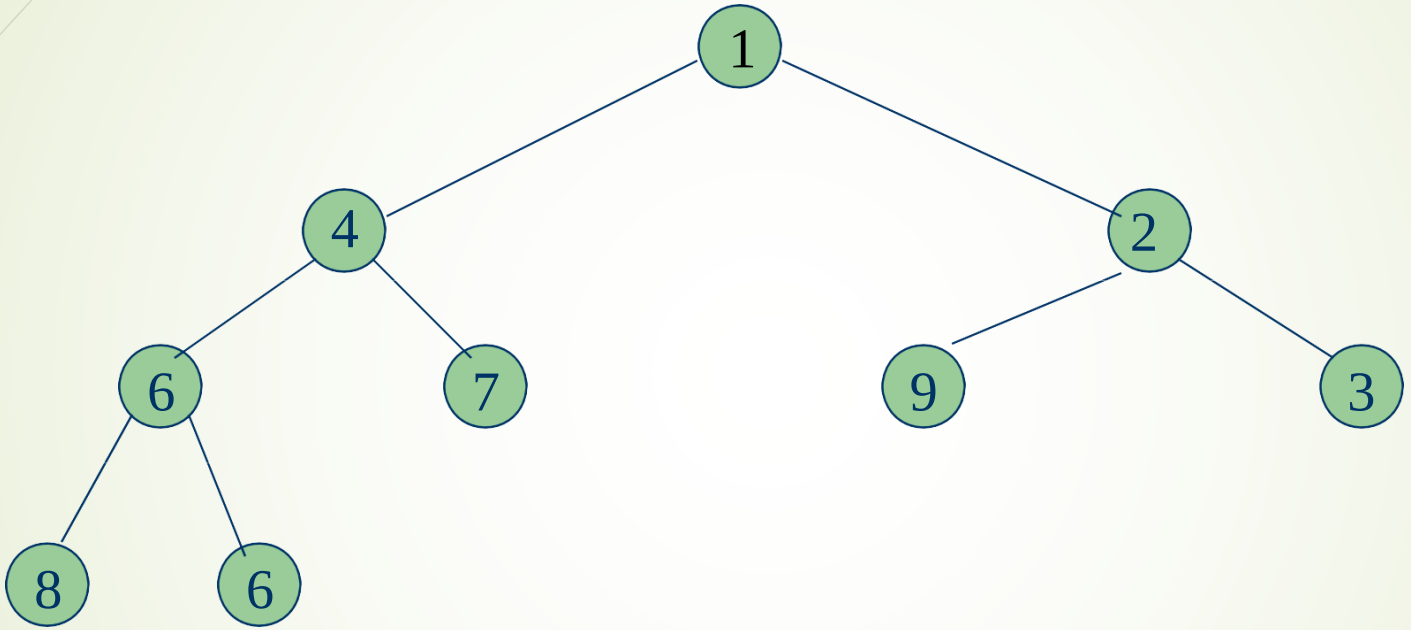
General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

Heap

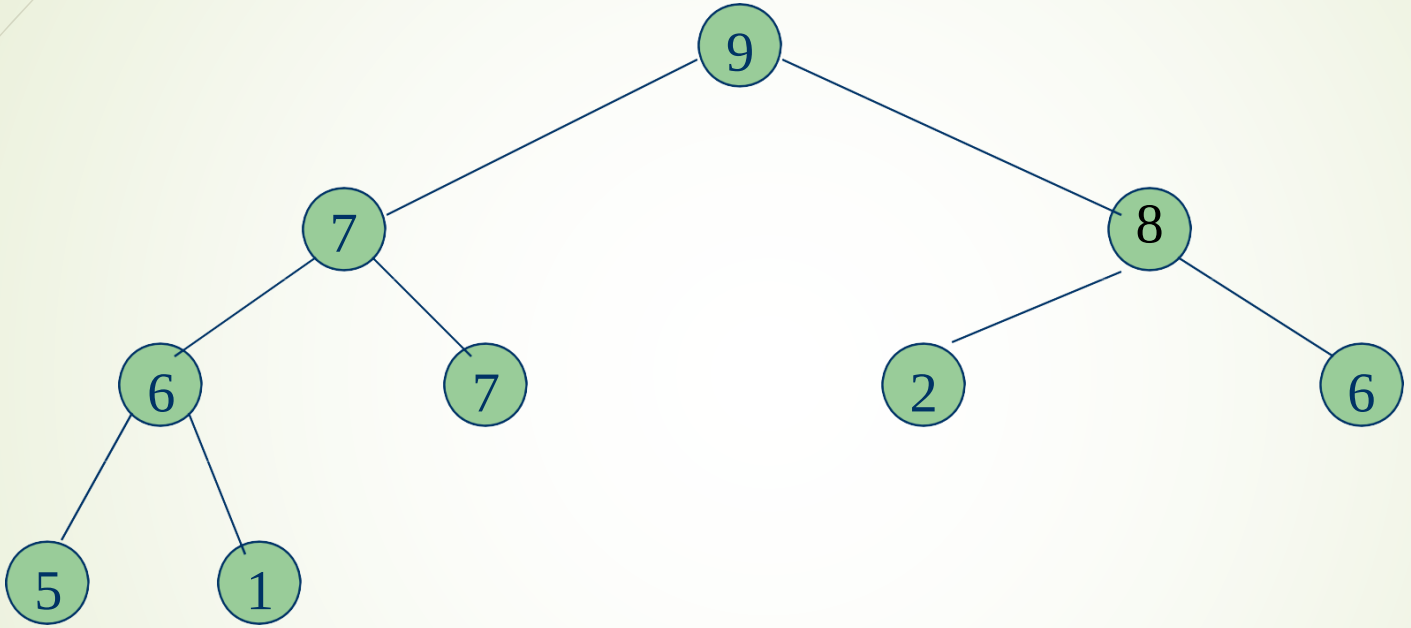
- A *max heap* is a binary tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is an almost complete binary tree.
- A *min heap* is a binary tree in which the key value in each node is no larger than the key values in its children. A *min heap* is an almost complete binary tree.

Min Heap With 9 Nodes



Complete binary tree with 9 nodes

Max Heap With 9 Nodes



Complete binary tree with 9 nodes

Heap Height

Since a heap is an almost complete binary tree, the height of an n node heap is $\lceil \log_2(n+1) - 1 \rceil$ or $O(\log n)$.

Operations in Heaps

- Delete the minimum/maximum value and return it. This operation is called
  - deleteMin / deleteMax.
- Insert a new data value

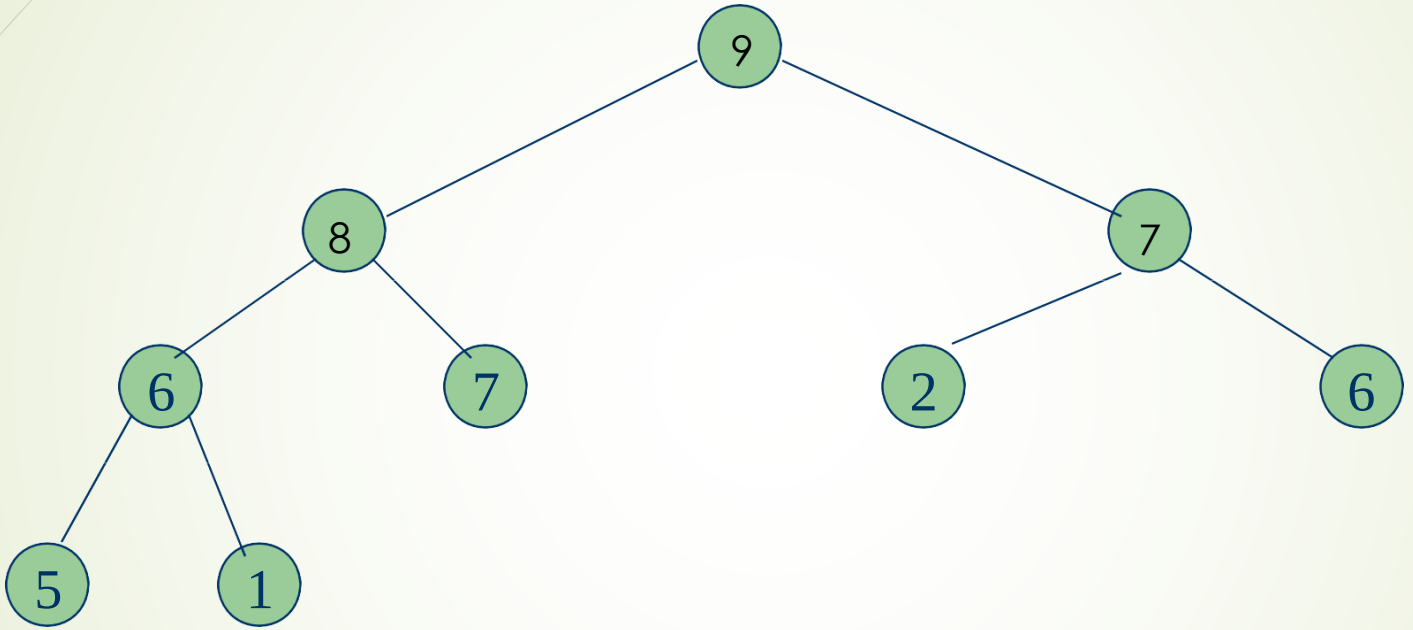
Application of Heaps

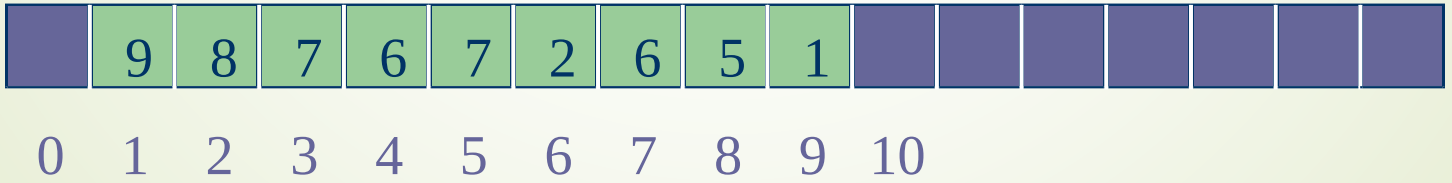
Applications of Heaps:

- A heap implements a **priority queue**, which is a queue that orders entities not a on first-come first-serve basis, but on a priority basis: the item of highest priority is at the head, and the item of the lowest priority is at the tail

- Another application: sorting, which will be seen later

A Heap Is Efficiently Represented as An Array

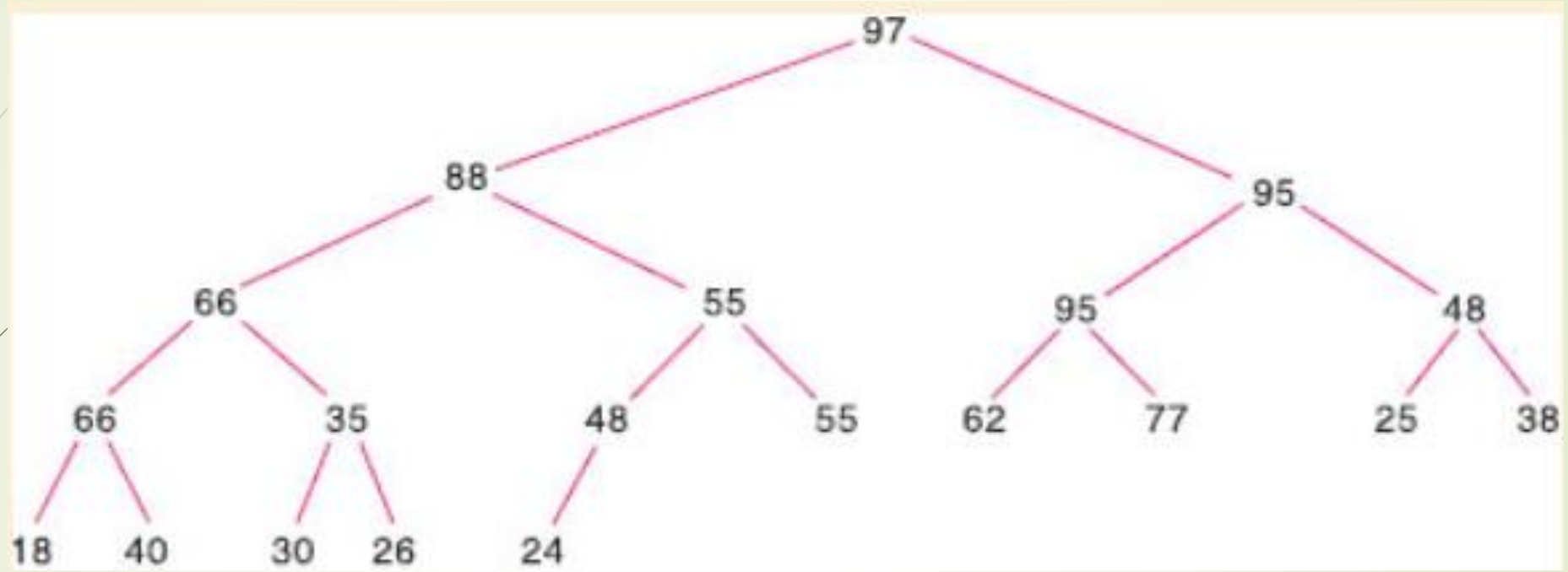


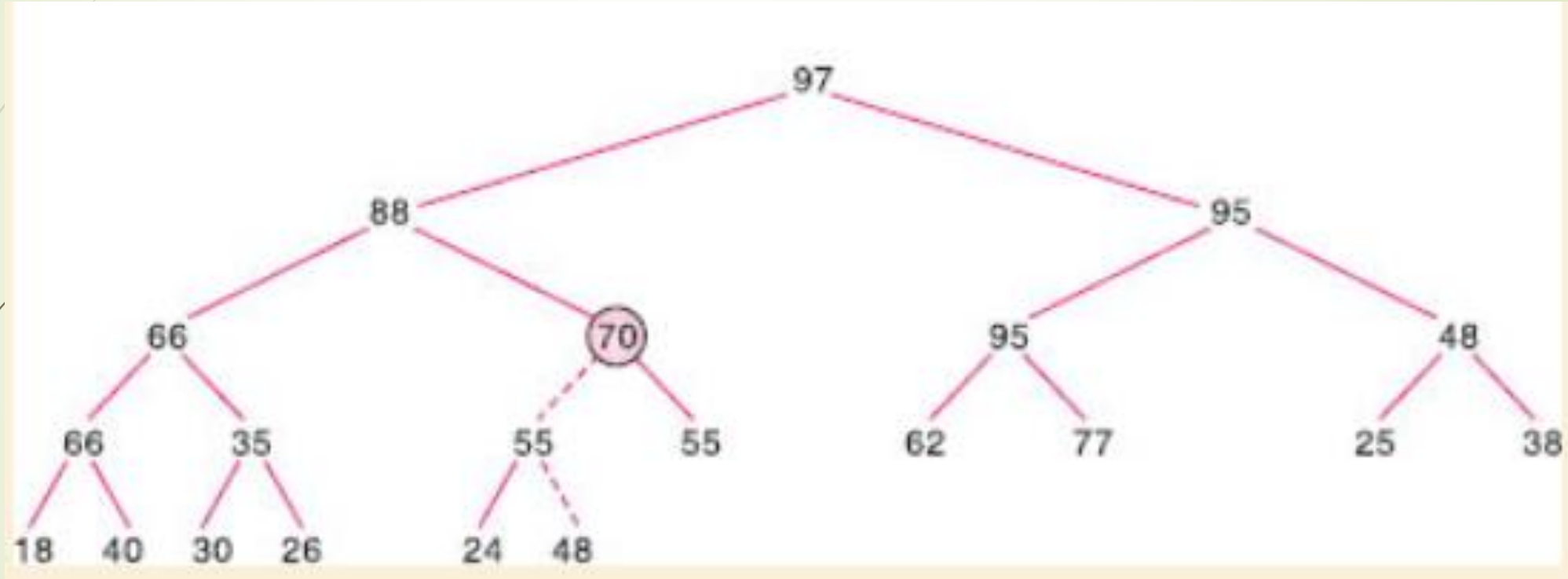


Inserting into a max-heap

- Suppose you want to insert a new value x into the heap
- Create a new node at the “end” of the heap (or insert *x* at the end of the array)
- If x is <= its parent, done
- Otherwise, we have to restore the heap:
  - Repeatedly swap *x* with its parent until either *x* reaches the root or *x* becomes <= its parent

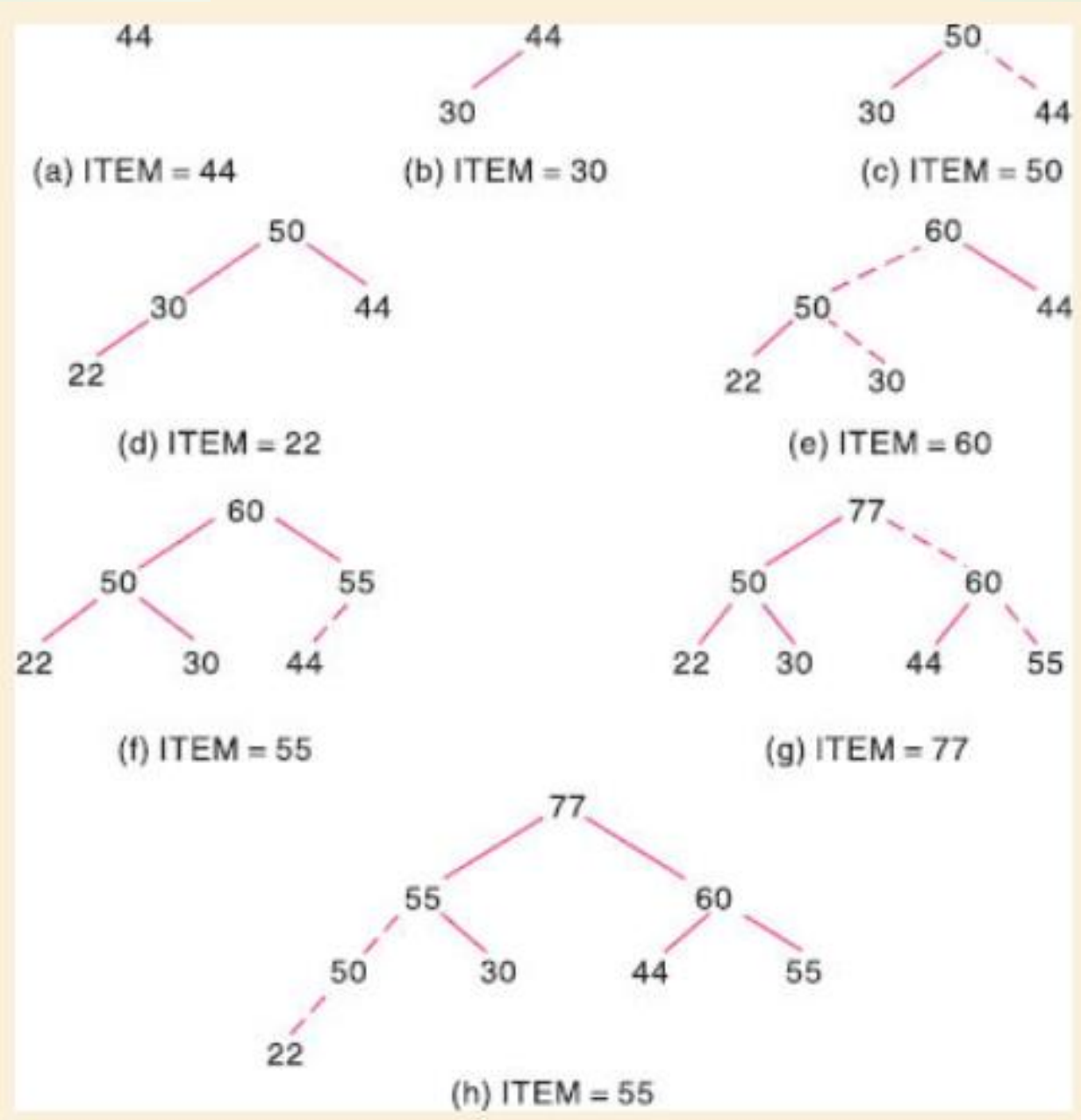
Insert new value in max heap 70




97
88
95
66
70
95
48
66
35
55
55
62
77
25
38
18
40
30
26
24
48

Building a new heap

44, 30, 50, 22, 60, 55, 77, 55

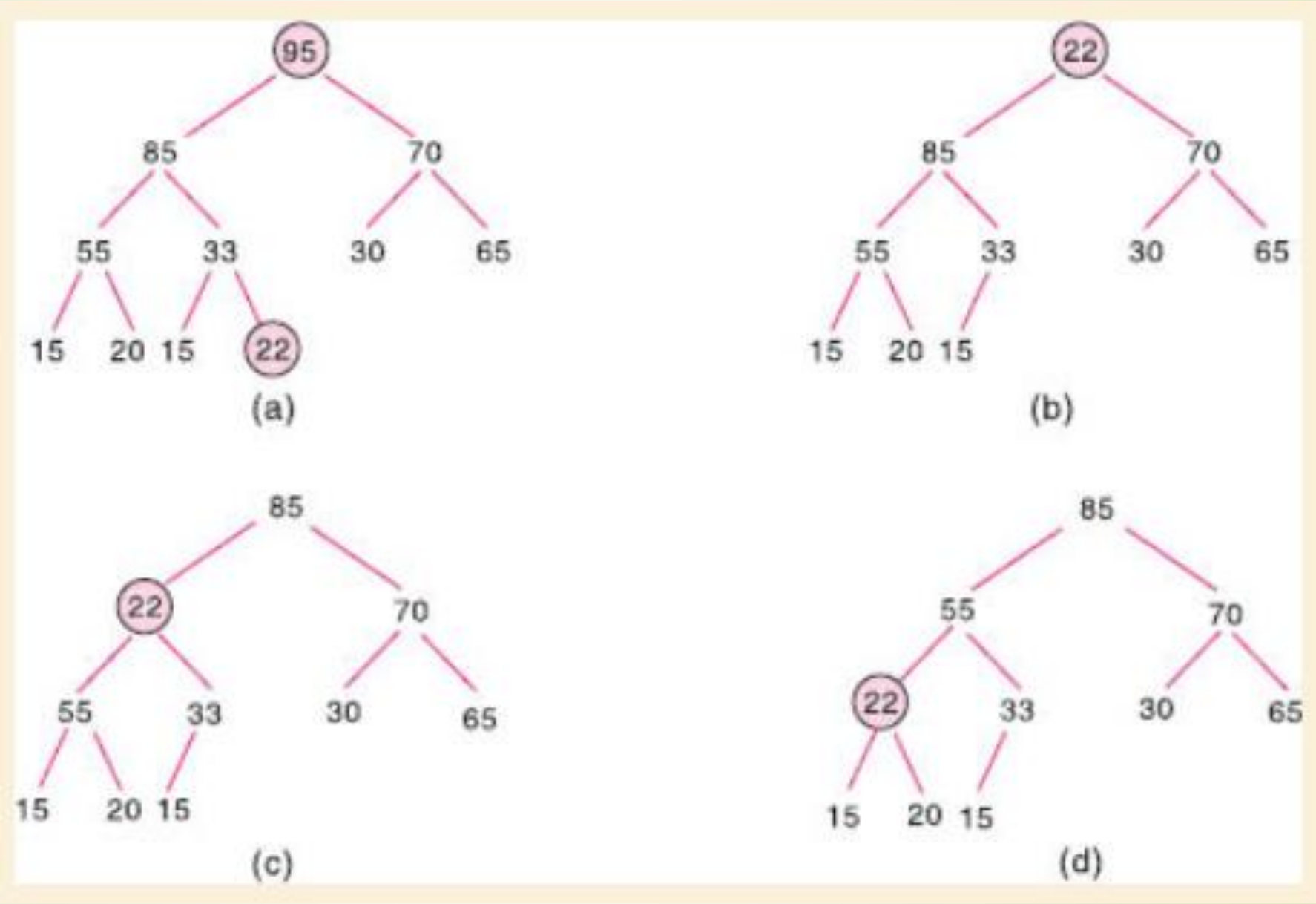


INSHEAP(TREE, N, ITEM)

1. [Add new node to H and initialize PTR.]
Set N := N + 1 and PTR := N.
2. [Find location to insert ITEM.]
Repeat Steps 3 to 6 while PTR < 1.
3. Set PAR := ⌊PTR/2⌋. [Location of parent node.]
4. If ITEM ≤ TREE[PAR], then:
Set TREE[PTR] := ITEM, and Return.
[End of If structure.]
5. Set TREE[PTR] := TREE[PAR]. [Moves node down.]
6. Set PTR := PAR. [Updates PTR.]
[End of Step 2 loop.]
7. [Assign ITEM as the root of H.]
Set TREE[I] := ITEM.
8. Return.

Deletion of root node

(1) Assign the root R to some variable ITEM.
(2) Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
(3) (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.


95
85
70
55
33
30
65
15
20
15
22
(a)
22
85
70
55
33
30
65
15
20
15
(b)
85
22
70
55
33
30
65
15
20
15
(c)
85
55
70
22
33
30
65
15
20
15
(d)

DELHEAP(TREE, N, ITEM)

1. Set ITEM := TREE[1]. [Removes root of H.]
2. Set LAST := TREE[N] and N := N − 1. [Removes last node of H.]
3. Set PTR := 1, LEFT := 2 and RIGHT := 3. [Initializes pointers.]
4. Repeat Steps 5 to 7 while RIGHT ≤ N:
5. If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then:
   Set TREE[PTR] := LAST and Return.
   [End of If structure.]
6. IF TREE[RIGHT] ≤ TREE[LEFT], then:
   Set TREE[PTR] := TREE[LEFT] and PTR := LEFT.
   Else:
   Set TREE[PTR] := TREE[RIGHT] and PTR := RIGHT.
   [End of If structure.]
7. Set LEFT := 2*PTR and RIGHT := LEFT + 1.

[End of Step 4 loop.]

8. If LEFT = N and if LAST < TREE[LEFT], then: Set PTR := LEFT.
9. Set TREE[PTR] := LAST.
10. Return.

Heapsort

- Array A with n elements is given

Build a heap H out of the elements of A.
Repeatedly delete the root element of H.

HEAPSORT(A, N)

1. [Build a heap H, [redacted]]

Repeat for J = 1 to N – 1:

    Call INSHEAP(A, J, A[J + 1]).

[End of loop.]

2. [Sort A by repeatedly deleting the root of H, [redacted].]

Repeat while N > 1:

    (a) Call DELHEAP(A, N, ITEM).

    (b) Set A[N + 1] := ITEM.

[End of Loop.]

3. Exit.

Heapsort Complexity

- Heap is an almost complete binary tree
  - Height is $\log_2 n$
- Complexity of building heap
  - Inserting a single element is bounded by $\log_2 n$ (maximum length of any branch)
  - Inserting n elements is bounded by $n\log_2 n$
- Deletion of root is bounded by $\log_2 n$ (sinking root along any branch)
  - Deleting root n times is bounded by $n\log_2 n$
- Complexity of heapsort is $O(\log_2 n)$

Balanced Binary Tree (AVL tree)

- Binary search trees store linearly ordered data
- Best case height: $\Theta(\ln(n))$
- Worst case height: $\mathbf{O}(n)$ (left or right skewed binary tree)

Requirement:

- Define and maintain a *balance* to ensure $\Theta(\ln(n))$ height

Height balanced Binary Search Tree (AVL search tree)

AVL tree: Named after Adelson-Velskii and Landis

An empty tree is height-balanced. A non-empty binary tree T is balanced if:

1) Left subtree of root of T is balanced
2) Right subtree of root of T is balanced
3) The difference between heights of left subtree and right subtree is not more than 1.

An empty tree has height –1

A tree with a single node has height 0

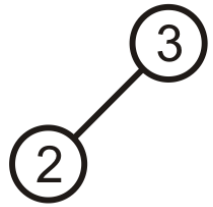
Balanced binary search tree is AVL search tree

Difference between the height of left and right subtrees for a node is called its balance factor

Balance factor for a node can be either -1, 0 or 1

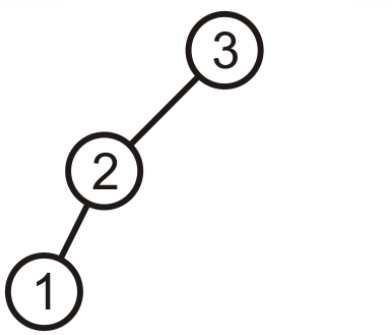
Prototypical Examples

These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



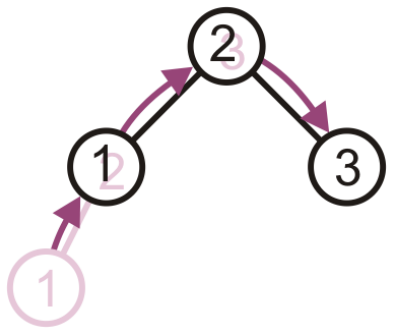
Prototypical Examples

This is more like a linked list; however, we can fix this…



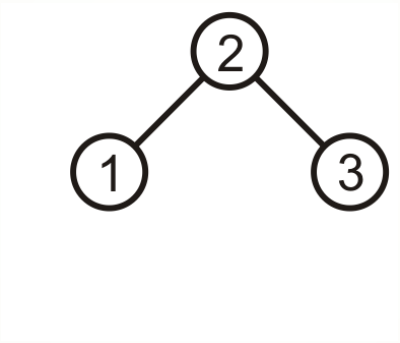
Prototypical Examples

Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2



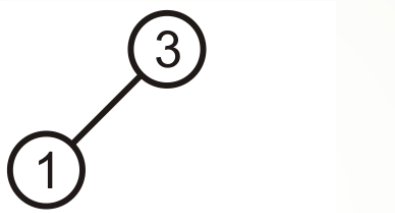
Prototypical Examples

The result is a perfect, though trivial tree



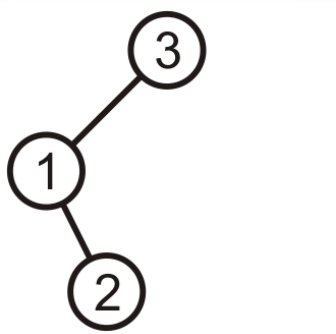
Prototypical Examples

Alternatively, given this tree, insert 2



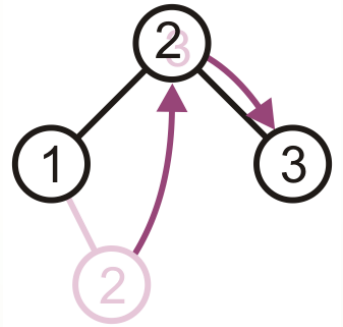
Prototypical Examples

Again, the product is a linked list; however, we can fix this, too



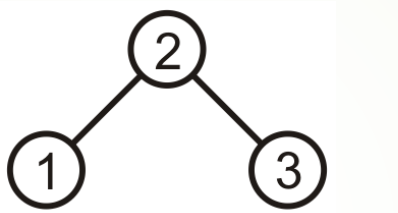
Prototypical Examples

Promote 2 to the root, and assign 1 and 3 to be its children



Prototypical Examples

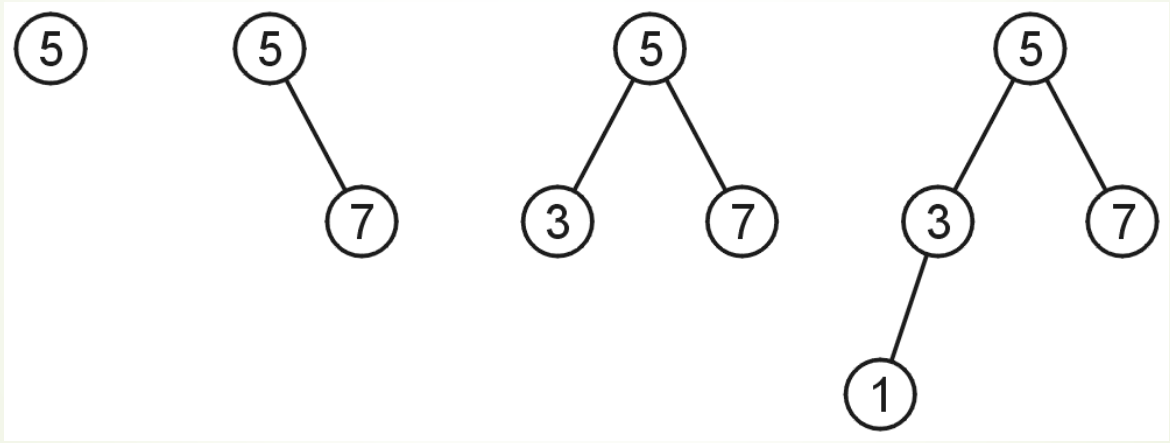
The result is, again, a perfect tree



These examples may seem trivial, but they are the basis for the corrections in the AVL trees

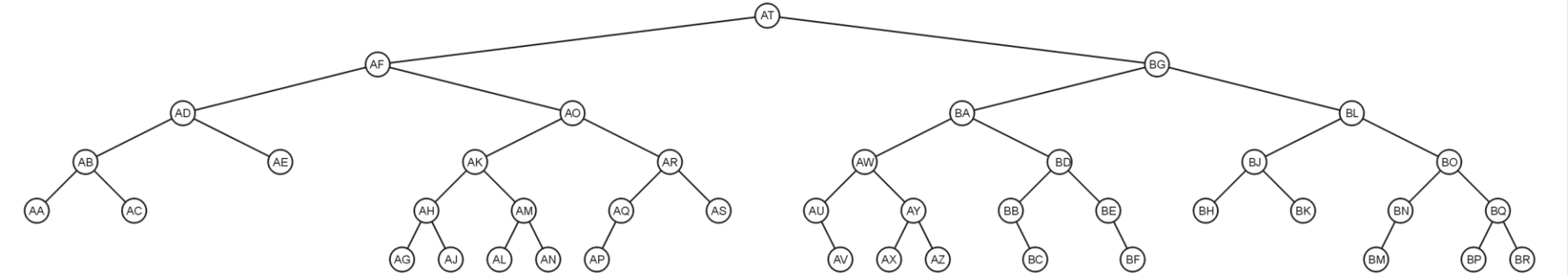
AVL Trees

AVL trees with 1, 2, 3, and 4 nodes:



AVL Trees

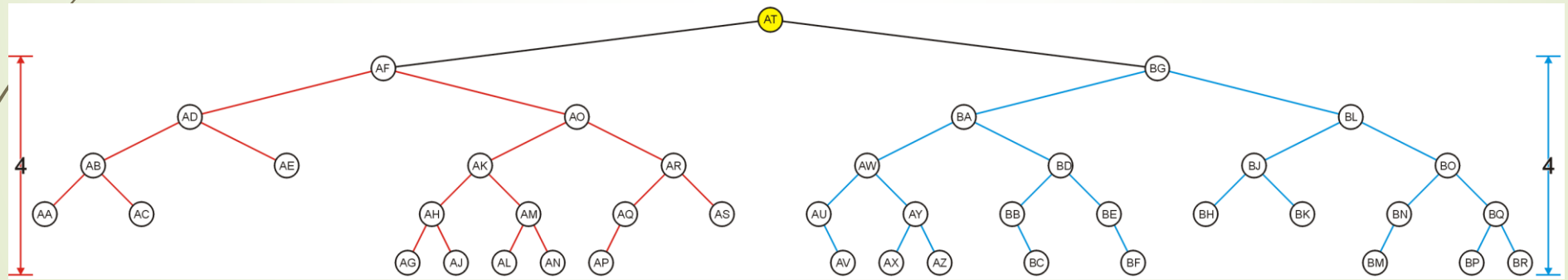
Here is a larger AVL tree (42 nodes):



AVL Trees

The root node is AVL-balanced:

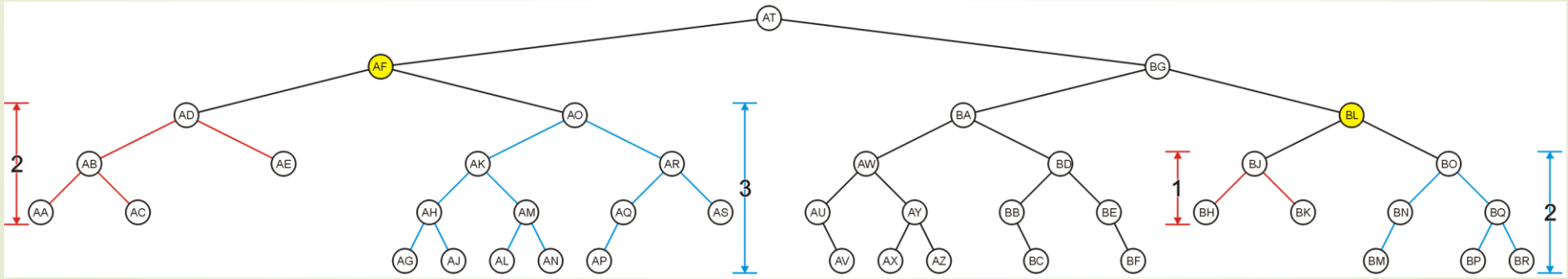
- Both sub-trees are of height 4:



AVL Trees

All other nodes are AVL balanced

- The sub-trees differ in height by at most one



Height of an AVL Tree

By the definition of complete trees, any complete binary search tree is an AVL tree

Thus an upper bound on the number of nodes in an AVL tree of height h a perfect binary tree with $2^{h+1} - 1$ nodes

Maintaining Balance: Rotations

To maintain AVL balance, observe that:

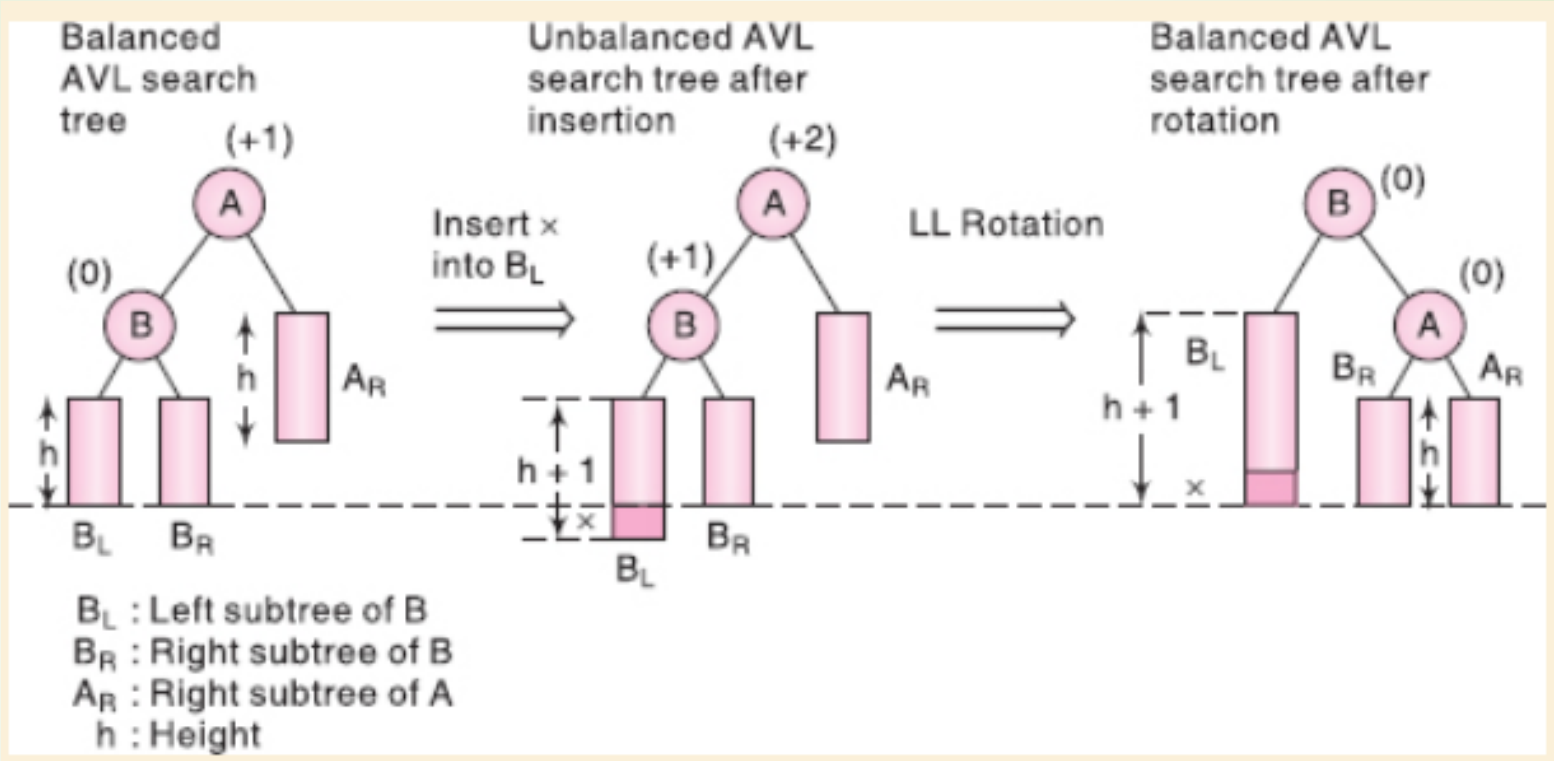
- Inserting a node can increase the height of a tree by at most 1
- Removing a node can decrease the height of a tree by at most 1

If tree is not balanced Rotations are performed to make the tree balanced

Identify node A for which balance factor is neither -1, 0 nor 1 and is nearest ancestor to the inserted node on the path from inserted node to the root.

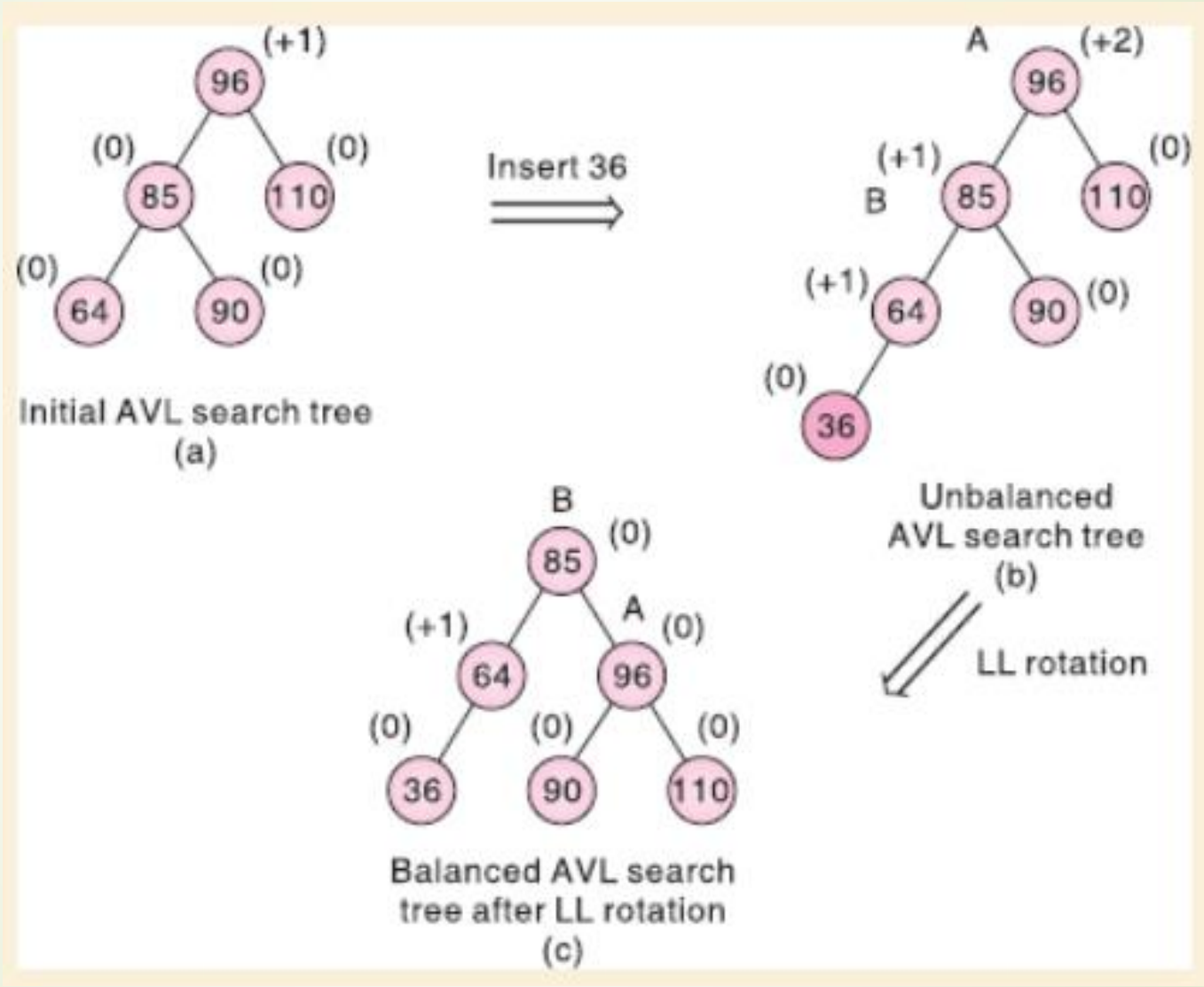
LL rotation: Inserted node is in the left subtree of left subtree of node A
RR rotation: Inserted node is in the right subtree of right subtree of node A
LR rotation: Inserted node is in the right subtree of left subtree of node A
RL rotation: Inserted node is in the left subtree of right subtree of node A

LL Rotation

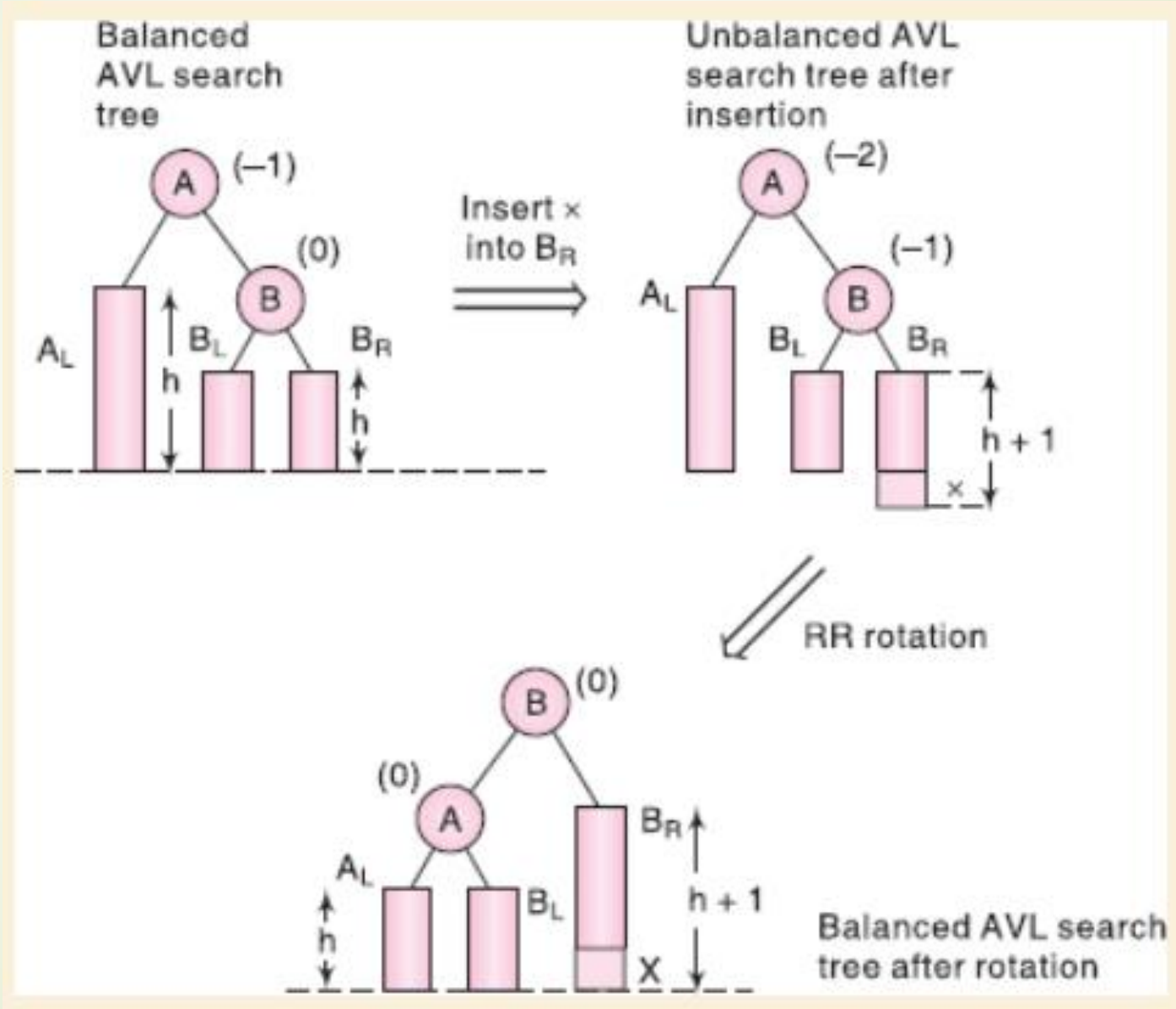


The new element X is inserted in the left subtree of left subtree of A, the closest ancestor node whose BF(A) becomes +2 after insertion. To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right child, and B_R and A_R to be the left and right subtrees of A. Observe how the rotation results in a balanced tree.

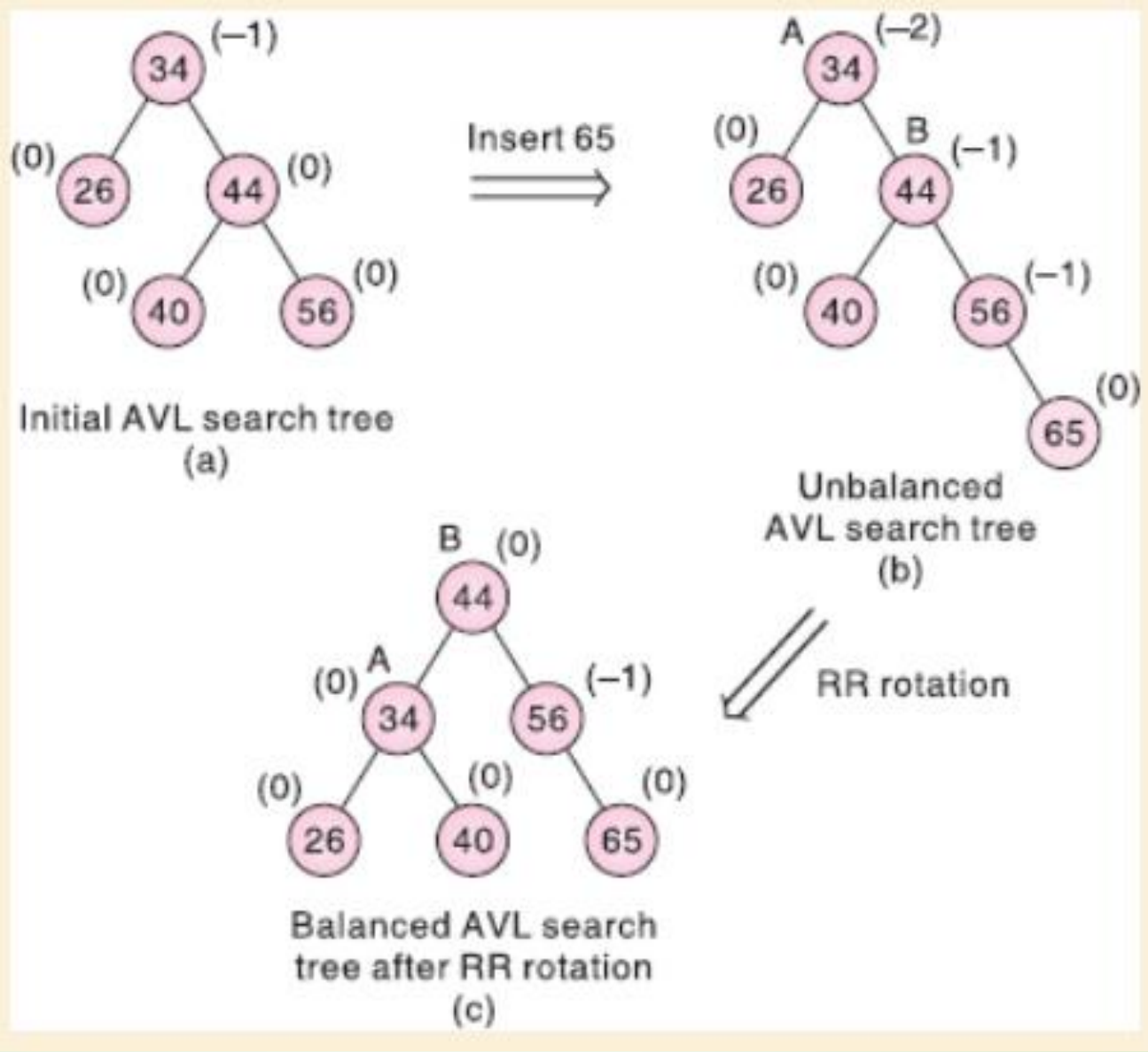
LL Rotation


(+1)
96
(0)
85
(0)
110
(0)
64
90
(0)
Initial AVL search tree
(a)
Insert 36
A
(+2)
96
(+1)
B
85
(0)
110
(+1)
64
90
(0)
(0)
36
Unbalanced
AVL search tree
(b)
LL rotation
B
(0)
85
(+1)
A
(0)
64
96
(0)
(0)
(0)
36
90
110
Balanced AVL search
tree after LL rotation
(c)

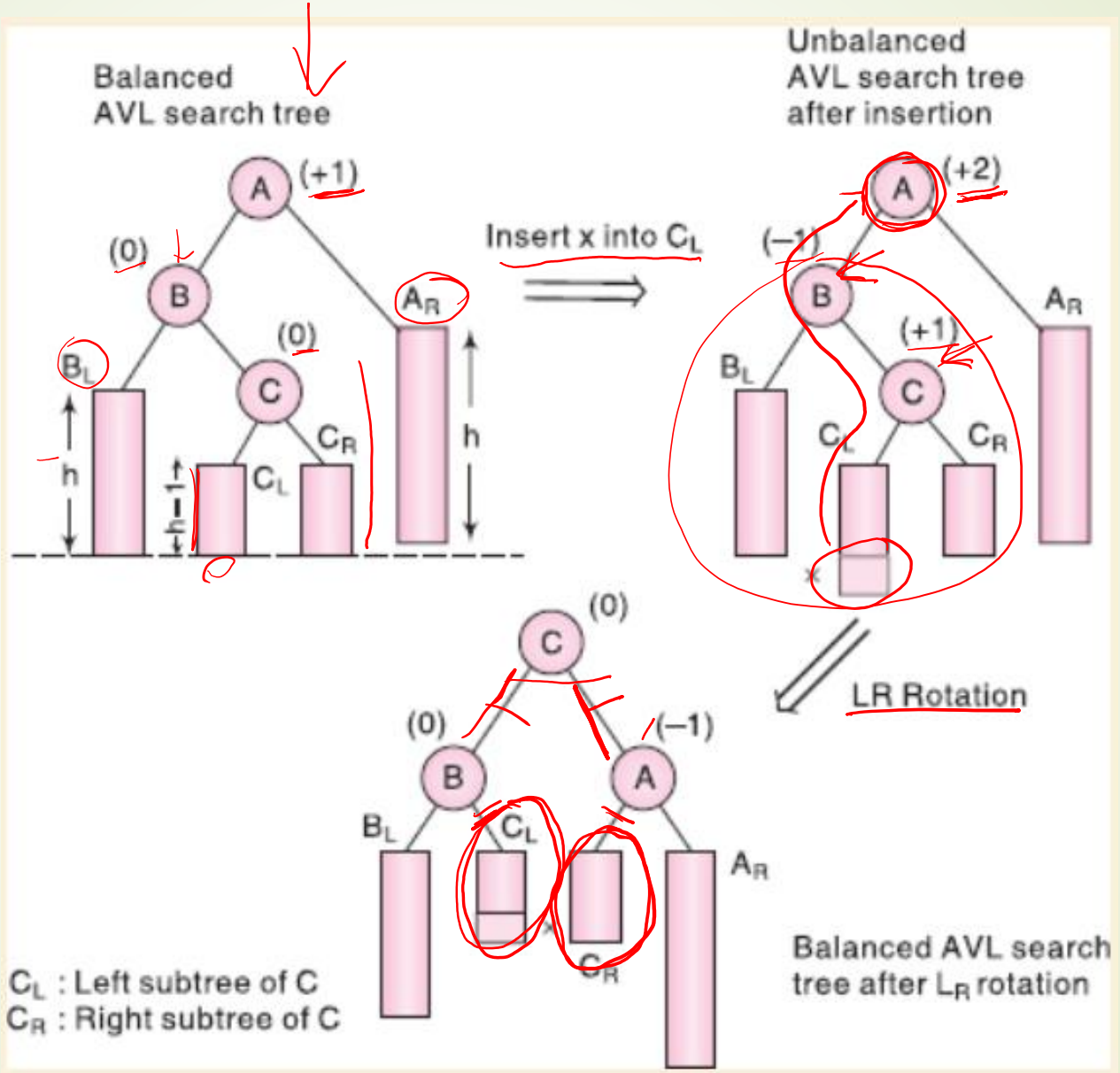
RR Rotation


Balanced AVL search tree
A
(–1)
B
(0)
A_L
B_L
B_R
h
h
Insert × into B_R
Unbalanced AVL search tree after insertion
A
(–2)
B
(–1)
A_L
B_L
B_R
h + 1
×
RR rotation
B
(0)
A
(0)
A_L
B_L
B_R
h
h + 1
X
Balanced AVL search tree after rotation

RR Rotation: Example

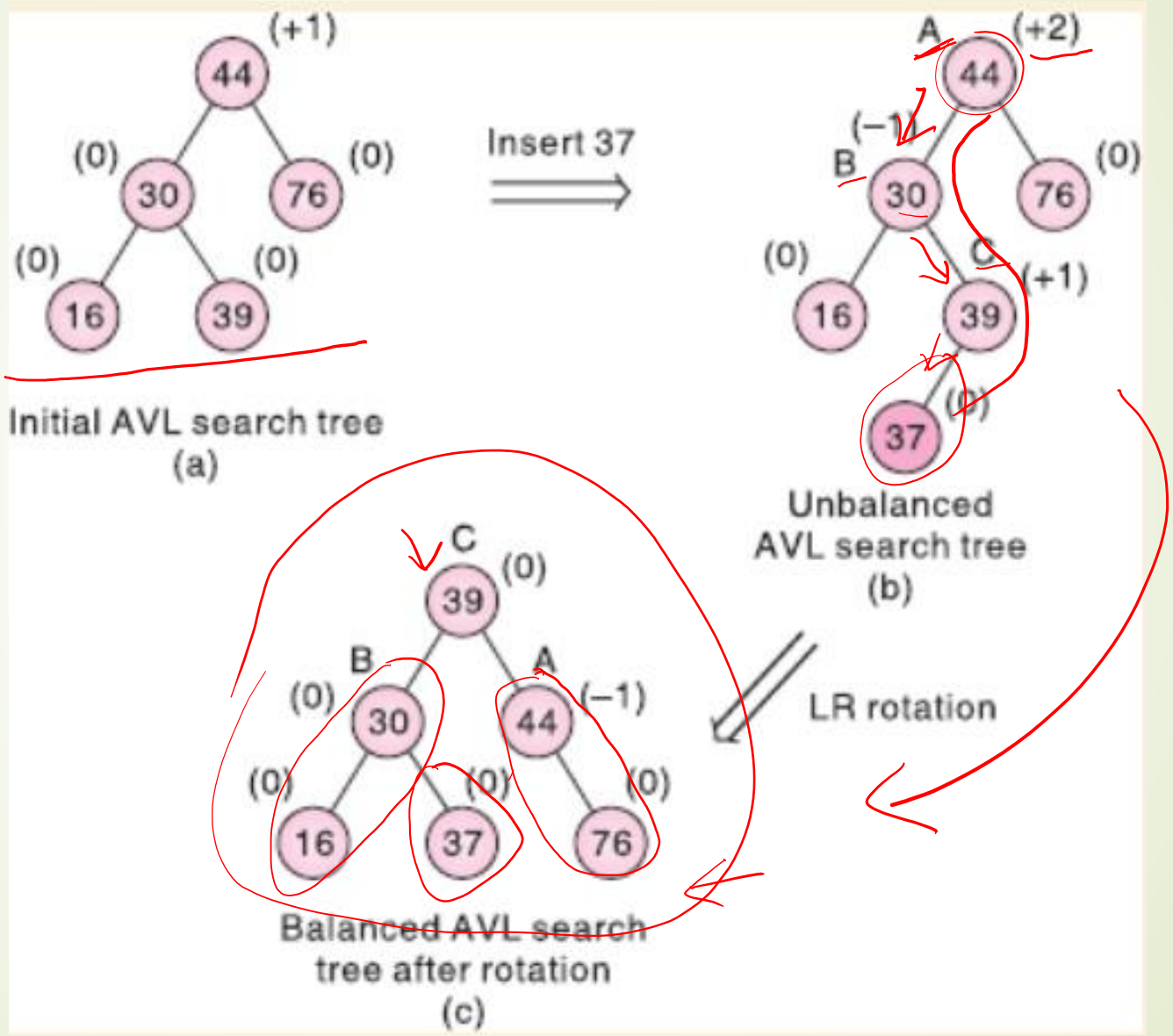


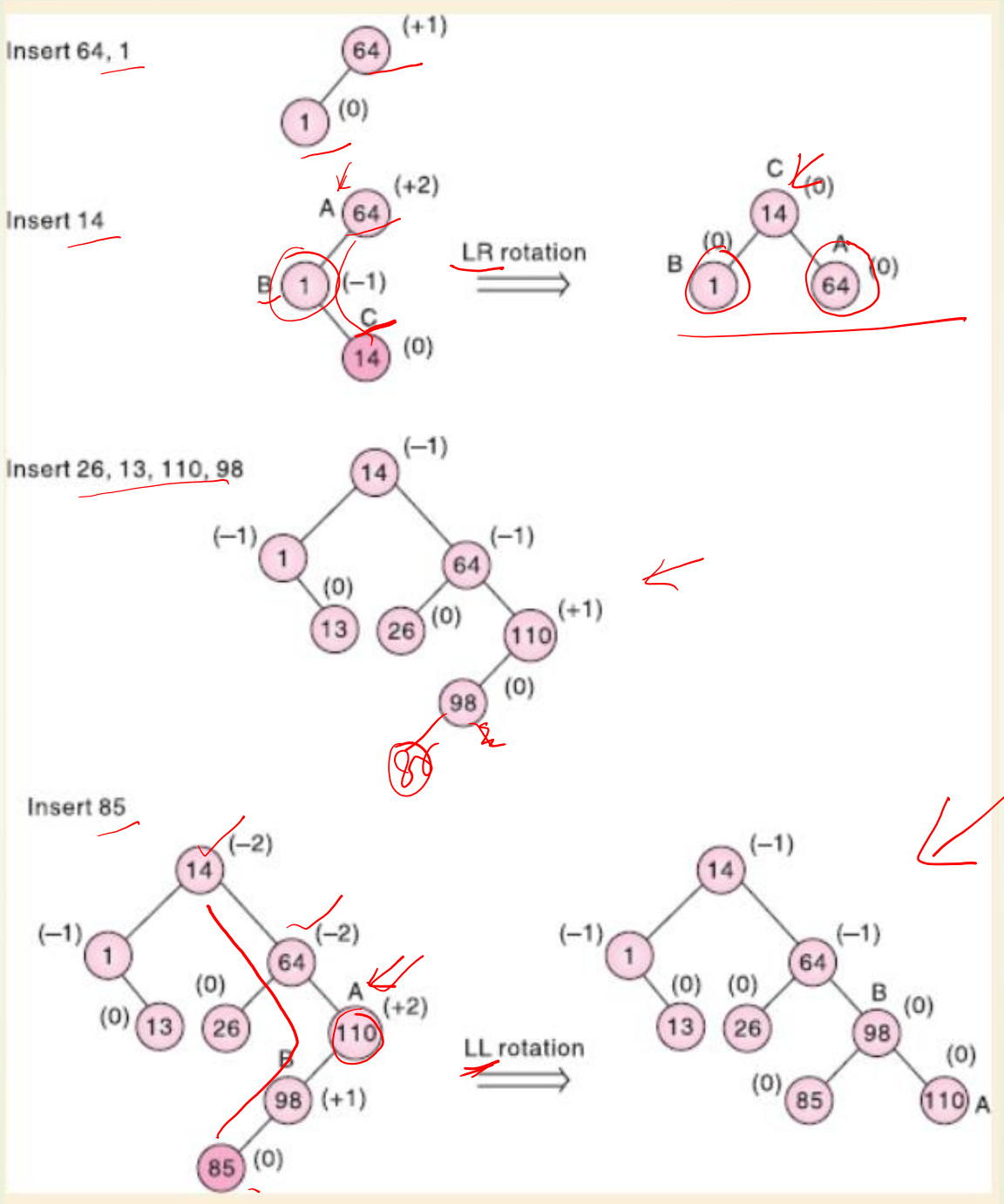
LR Rotation



LR

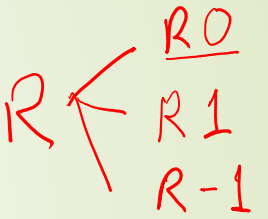
LR Rotation: Example

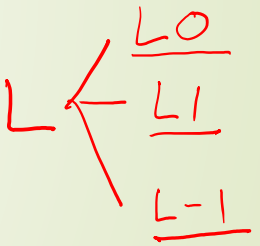



Insert 64, 1
64
(+1)
1
(0)
Insert 14
A
64
(+2)
B
1
(−1)
C
14
(0)
LR rotation
C
14
(0)
B
1
(0)
A
64
(0)
Insert 26, 13, 110, 98
14
(−1)
(−1)
1
(−1)
64
(0)
13
26
(0)
110
(+1)
98
(0)
Insert 85
14
(−2)
(−1)
1
(−2)
64
(0)
(0)
13
26
A
110
(+2)
B
98
(+1)
85
(0)
LL rotation
14
(−1)
(−1)
1
(−1)
64
(0)
(0)
13
26
B
98
(0)
(0)
85
(0)
110
A

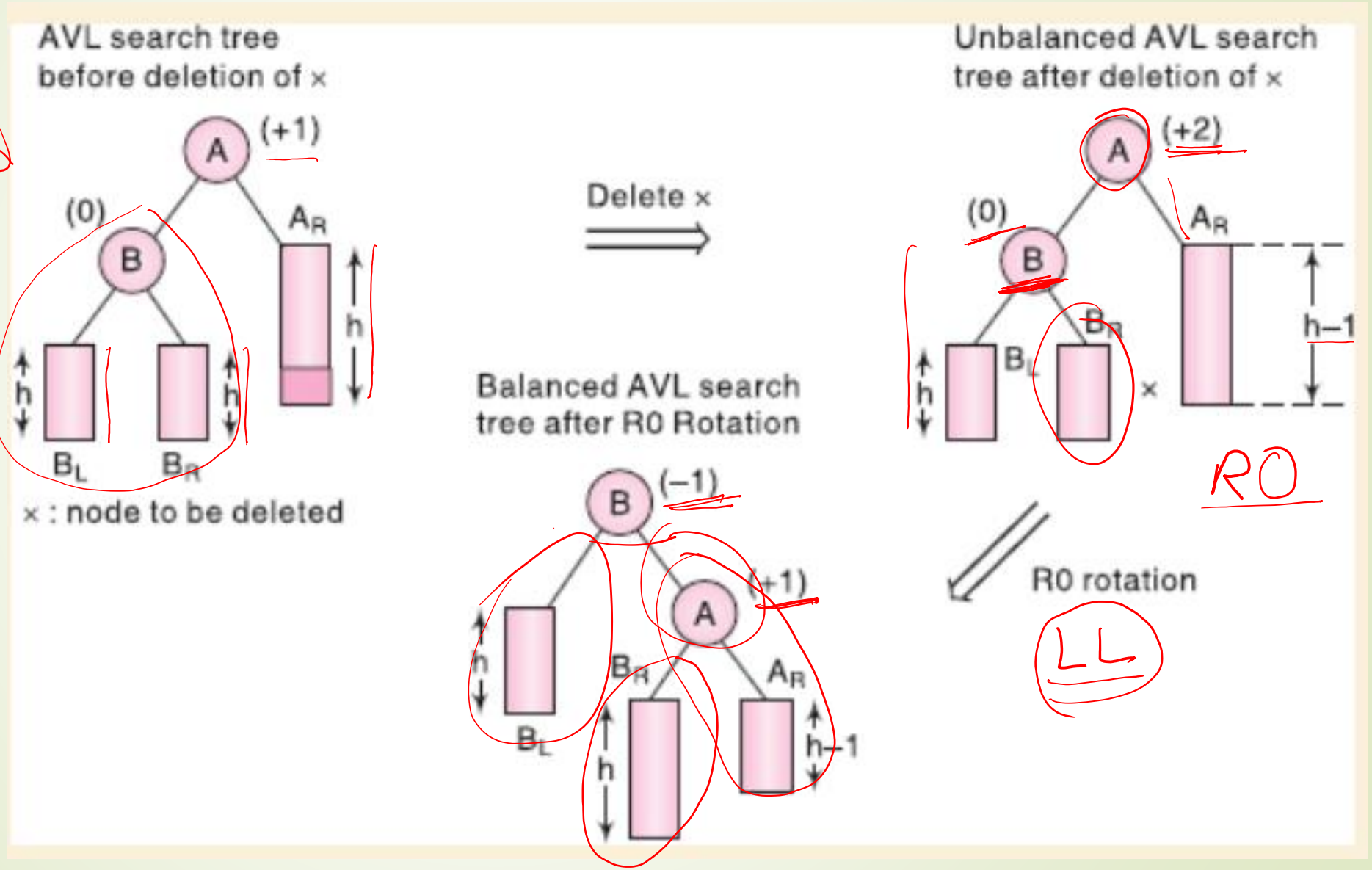
Deletion

- Identify node A for which balance factor is neither -1, 0 nor 1 and is nearest ancestor to the deleted node on the path from deleted node to the root
- Find if deleted node is in the left (L) or right (R) subtree of node A
- Find BF of node B which is root of the left or right subtree of node A
  - If deleted node is in left subtree of A then B is root of A's right subtree
  - If deleted node is in right subtree of A then B is root of A's left subtree
- Three R rotations
  - Rotation R0: deleted node is in right subtree of node A, BF(B)=0
  - Rotation R1: deleted node is in right subtree of node A, BF(B)=1
  - Rotation R-1: deleted node is in right subtree of node A, BF(B)=-1
- Three L Rotations
  - Mirror image of R rotations

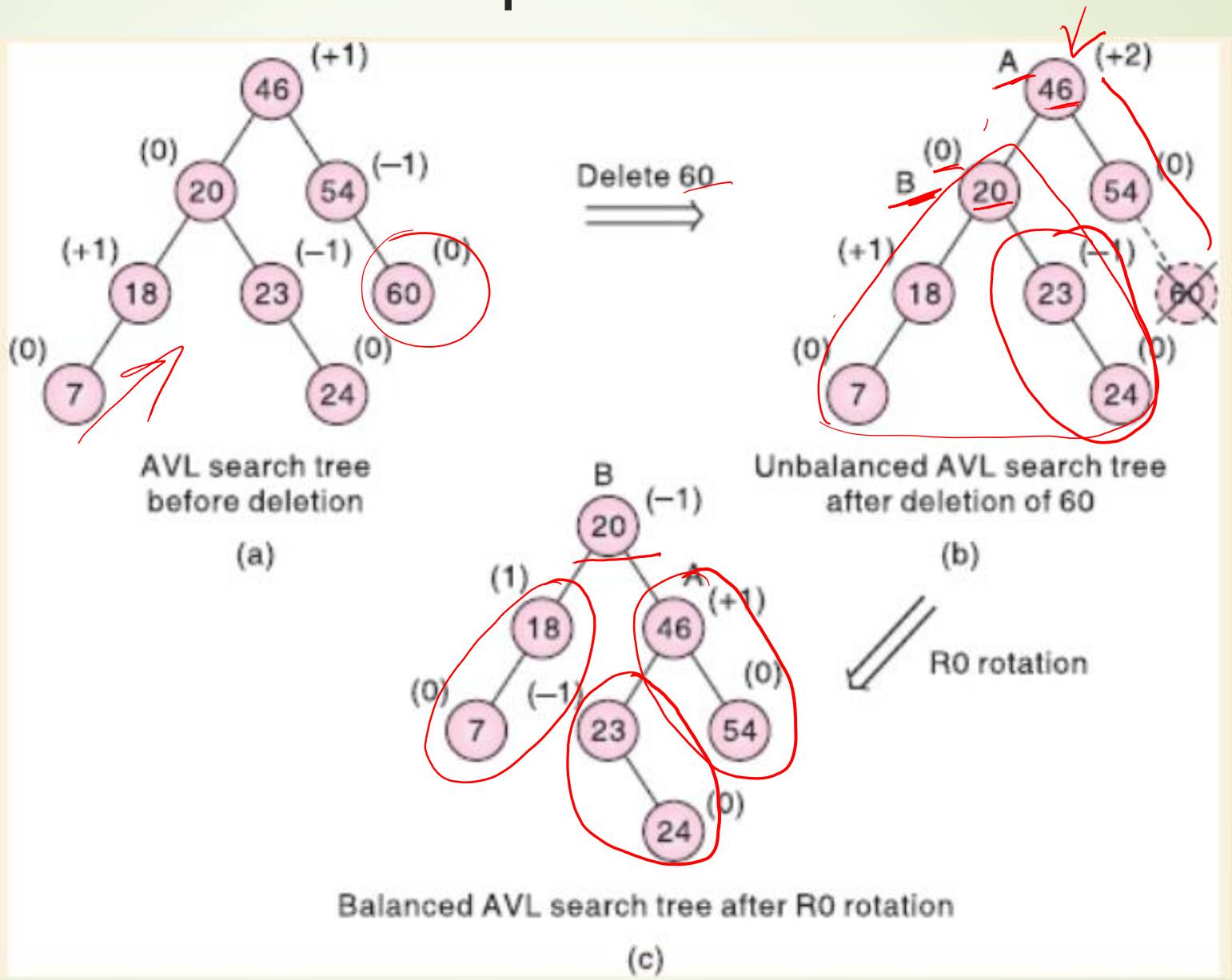




R0 Rotation

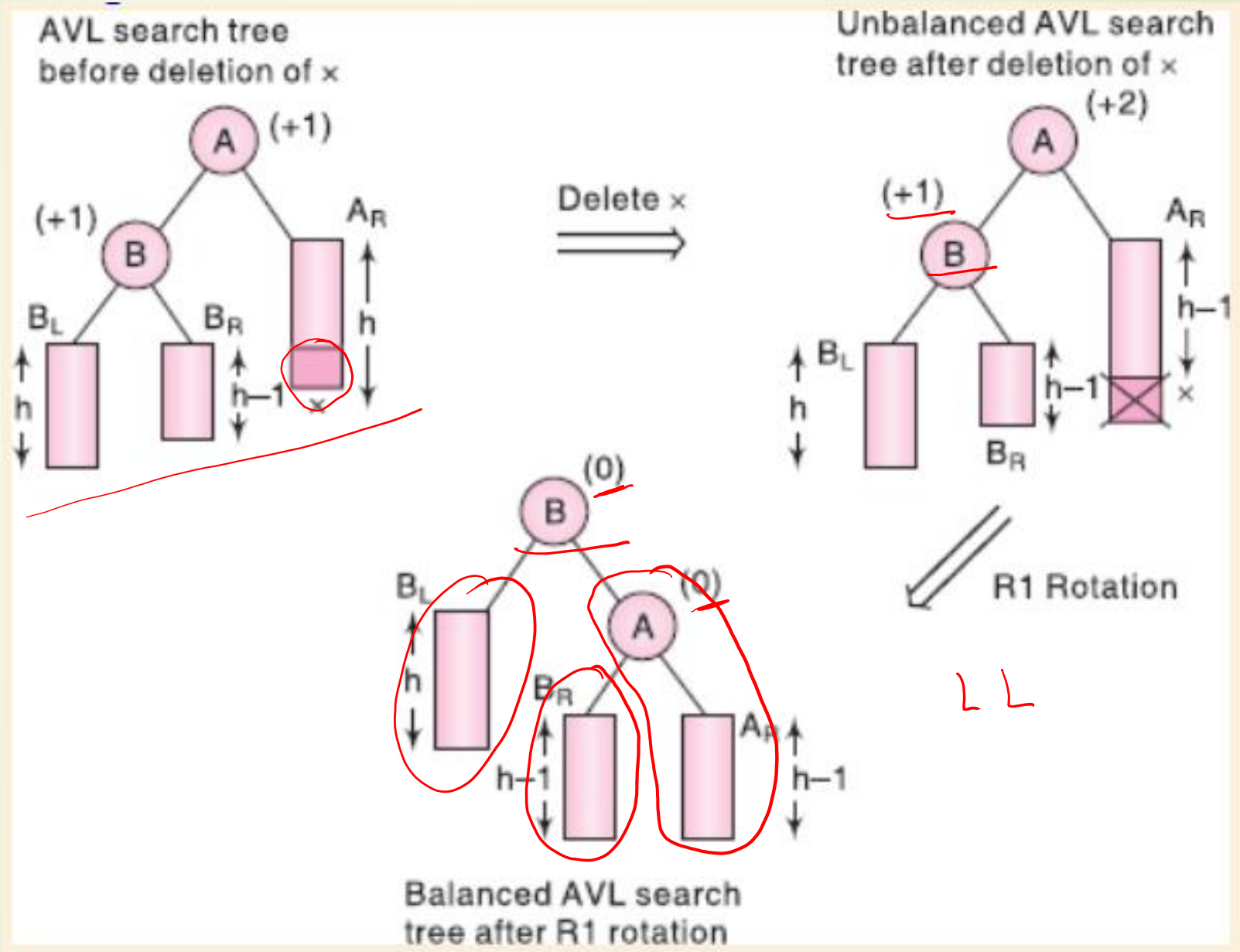

AVL search tree
before deletion of ×
A
(+1)
(0)
B
A_R
h
h
h
B_L
B_R
× : node to be deleted
Delete ×
Balanced AVL search
tree after R0 Rotation
B
(−1)
A
(+1)
h
B_L
B_R
A_R
h
h−1
R0 rotation
Unbalanced AVL search
tree after deletion of ×
A
(+2)
(0)
B
A_R
h
B_L
B_R
×
h−1
R0
LL

R0 Rotation: Example

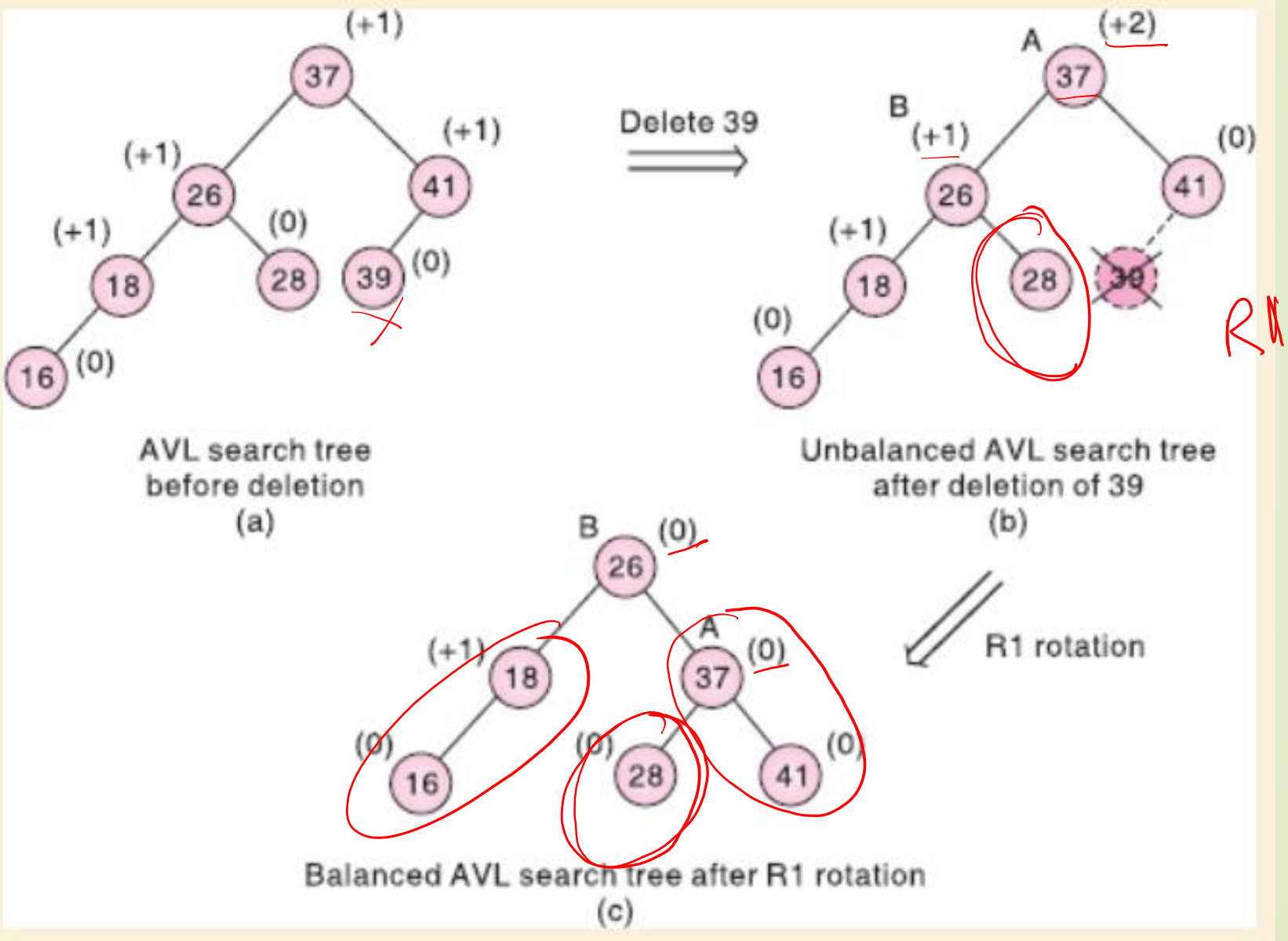


R0

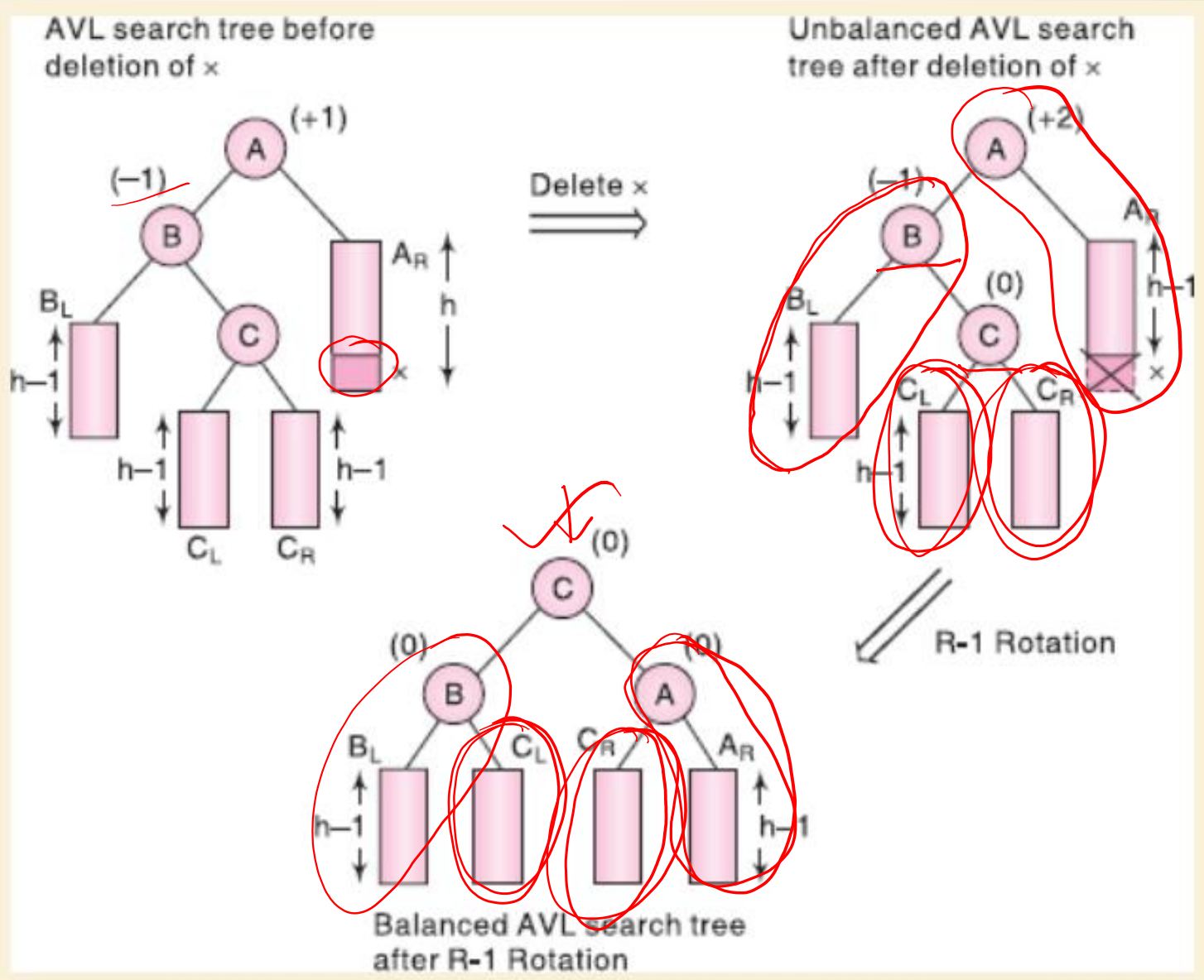
R1 Rotation


AVL search tree
before deletion of x
A
(+1)
(+1)
B
A_R
B_L
B_R
h
h
h–1
Delete x
Unbalanced AVL search
tree after deletion of x
(+2)
A
(+1)
B
A_R
h–1
B_L
h
h–1
x
B_R
R1 Rotation
(0)
B
B_L
(0)
A
h
B_R
A_R
h–1
h–1
Balanced AVL search
tree after R1 rotation

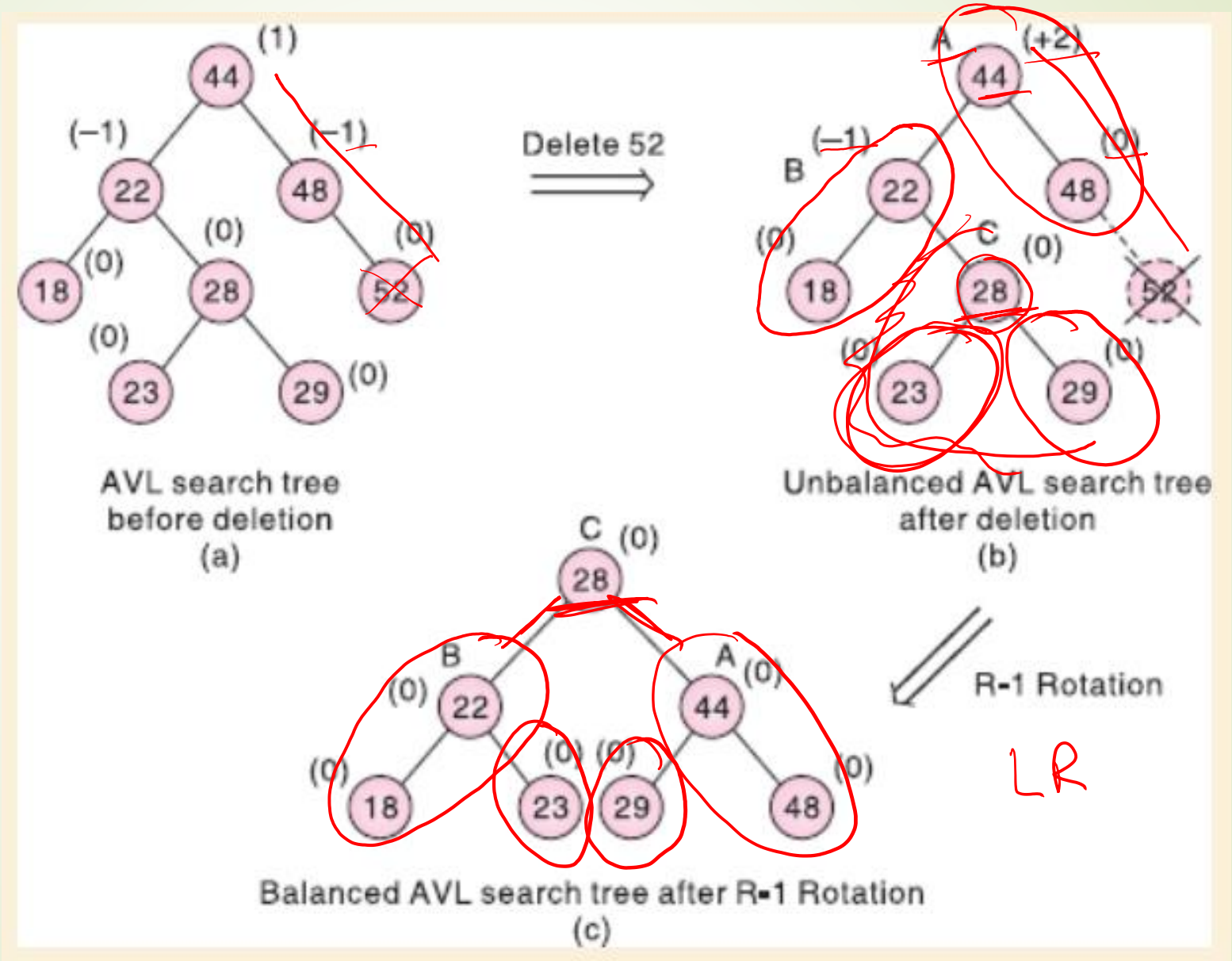
R1 Rotation: Example



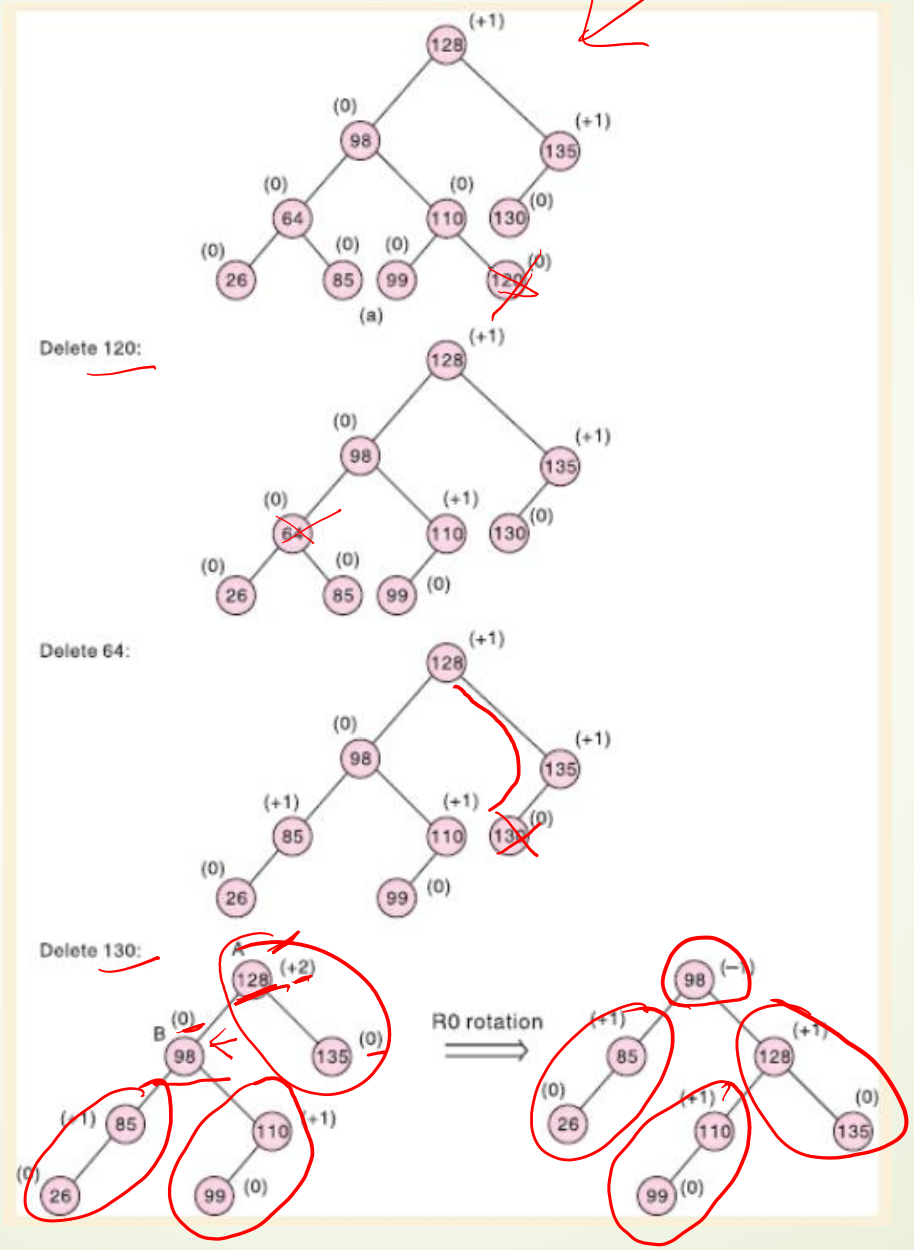
R-1 Rotation


AVL search tree before deletion of x
Unbalanced AVL search tree after deletion of x
Delete x
A
B
C
(+1)
(−1)
(0)
(+2)
B_L
C_L
C_R
A_R
h
h−1
x
R-1 Rotation
Balanced AVL search tree after R-1 Rotation

R-1 Rotation: Example



Deletion Example



Other Balanced BST

- Weight balanced BST
  - Number of nodes in the subtree rooted at a node is the weight of the node
- Red-black tree
  - Two colors: red and black

Multi-way search trees

- Generalized version of binary search trees
- Number of comparisons in searching is of the order of the height of tree i.e $O(\log_m n)$ (where m is the maximum number of children a node can have)
- $m>2$ so $\log_m n < \log_2 n$
- Max number of comparisons in worst case is still O(n) if tree is skewed
- Balanced multi way search tree
  - B-tree
  - B+-tree
  - B*-tree

Huffman Algorithm

- Huffman algorithm is a method for building an extended binary tree (2-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

- 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

- Let us take any four characters and their frequencies, and sort this list by increasing frequency.

- Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

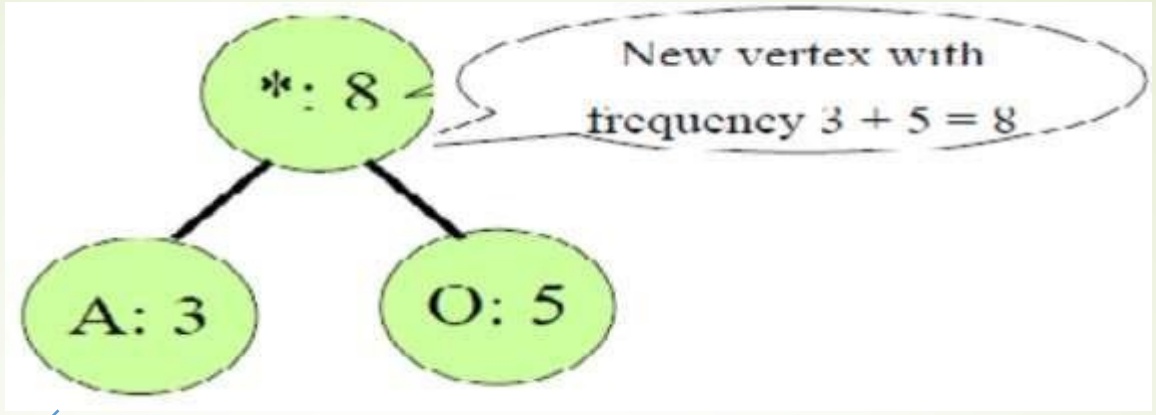
character	frequencies
E	10
T	7
O	5
A	3

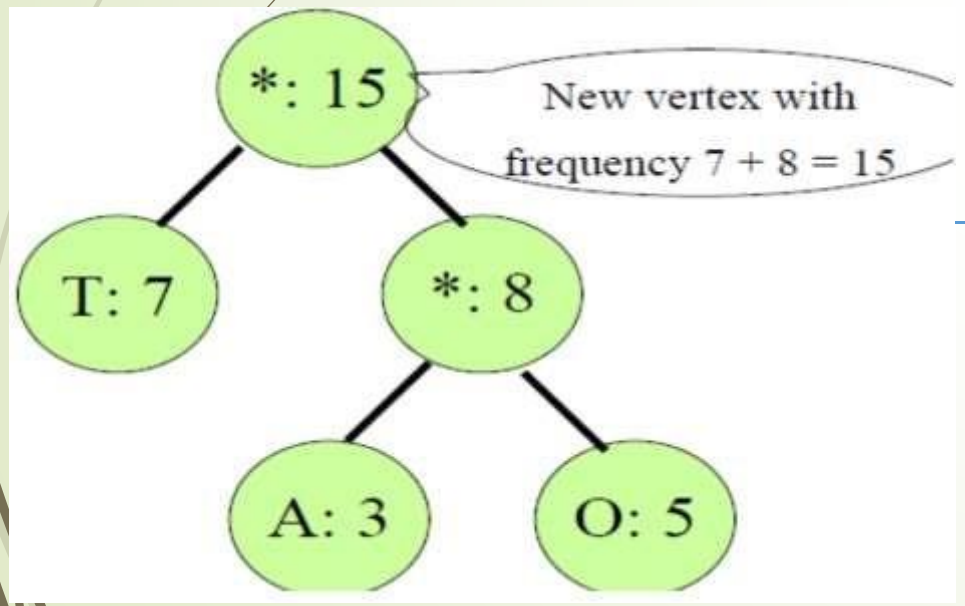
sort →

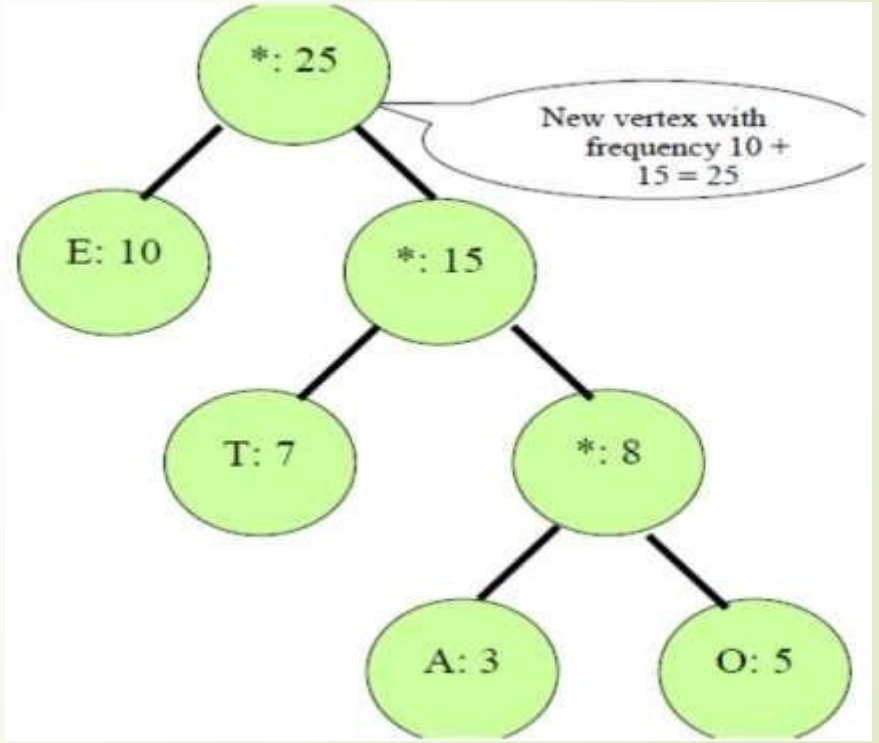
Character	frequencies	code
A	3	00
O	5	01
T	7	10
E	10	11

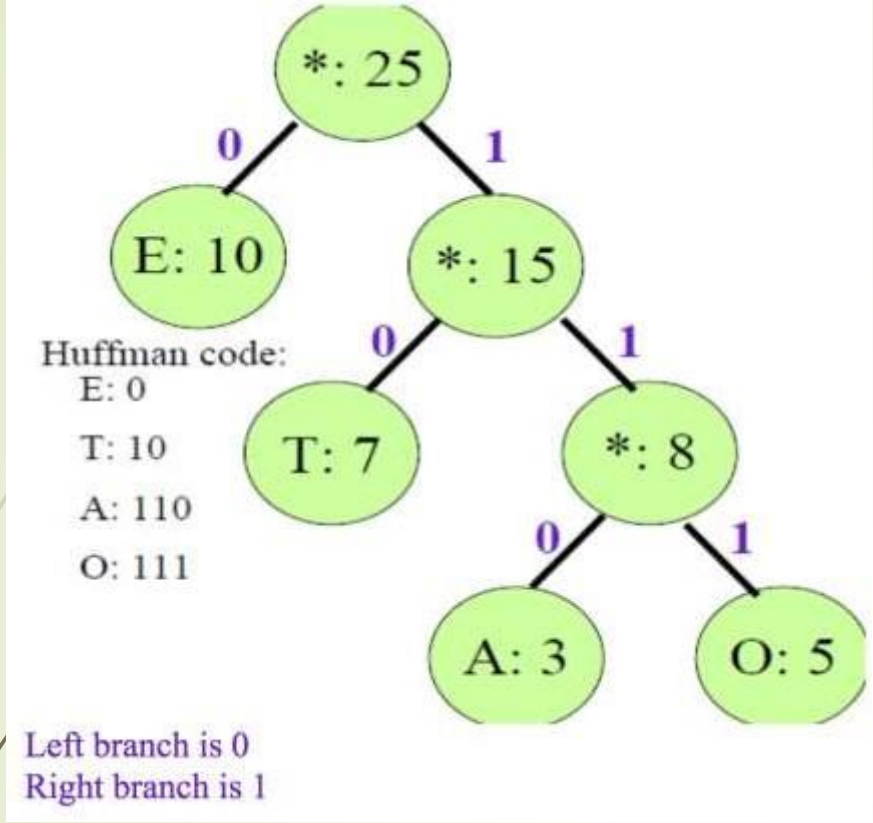
- Here before using Huffman algorithm the total number of bits required is:

  nb=3*2+5*2+7*2+10*2 =06+10+14+20 =50bits


*: 8
New vertex with
frequency 3 + 5 = 8
A: 3
O: 5


*: 15
New vertex with
frequency 7 + 8 = 15
T: 7
*: 8
A: 3
O: 5


*: 25
New vertex with
frequency 10 +
15 = 25
E: 10
*: 15
T: 7
*: 8
A: 3
O: 5



Character	frequencies	code
A	3	110
O	5	111
T	7	10
E	10	0

- Thus after using Huffman algorithm the total number of bits required is

nb=3*3+5*3+7*2+10*1 =09+15+14+10 =48bits

i.e

(50-48)/50*100%=4%

Since in this small example we save about 4% space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies we can save a lot of space

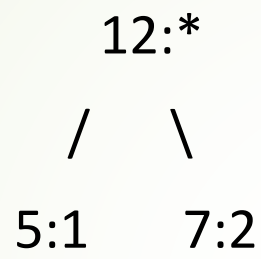
Huffman Algorithm

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

Value	Frequencies
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:

10:3

12:*

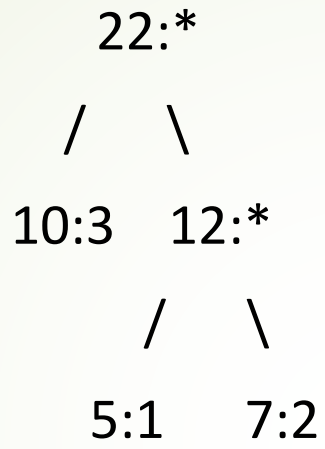
15:4

20:5

45:6

Huffman Algorithm

- You then repeat the loop, combining the two lowest elements. This results in:



- 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:

15:4

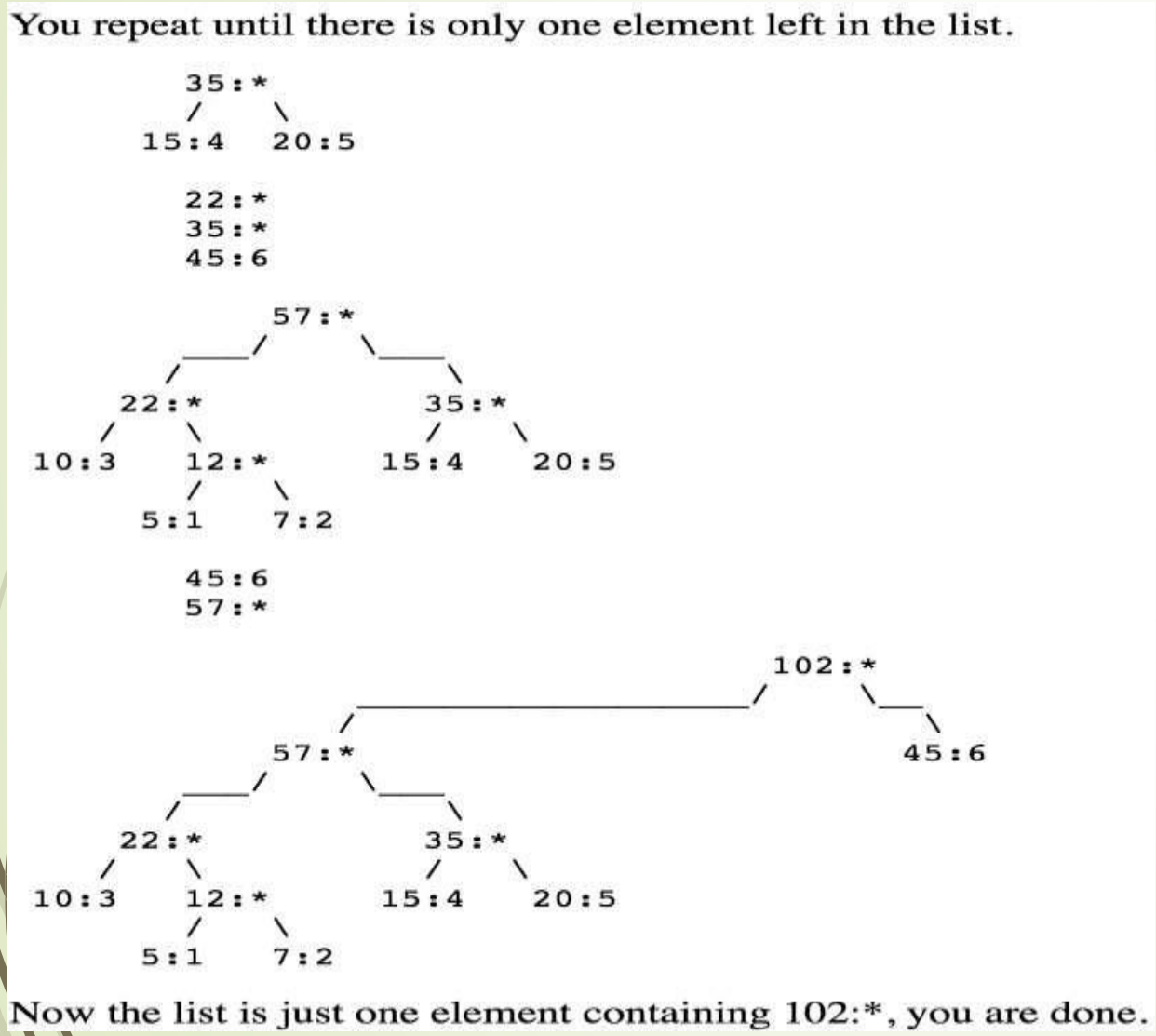
20:5

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.

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