CSE-215

12.7.20

Trees

Linear lists are useful for serially ordered data

→ $e_1, e_2, e_3 - - e_n$

→ Days of week

→ Months in a year

→ Students in a class

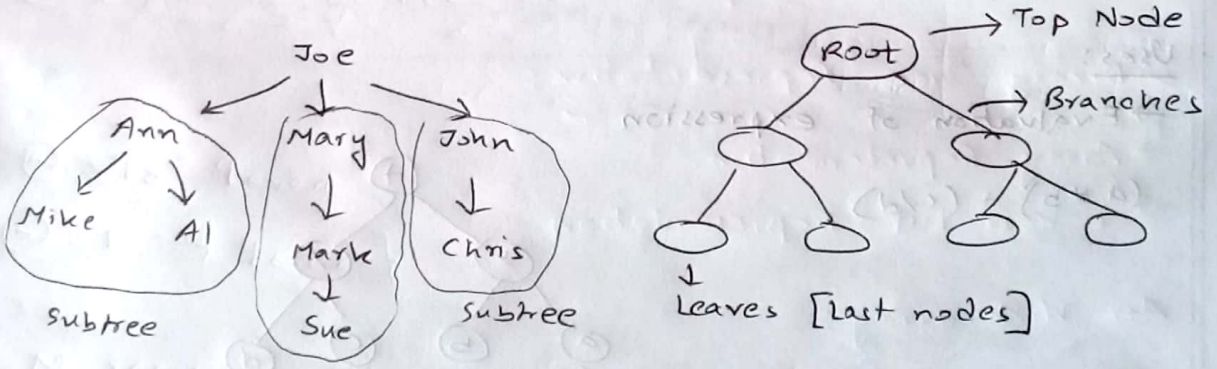
Trees are useful for heirarchical data structure

→ Joe's descendants

→ Corporate structure

→ Government subdivisions

→ Software structure



Note: ER Diagram of Database is like tree

→ A tree t is a finite nonempty set of elements.

→ The topnode is root.

→ The remaining elements, if any, are ~~parti~~ partitioned into trees, which are called the subtrees of t.

→ Leaves are nodes with no descendants

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

Node Degree:

The number of children a node has is the node degree

Tree Degree:

The maximum of node degrees is the Tree Degree

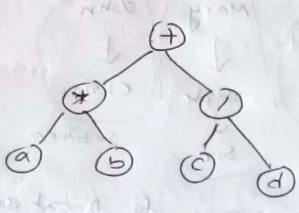
*) Binary Tree:

→ All nodes in a binary tree have 0, 1, or 2 children.
→ Each node can have two subtrees left and right subtree.
→ A tree with one node is also a binary tree.

Uses:

→ Evaluation of expression

(a*b) + (c/d)

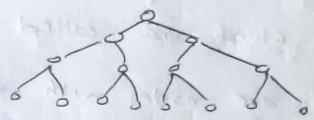


Properties:

① Binary tree with n elements has exactly (n-1) edges.

② A binary tree of height h, h>=0 has atleast h+1 and atmost $2^{h+1}-1$ elements in it.

Atleast h+1 = 4 ~~edges~~ nodes



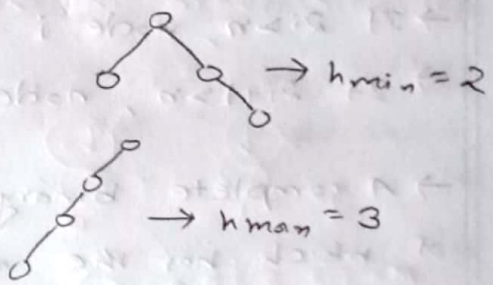
Atmost $2^{h+1}-1$ = 15 nodes

③ Height of a binary tree that contains n elements $n >= 0$, is atleast $\lceil \log_2(n+1) \rceil - 1$ and atmost $n-1$

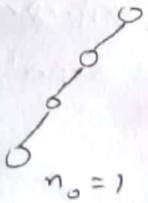
For $n = 4$,

$h_{min} = \lceil \log_2(5) \rceil - 1 = 3 - 1 = 2$

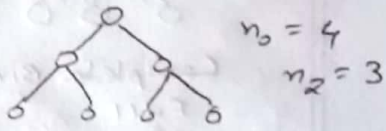
$h_{max} = 4 - 1 = 3$



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



$\therefore n_0 = n_2 + 1$ [Proved]



*) Full Binary Tree:

A full binary tree of height h has exactly $2^{h+1} - 1$ nodes.

Full Binary Tree

$h = 2$

$\therefore$ nodes $= 2^{2+1} - 1 = 2^3 - 1 = 8 - 1 = 7$

→ Who is parent of 4? [Node Number Property]

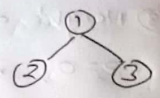
Ans: $\lfloor i/2 \rfloor = \lfloor 4/2 \rfloor = \lfloor 2 \rfloor = 2$

for $i = 5$, $\lfloor 5/2 \rfloor = \lfloor 2.5 \rfloor = 2$

For $i = 1$, $\lfloor 1/2 \rfloor = \lfloor 0.5 \rfloor = 0$ [No parents]

∴ If a node is i,

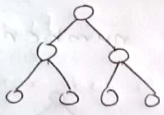
Left child = 2i , Right child = 2i + 1



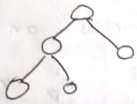
→ If 2i > n, node i has no left child

→ If 2i+1 > n, node i has no right child

→ A complete binary tree is a binary tree every level of which has the maximum possible number of nodes except possibly the last level.



~~Complete~~ Full Binary Tree



~~Full~~ Complete Binary Tree

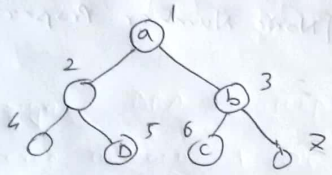
*) Binary Tree Representation :

→ Array Representation

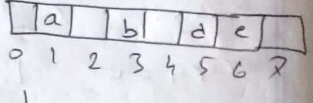
→ Linked List Representation

~~Arr~~ With array :

It is represented by storing each element at the array position corresponding the node number.



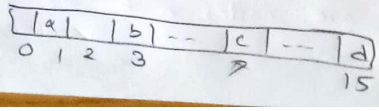
∴ Array ⇒



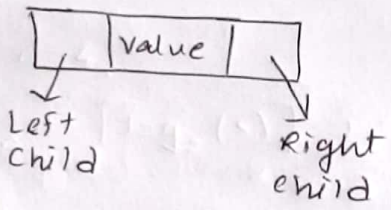
↓

Lot of wastage

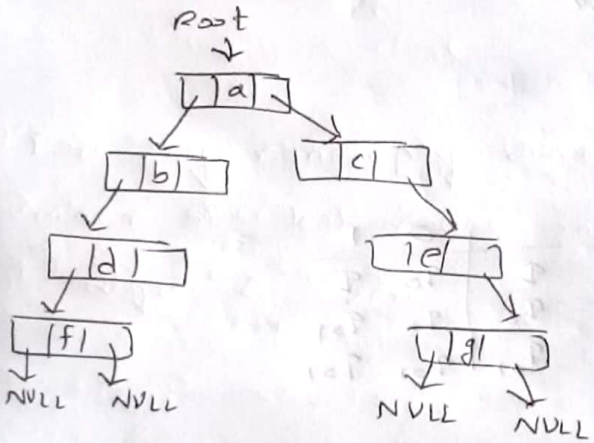
For right-skewed tree ⇒



*) With Linked List :-




```
struct node {
    int value;
    node * left_child;
    node * right_child;
}
```




```
Void Tree :: freenode (node* nd)
```

*) Binary Tree Traversal :-

→ Each element is visited exactly once.
→ During the visit, all actions are taken on the element

4 types of traversal :-

① Pre order traversal → Root, Left, & Right
② In order traversal → Left, Root, Right
③ Post order traversal → Left, Right, Root
④ Level order traversal

⑥

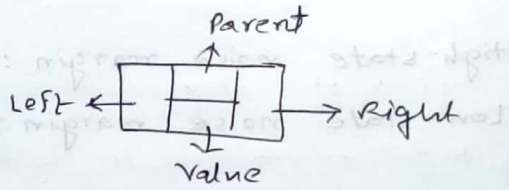
CW

CSE-215

15.7.20

*) Representation of Binary Tree:

```
struct node
{
    int value;
    node * left, * right, * parent;
}
```



*) Binary Search Tree:

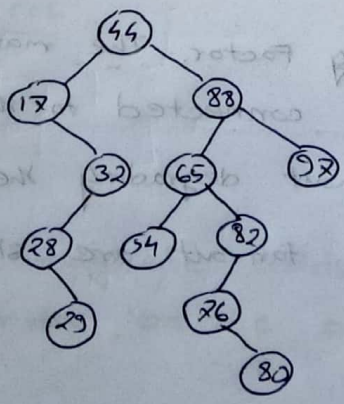
→ A special type of Binary Tree

→ Left < Parent < Right [Value]

→ Each subtree is a binary search tree

Insertion:

44, 17, 88, 32, 65, 97, 28, 54, 82, 29, 76, 80



Binary Search Tree

→ BST structure is dependent on insertion order.

*) Inorder Traversal in BST:

Traverse Left, then Root, then child

→ The traversal is ~~like~~ in sorted order

Traversal: 12 28 29 32 44 54 65 76 80 82 88 92

→ DFS call is enough for inorder traversal

```
void dfs (Node * node) {
    if(node == NULL)
        return;
    dfs(node->left);
    cout<< node->key<< " ";
    dfs(node->right);
}
```

*) Deletion in BST:

3 Cases :- ① Nodes with 0 children (Leaf Nodes) [simply free the node]
② Nodes with 1 child
③ Nodes with 2 children

~~void deleteNode~~

Nodes with 1 child → Make the successor and ancestor of the node point to each other.

Nodes with 2 children → Make the minimum node of the right subtree the root if we delete the root.

54 is inorder successor of 44.

CSE-215

19.7.20

Trees

Traversal = Pre order, In order, Post order → Order is always in respect of root

Pre order:

Root → Left → Right

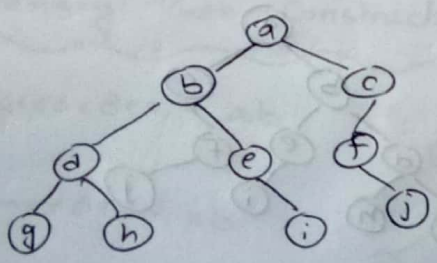
Inorder:

Left → Root → Right

Post order:

Left → Right → Root

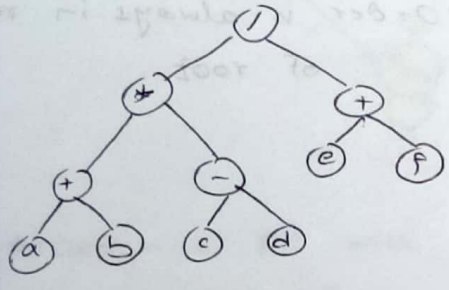
Pre-order:



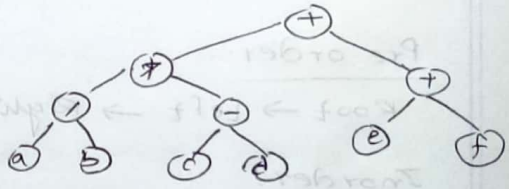
Traversal: a, b, d, g, h, e, i, c, f, j

```
void Tree::preOrder(Node *n)
{
    if (n != NULL)
    {
        cout << n->key << " ";
        preOrder(n->Left);
        preOrder(n->Right);
    }
}
```

Note: Preorder of Expression Tree gives prefix form of expression.



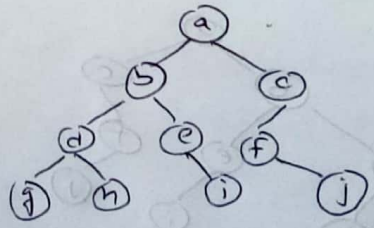
Traversal: / * + a b - c d + e f



An Expression: (((a/b)*(c-d))+(e+f))

Inorder Traversal:

```
void Tree::inorder(Node *n)
{
    if(n!=NULL)
    {
        inorder(n->left);
        cout<<n->key<<" ";
        inorder(n->right);
    }
}
```



Traversal: g, d, h, b, e, i, a, f, j, c

Postorder Traversal:

```
void Tree::postorder(Node *n)
{
    if(n!=NULL)
    {
        postorder(n->left);
        postorder(n->right);
        cout<<n->key<<" ";
    }
}
```

Traversal: g, h, d, i, e, b, j, f, c, a

Note: Postorder of expression tree gives postfix form of expression and inorder gives infix expression.

*) Level Order:

Prints level-wise. Apply bfs.

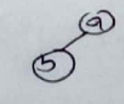
Traversal: a, b, c, d, e, f, g, h, i, j

*) Time Complexity:

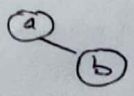
Time complexity of each of the four traversal algorithm is O(n) because each node is visited exactly once.

*) Binary Tree Construction:

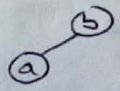
preorder - ab



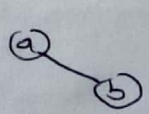
or



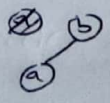
inorder - ab



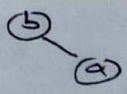
or



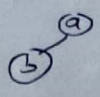
postorder - ab



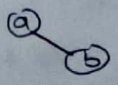
or



level order - ab



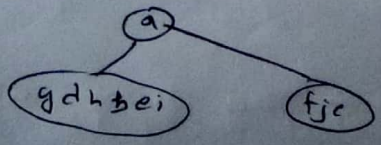
or



Inorder: g, d, h, b, e, i, a, f, j, c

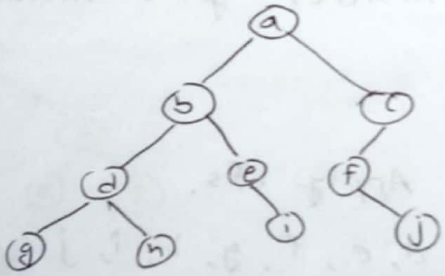
Preorder: a, b, d, g, h, e, i, c, f, j [Root always at first position]

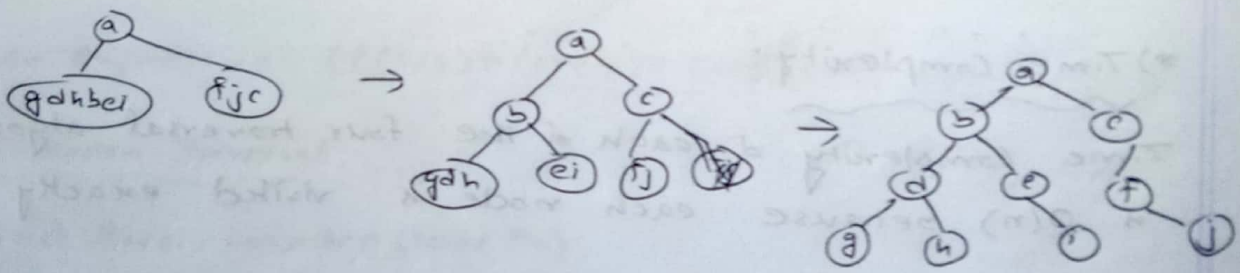
→ Scan the inorder and divide the tree as root found above.



(11)

In this way, we can make the tree





CW

CSE-215

Heaps

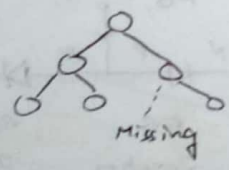
22.7.20

*) Tree Representation by Array:

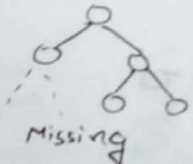
Complete Binary Tree:

→ Vertex filled from left to right

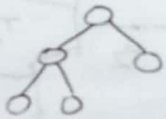
→ If in any level, a ~~left~~ vertex is missing from a node, we can call it a complete binary tree.



Not complete

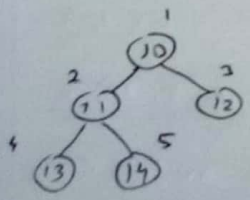


Not complete



Complete Binary Tree

→ To allocate the nodes in array, we will use bfs to locate the index.



5 vertices:

x	10	11	12	13	14
0	1	2	3	4	5

→ ith node's children are 2ith and (2i+1)th nodes.

→ Left child of i: 2*i

→ Right child of i: 2*i+1

→ Parent of i: $\lfloor i/2 \rfloor$

→ Root stays at 1st index

→ First characteristic of heap is that it is a complete binary tree.

→ We can determine from the array ~~whether the tree~~ how many leafnodes and internal nodes, a tree has.

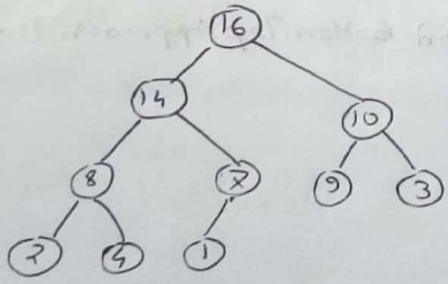
Heap:- ① Max Heap
② Min Heap

Max Heap:-

→ Parent Value >= Left child and Right child
→ All the subtrees maintain the above rule

Min Heap:-

Parent Value <= Left child and Right child



Max Heap

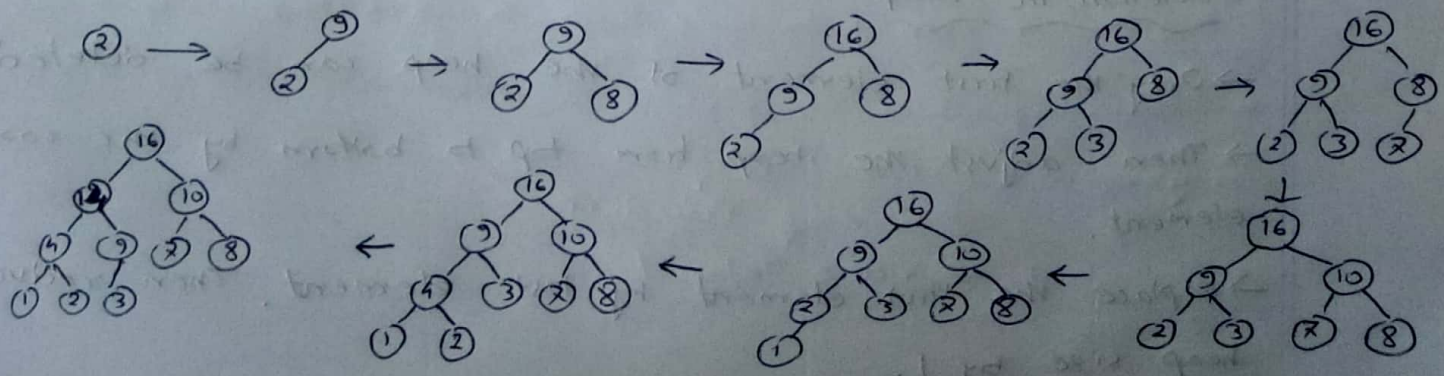
Array Representation:

X	16	14	10	8	7	9	3	2	4	1	X
0	1	2	3	4	5	6	7	8	9	10	11

Insertion in Heap:

→ The first element should be inserted at the ~~last~~ first initially. If the characteristics of heap breaks for that, we need to heapify.

Insertion: 2, 9, 8, 16, 3, 7, 10, 1, 4, 14



→ Heap insertion approach is bottom to top approach.

```
bool insert (int val){
    if (size == 100) return false;
    a[++size] = val;
    int i=size;
    while (i!=1) {
        if (a[i] > a[i/2])
            swap (a[i], a[i/2]);
        else
            break;
        i=i/2;
    }
    return true;
}
```

→ This portion can be put in another function like

```
void BottomTopApproach (int i)
```

Insertion Time Complexity : O(log n)

*) Effect of Insertion Order :

→ Unlike BST, the structure of heap doesn't depend on the insertion order of elements.

→ Height of heap is always fixed for a fixed number of vertices.

→ But order of children can change.

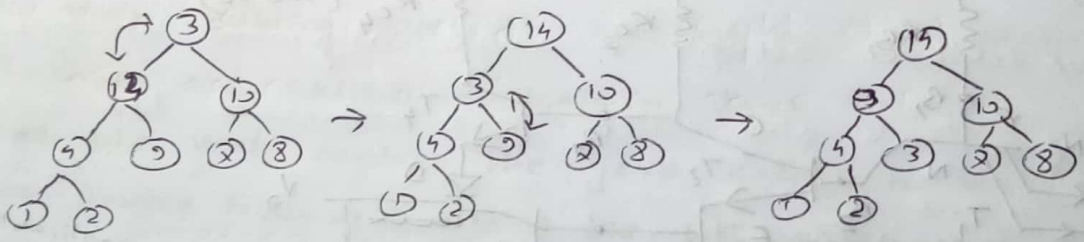
*) Deletion in Heap :

→ Only the first element of the heap can be deleted.

→ Then adjust the heap from top to bottom by the root element.

→ Replace the first element by last element. Then reduce the heap size by 1.

→ In top to bottom approach, find the max child. If we delete root which is 16, we do deletion and heapify in the following process



→ Top to Bottom Approach is called "Heapify" Function.

```
void topBottomApproach (int i){
    while (i<=size/2) {
        int leftchild = 2*i;
        int rightchild = 2*i+1;
        int max = a [leftChild];
        int maxIndex = leftChild;
        if (rightChild <= size && a [rightchild] > max) {
                                  → Possibility of RTE
            max = a [rightChild];
            maxInden = rightChild;
        }
        if (a [maxInden] > a [i])
            swap (a [maxInden], a [i]);
        else
            return;
        i = maxIndex;
}
```

(16)

CW

CSE-215

8.8.20

Heap Sort

Extracting all the roots sequentially produces the sorted sequen

Process:-

① Swap the root with last element. ~~Hit the last element~~

② Heapify

```
void sort()
{   int actualSize = size;
    while (size != 1)
    {
        deleteRoot();
    }
    size = actualSize;
    for(int i=1; i<=size; i++)
        cout<<a[i]<<" ";
}
```

Note: Heap sort destroys the arrangement of heap

Time Complexity: O(nlogn)

→ deleteRoot() is called n times and deleteRoot() function works for (logn) time. so, O(nlogn).

a) Build Heap:

Restoring a heap.

Checks from the last to first node.

~~Optimal~~

```
void buildHeap(){
    for(int i=size/2; i>=1; i--){
        topBottomAdjust(i);
    }
}
```

Complexity: O(nlogn)

Priority Queue

Enqueue in Queue : 10, 15, 4, 8, 9, 20, 12, 18

Dequeue in Queue : same Order

But Dequeue in Priority Queue : ~~15~~ 20, ~~15~~ 18, 12, 15, 10, 9, 8, 4

→ Priority Queue is implemented ~~in~~ with heap. (Max Heap)

→ Insert, ~~Top~~ → Inserts element

→ Top → Delets front value

→ Extract Max : Removes and returns the max element