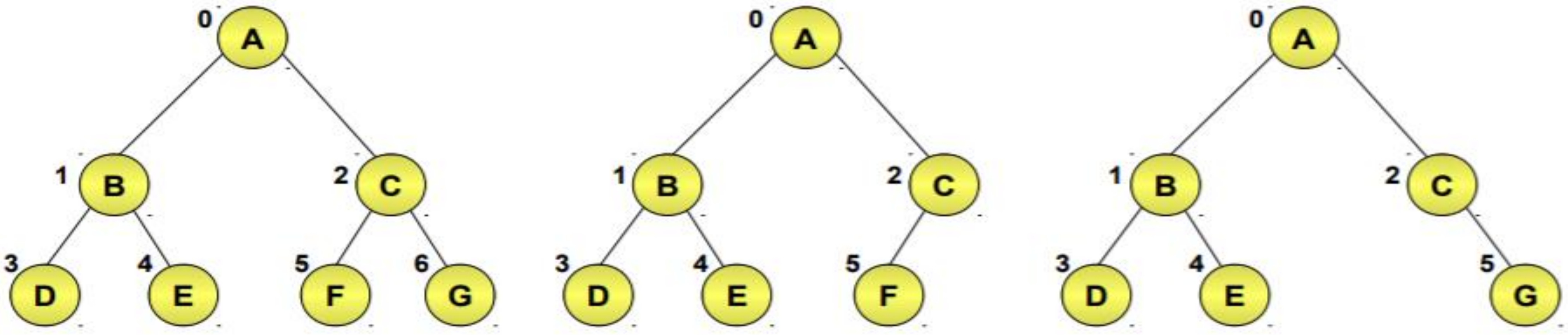
Sep23 : Day 6

Kiran Waghmare

CDAC Mumbai

Defining Binary Trees (Contd.)

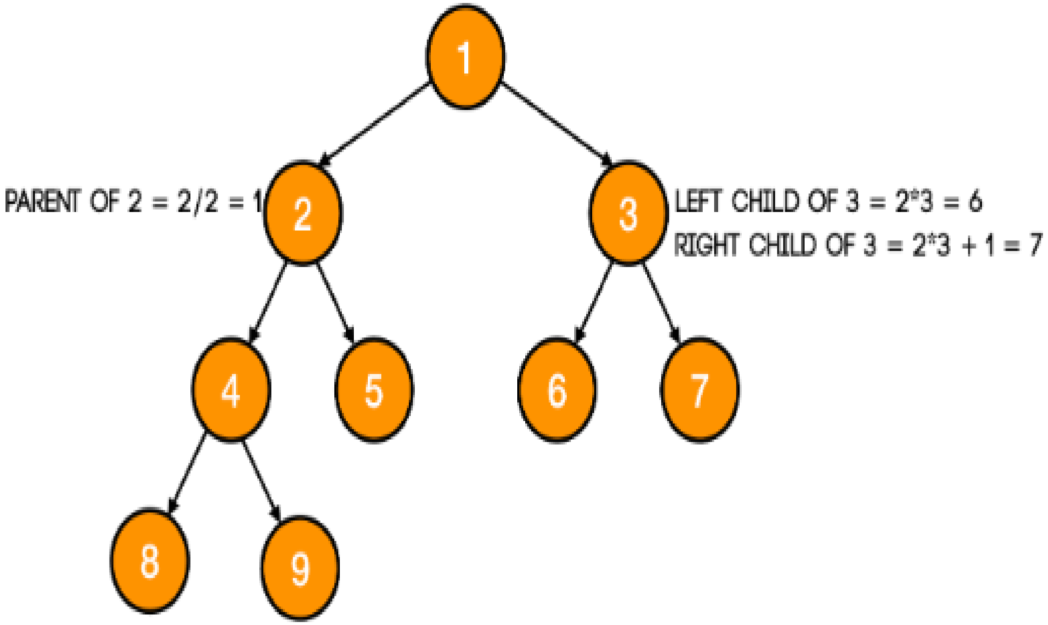
- Complete binary tree:
  - A binary tree with n nodes and depth d whose nodes correspond to the nodes numbered from 0 to n − 1 in the full binary tree of depth k.



Full Binary Tree | Complete Binary Tree | Incomplete Binary Tree

A complete binary tree also holds some important properties. So, let's look at them.

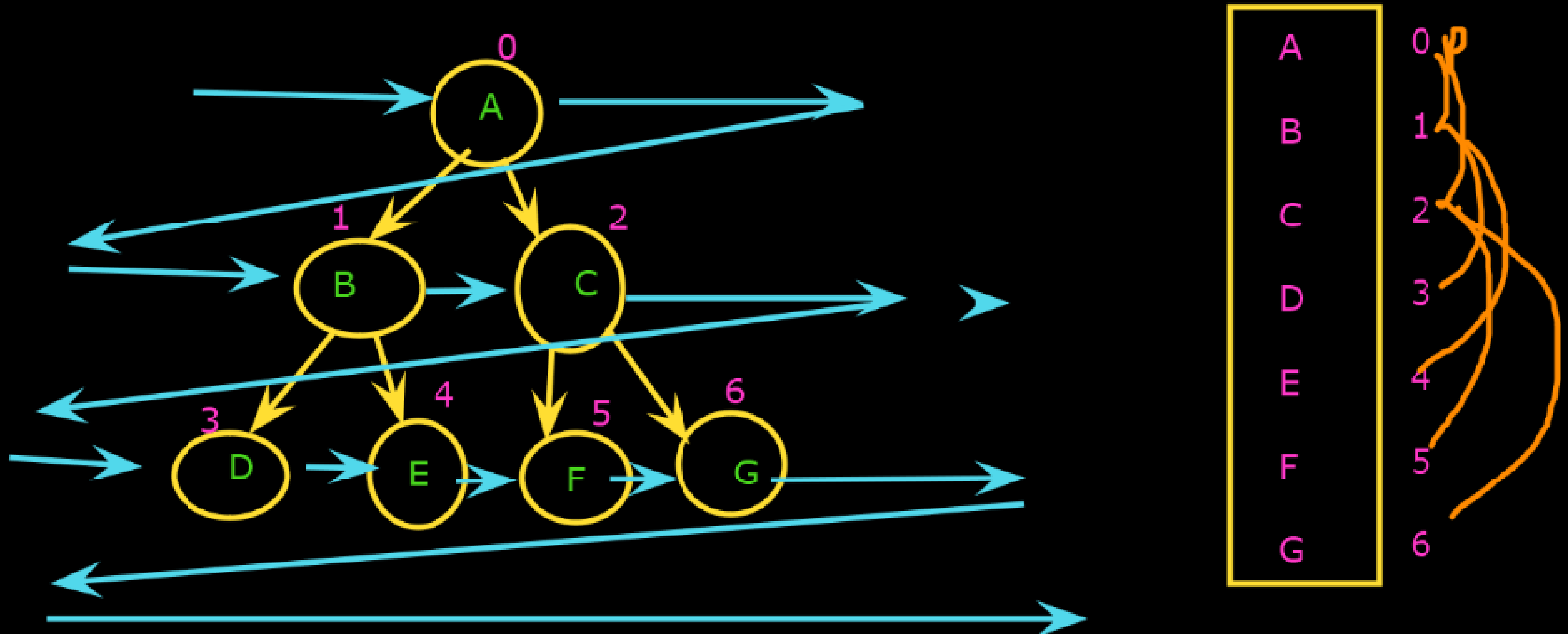
- The **parent of node i** is $\left\lfloor \frac{i}{2} \right\rfloor$. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is $2i$.
- The **right child of node i** is $2i + 1$



Binary Tree:

A binary tree is a tree in which every node has at most two children.

0,1,2,2



Array Representation of Binary Tree

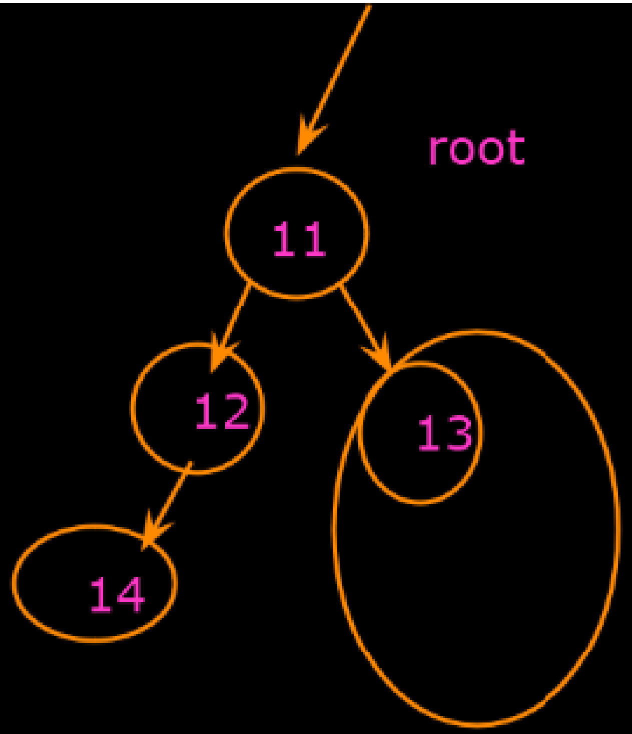
OPERATIONS ON TREES

Traversing a Binary Tree

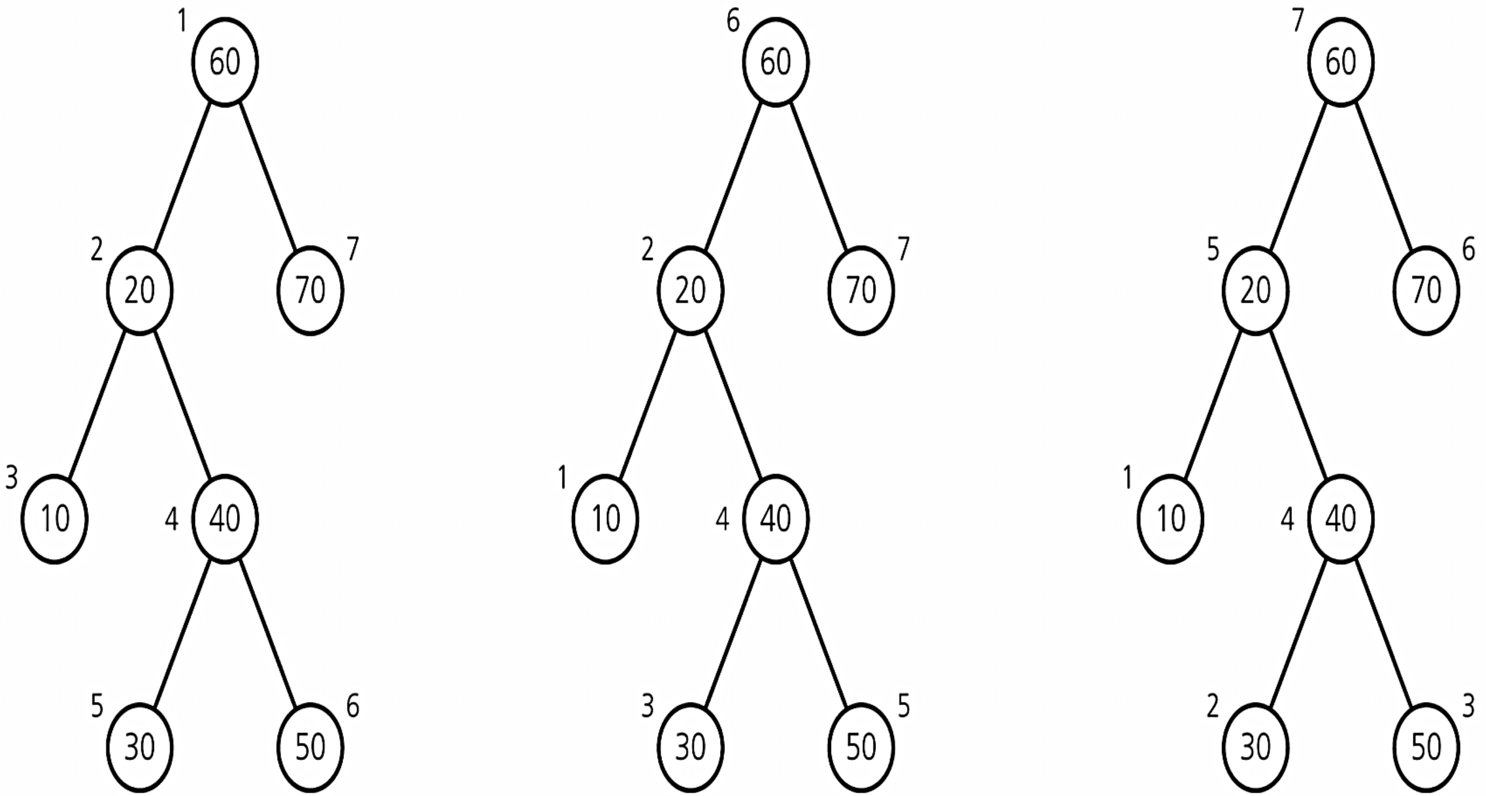
1)TRAVERSING

- You can implement various operations on a binary tree.
- A common operation on a binary tree is traversal.
- Traversal refers to the process of visiting all the nodes of a binary tree once.
- There are three ways for traversing a binary tree:
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal

```
    System.out.println(root.data+ " ");
    printInorder(root.right);

    }

    void printPreorder(Node root)
{
    if(root == null)
        return;

    System.out.println(root.data+ " ")
    printPreorder(root.left);
    printPreorder(root.right;

}

void printPostorder(Node root)
{
    if(root == null)
        return;
```



Binary Tree Traversals



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

InOrder(root) visits nodes in the following order:

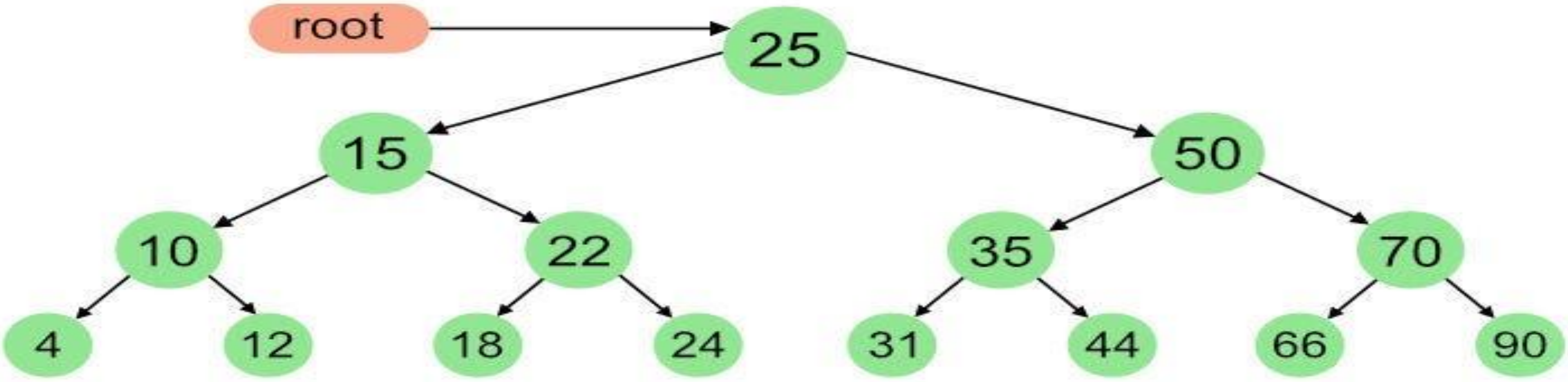
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

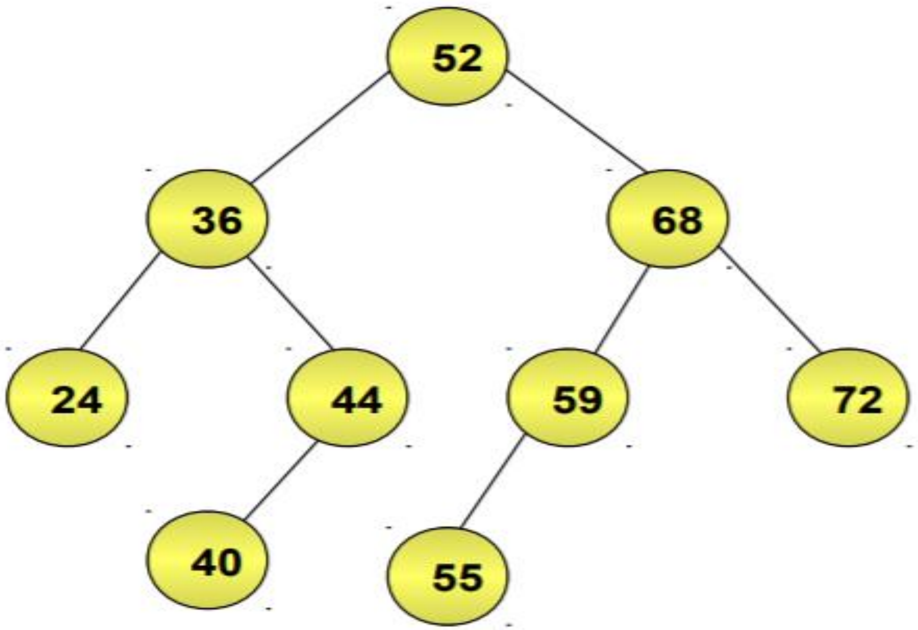
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



```java
class BT1{

    Node root;
    static class Node{
    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
    }

    BT1()
    {
        root = null;
    }
```

Binary Search Tree

- Binary search tree is a binary tree in which every node satisfies the following conditions:
  - All values in the left subtree of a node are less than the value of the node.
  - All values in the right subtree of a node are greater than the value of the node.
- The following is an example of a binary search tree.



Operations on a Binary Search Tree

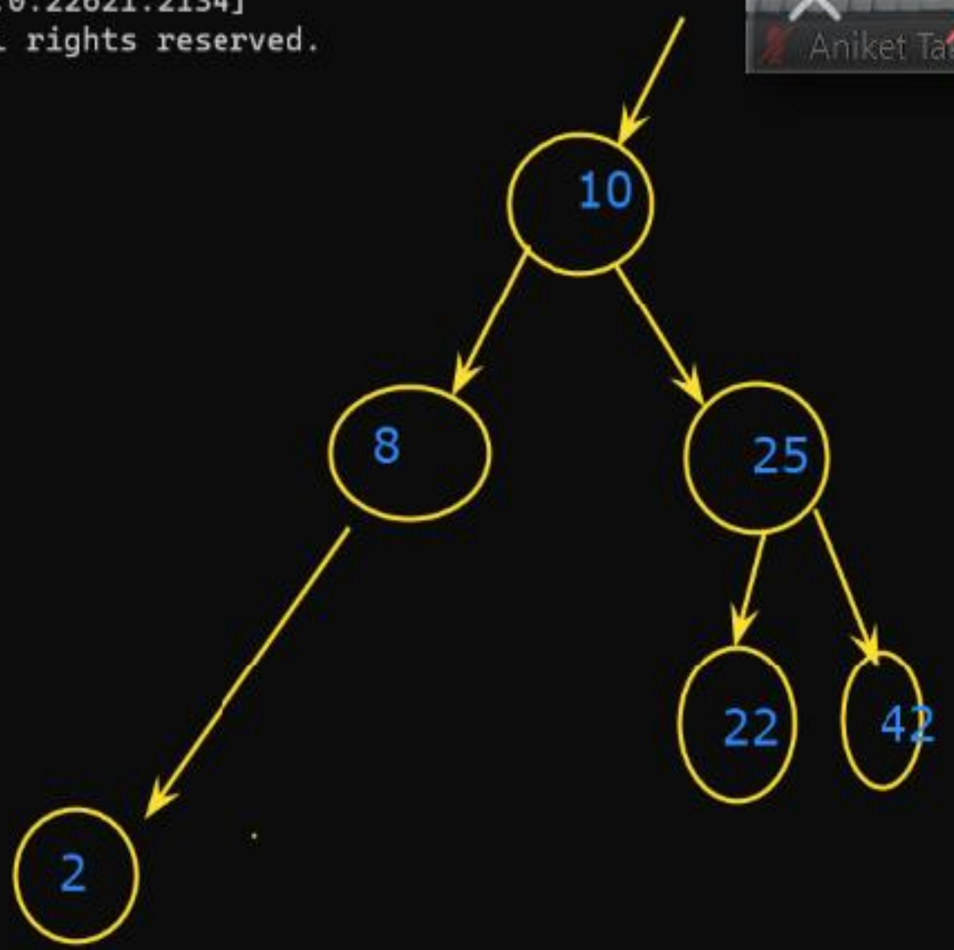
- The following operations are performed on a binary earch tree...
  - Search
  - Insertion
  - Deletion
  - Traversal

Insertion of a key in a BST

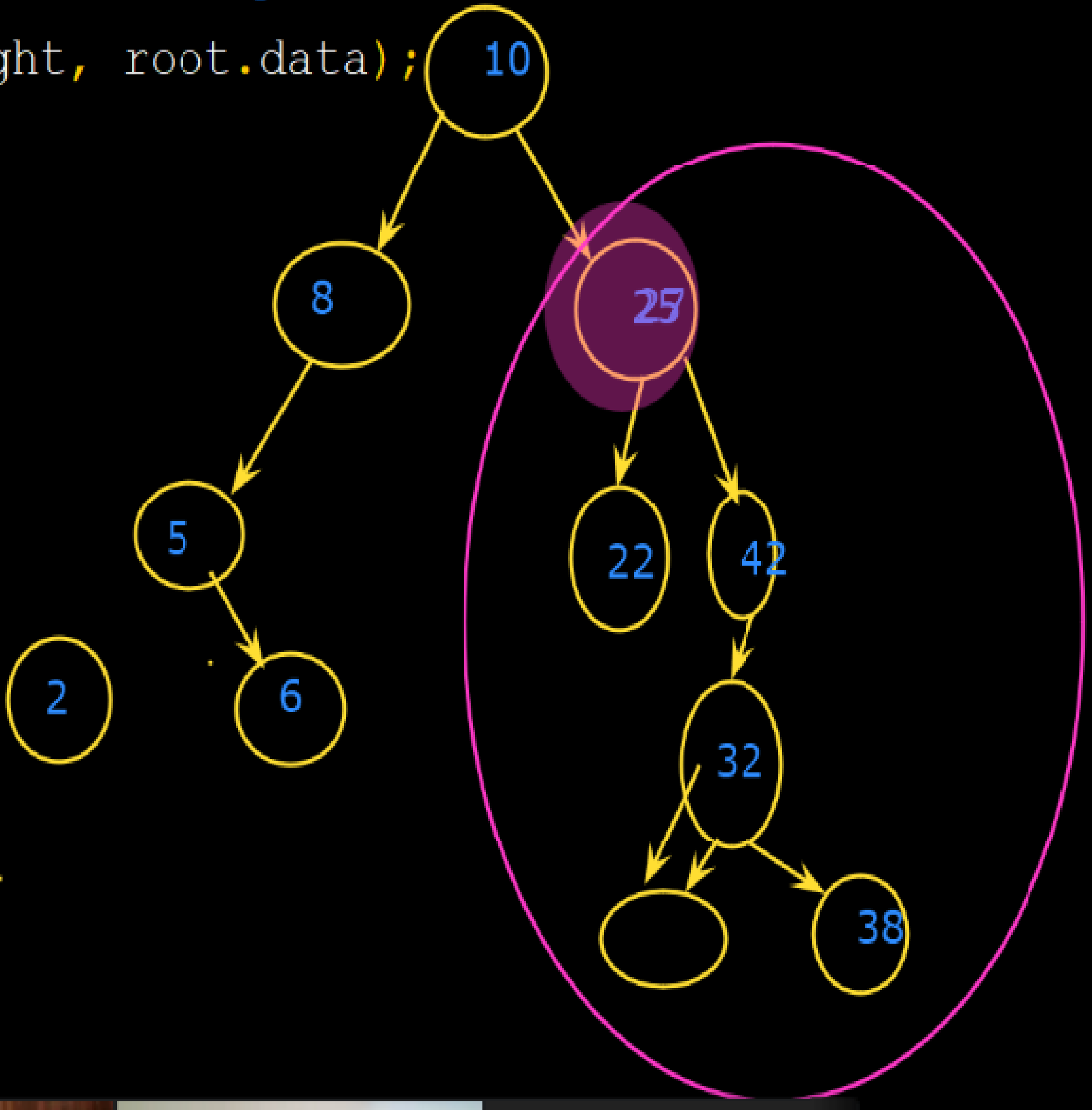
```
Algorithm:- InsertBST (info, left, right, root, key, LOC)
{
  key is the value to be inserted.
  1. call SearchBST ( info, left, right, root, key, LOC , PAR )    // Find the parent of the new node
  2. If ( LOC != NULL)
  2.1 Print " Node alredy exist"
  2.2 Exit
  3. create a node [ new1 =  ( struct node*) malloc ( sizeof( struct node) ) ]
  4. new1 -> info = key
  5. new1 -> left = NULL , new1 -> right = NULL
  6. If ( PAR = NULL ) Then
  6.1 root = new1
  6.2 exit
    elseif ( new1 -> info < PAR -> info)
  6.1 PAR -> left = new1
  6.2 exit
    else
  6.1 PAR -> right = new1
  6.2 exit
 }
```

```
System.out.println
System.out.println
t1.Postorder();

//case 1: Deletion
t1.delete(9);
System.out.println
System.out.println
t1.Inorder();

//case 2: Deletion
t1.delete(5);
System.out.println
System.out.println
t1.Inorder();

//case 3: Deletion
t1.delete(25);
System.out.println
System.out.println("Inorder");
t1.Inorder();
```

```
Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

D:\Test>javac BST.java

D:\Test>java BST
Inorder
10 20 25 30
Preorder
10 20 30 25
Postorder
25 30 20 10
D:\Test>javac BST.java

D:\Test>java BST
Inorder
2 5 8 9 10 22 25 42
Preorder
10 8 5 2 9 25 22 42
Postorder
2 5 9 8 22 42 25 10
Inorder
2 5 8 10 22 25 42
Inorder
2 8 10 22 25 42
Inorder
2 8 10 22 42
D:\Test>
```



```
        //case 3
        root.data = minvalue(root.right);
        root.right = deletedata(root.right, root.data);
    }
    return root;
}


int minvalue(Node root)
{
    int x =root.data;
    while(root.left !=null)
    {
        x = root.left.data;
        root =root.left;
    }
    return x;
}

void delete(int key)
```



Deleting Nodes from a Binary Search Tree

- Write an algorithm to locate the position of the node to deleted from a binary search tree.

- Delete operation in a binary search tree refers to the process of deleting the specified node from the tree.
- Before implementing a delete operation, you first need to locate the position of the node to be deleted and its parent.
- To locate the position of the node to be deleted and its parent, you need to implement a search operation.

Deleting Nodes from a Binary Search Tree (Contd.)

- Once the nodes are located, there can be three cases:
  - **Case I**: Node to be deleted is the leaf node
  - **Case II**: Node to be deleted has one child (left or right)
  - **Case III**: Node to be deleted has two children

Deletion of a key from a BST

```
Algorithm:- Delete1BST (info, left, right, root, LOC, PAR)
            // When leaf node has no child or only one child
{
    1.  if ( ( LOC -> left = NULL)  and  ( LOC -> right = NULL))
            1.1 Child = NULL
        elseIf ( LOC -> left != NULL)
            1.1 Child = LOC -> left
        else
            1.1 Child = LOC -> right
    2. if ( PAR != NULL)
            2.1   if ( LOC = PAR -> left )
                        2.1.1 PAR -> left = Child
            2.1   else
                        2.1.1 PAR -> right = Child
        else
            2.1 root = Child
 }
```

Deletion of a key from a BST

```
Algorithm:- Delete2BST (info, left, right, root, LOC, PAR)
            // When leaf node has both child
{
   1. ptr1 = LOC
   2. ptr2 = LOC -> right
   3. While ( ptr2 -> left != NULL )
            3.1 ptr1 = ptr2
            3.2 ptr2 = ptr2 -> left
   4. call Delete1BST (info, left, right, root, ptr2, ptr1)
   5. If ( PAR != NULL)
            5.1   If LOC = PAR -> left
                    5.1.1 PAR -> left = ptr2
            5.1 else
                          5.1.1 PAR -> right = ptr2
      else
            5.1 root = ptr2
   6. ptr2 -> left = LOC -> left
   7. ptr2 -> right = LOC -> right
}
```

Deletion of a key from a BST

Algorithm:- DeleteBST (info, left, right, root, key)

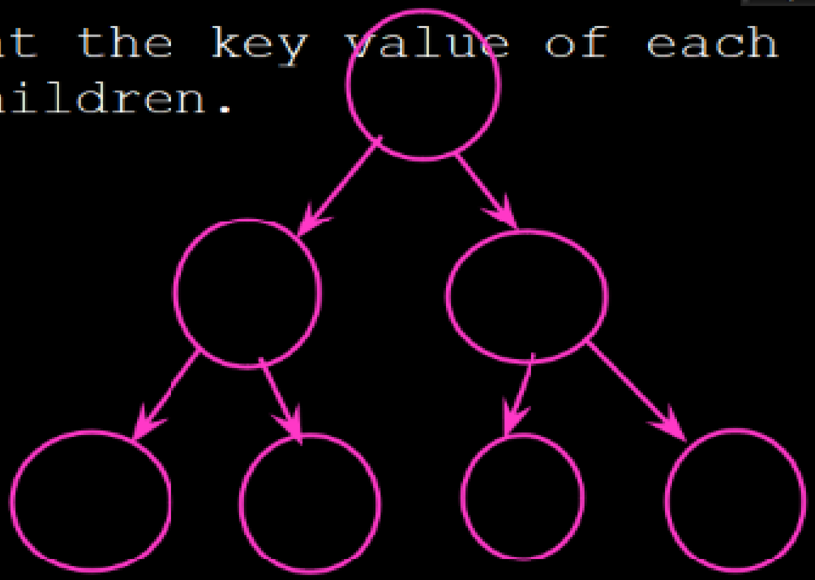
```
{
key is the value to be deleted.
    1. call SearchBST ( info, left, right, root, key, LOC, PAR )
            // To find the location LOC and parent  PAR of the
                                node to be deleted.
    2. If ( LOC = NULL ) Then
            2.1 Print " Node does not exist"
            2.2 exit
    3.  if ( ( LOC -> left != NULL)  and  ( LOC -> right != NULL))
                                    // when the node to be deleted has both child
            3.1 call Delete2BST (info, left, right, root, LOC, PAR)
        else
            3.1 call Delete1BST (info, left, right, root, LOC, PAR)
    }
```

```java
private boolean search(BSTNode r, int val)
    {
        boolean found = false;
        while ((r != null) && !found)
        {
            int rval = r.getData();
            if (val < rval)
                r = r.getLeft();
            else if (val > rval)
                r = r.getRight();
            else
            {
                found = true;
                break;
            }
            found = search(r, val);
        }
        return found;
    }
```

Heap

DEfinition:

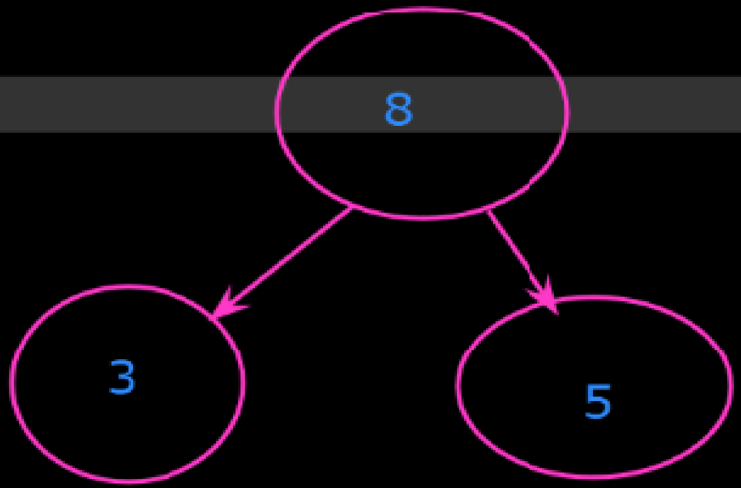
-a special form of complete binary tree that the key value of each no
smaller(larger) than the key value of it children.



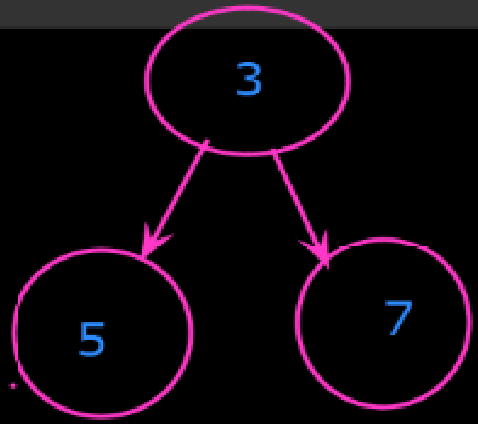
Types of heap:

1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value



Max Heap



Min Heap

Heap

Heap

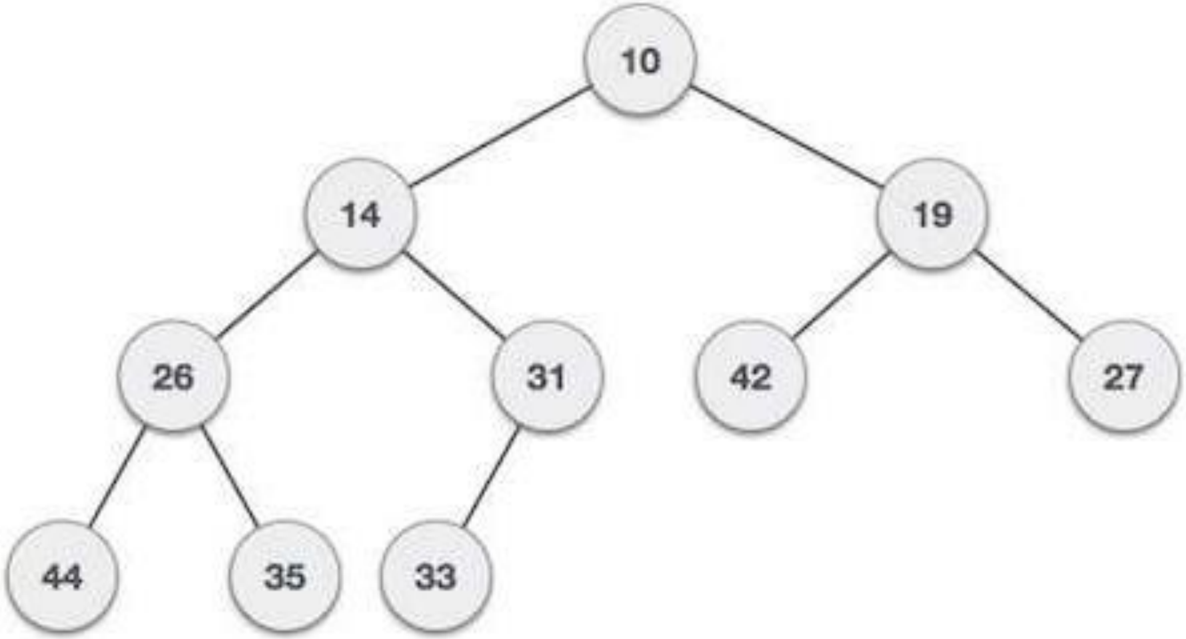
- **Definition in Data Structure**
  - **Heap:** A special form of complete binary tree that key value of each node is no smaller (larger) than the key value of its children (if any).

- **Max-Heap: root node has the largest key.**
  - A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children.
  - A *max heap* is a complete binary tree that is also a max tree.

- **Min-Heap: root node has the smallest key.**
  - A *min tree* is a tree in which the key value in each node is no larger than the key values in its children.
  - A *min heap* is a complete binary tree that is also a min tree.

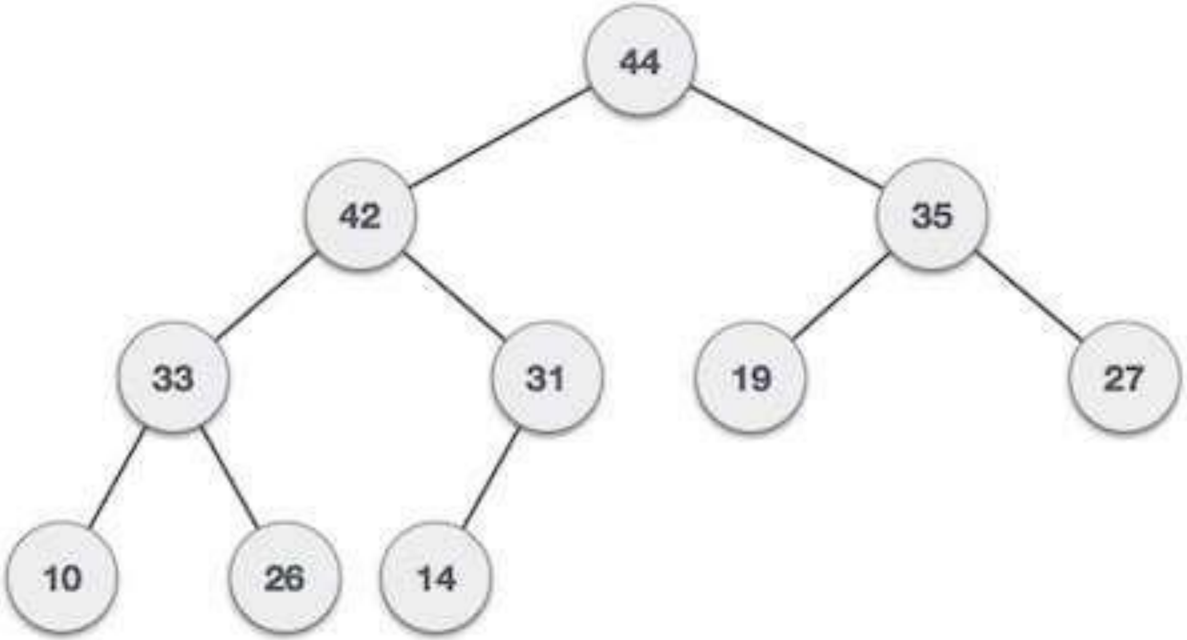
Heap

- Min-Heap
  - Where the value of the root node is less than or equal to either of its children
  - For input 35 33 42 10 14 19 27 44 26 31



Heap

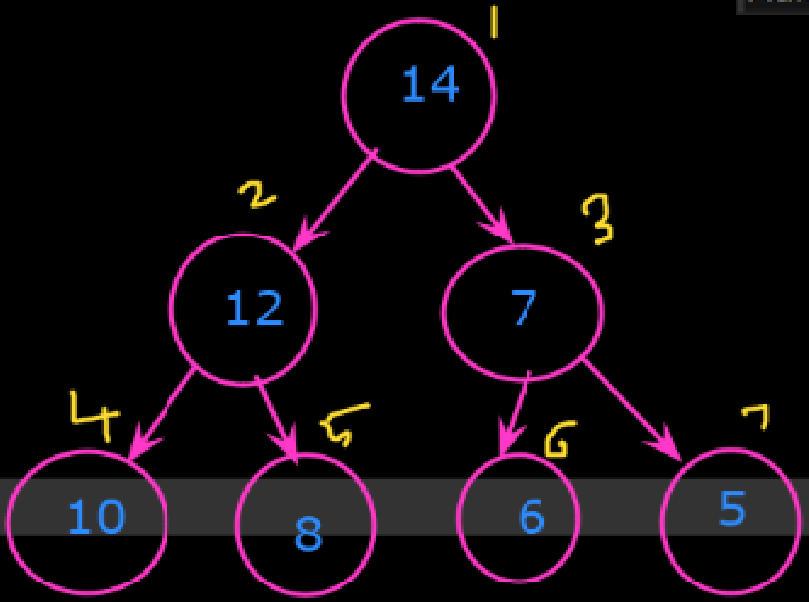
- Max-Heap −
  - where the value of root node is greater than or equal to either of its children.
  - For input 35 33 42 10 14 19 27 44 26 31



Types of heap:

1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value



Max heap

Parent = i/2

Lc = 2i

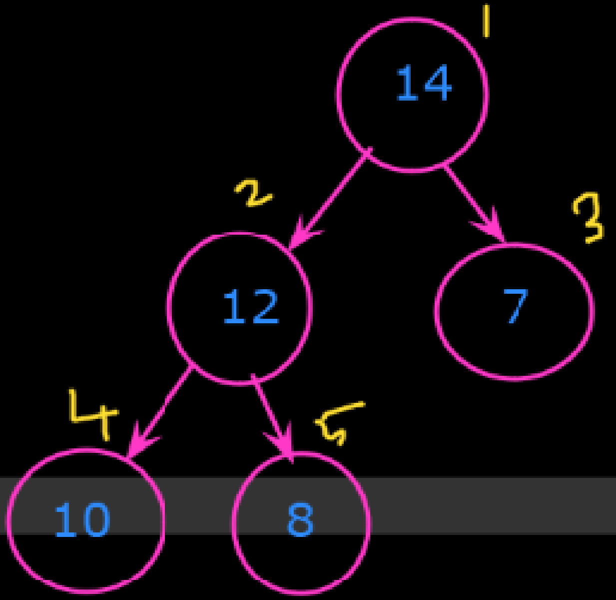
RC = 2i+1

14	12	7	10	8	6	5
1	2	3	4	5	6	7

Types of heap:

1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value



Max heap

Parent = i/2

Lc = 2i

RC = 2i+1

14	12	7	10	8

1 2 3 4 5 6 7

Types of heap:

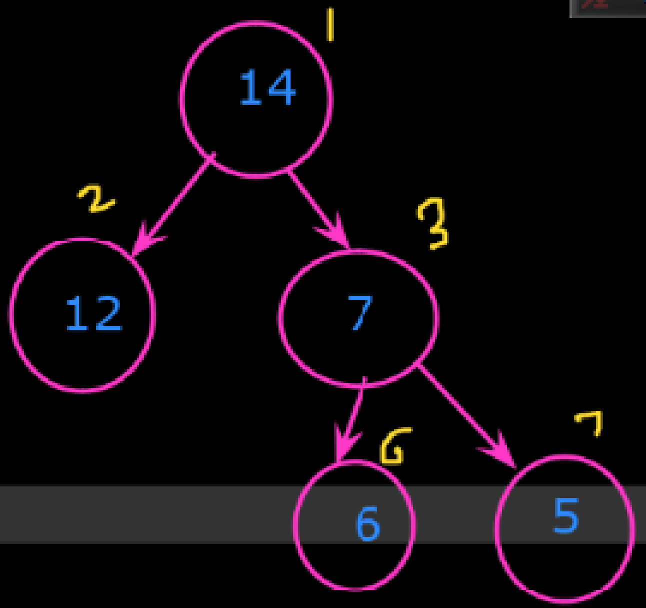
1. Max-Heap: root node has largest value

2. Min-Heap: root node has smallest value

Parent = i/2

Lc = 2i

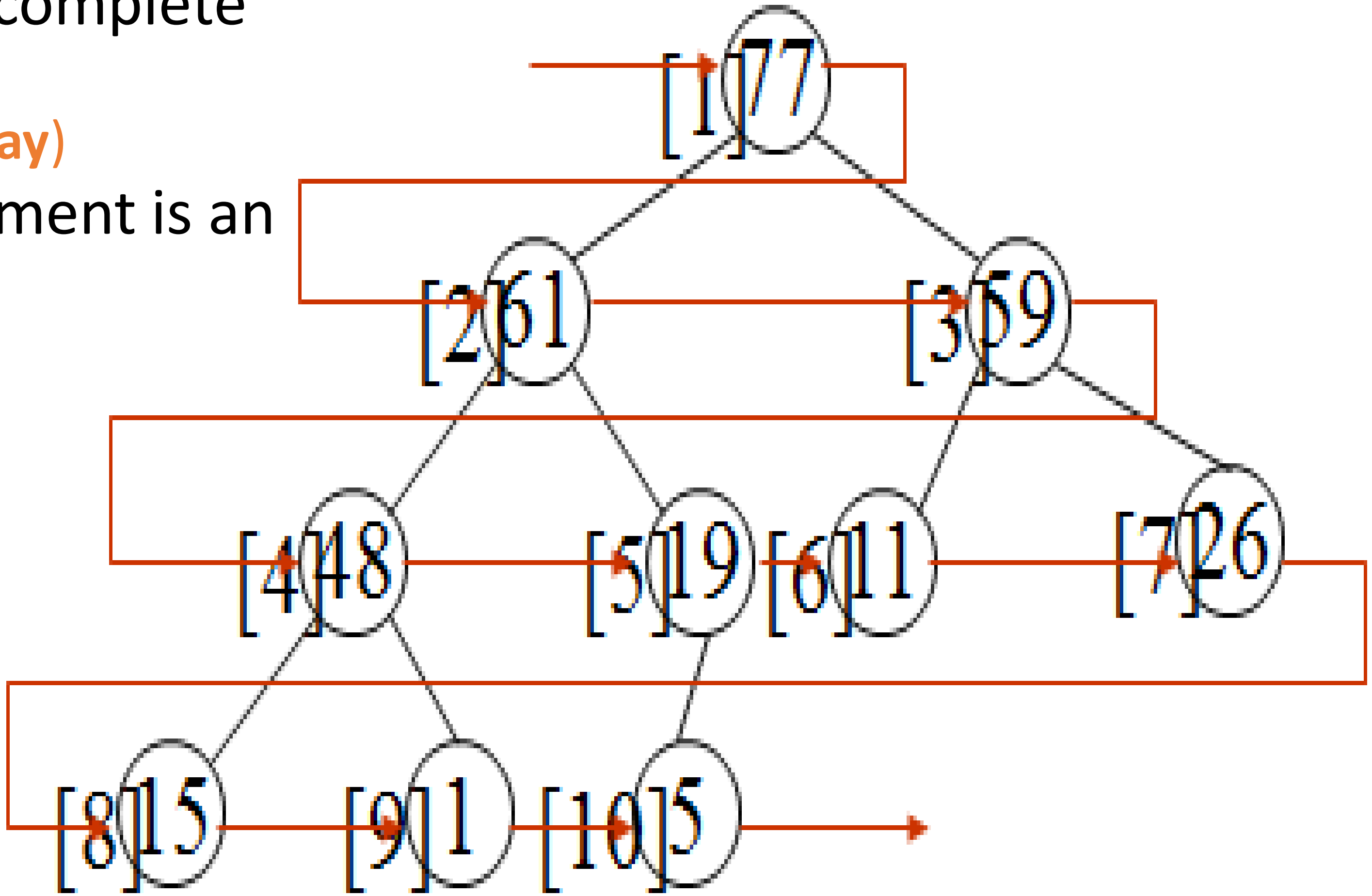
RC = 2i+1



Max heap

14	12	7	-	-	6	5
1	2	3	4	5	6	7

- **Note:**
  - Heap in data structure is a complete binary tree!
    - **(Nice representation in Array)**
  - Heap in C program environment is an array of memory.
-



– Stored using array in C

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5

Thanks