Linked List in Java :-

Linked List is part of collection framework present in java.util.package. It is linear data structure. where the elements are not stored in contigous locations & every object is separate object with the data part & address part. The element is known as Node.

1. LinkedList ll = new LinkedList();
2. LinkedList ll = new LinkedList(c);

Here c is the collection of elements. (list of elements)

Method of LinkedLists :- (E → Generic type)

add(int index, E element) : To add element to linkedList.
add(E element) : To append the element.
addAll(Collection<E> c) :- To append collection of elements
addAll(int index, collection<E>) :- To add collection of element at certain index.
addFirst(E e) :- Add elements at start index
addLast(E e) :- Append the element at last.
clone() :- returns the clone linkedList.

Ex :-

```
LinkedList ll = new LinkedList();
LinkedList si = new LinkedList();
si = (LinkedList) list.clone();
```

clear():- The method clear the Elements in LinkedList

Contains(Object o):- It return the boolean type which check the object o is contain in LinkedList

get(Index I):- returns the Element which of Index of I

indexOf(Object o):- returns the index of certain element

pop():- pops (or) retrieves the Element of 1st element

push(E e):- push the element to End of list.

remove():- Same as pop.

size():- return size of list (Inbuilt Class)

toArray():- Returns Array of proper sequence.

toString():- Returns String containing all the Elements

set(Index i, Element E):- Update the element that index of i.

Advantages of LinkedList:-

1. Dynamic Size
2. Efficient Insertion & deletion
3. Flexible iterations

Disadvantages:-

1. performance.
2. memory.

Double Linked List:-

We can Implement the feature of front & back navigation using double linked list.

LRU (least Recently used) & mru are constructed by using a doubly linked list.

In Double linked list the Insertion & deletion have Complexity of O(1) & O(n) for the node Indexing & Searching.

The double linked list consist of data, next & prev values.

```
public class DLL {
    class Node {
        public int data;
        public Node next;
        public Node prev;
    }
    Node head = null;
    Node tail = null;
    public void addNodeFront(int val) {
```

← Used without constructors

Code for Double linked list:-

```
public class DLL {
    private class Node {          // Implementation Node class with Constructor.
        int val;
        Node next;
        Node prev;
        public Node(int val){
            this.val = val;
        }
        public Node(int val, Node next, Node prev){
            this.val = val;
            this.next = next;
            this.prev = prev;
        }
    }
    public void InsertFirst(int val){
        Node node = new Node(val);
        node.next = head;
        node.prev = null;
        if (head != null){
            head.prev = node;
        }
        head = node;
    }
```

```
public void display() {
    Node node = head;
    Node last = null;
    while (node != null) {
        Sout (node.val + " -> ");
        node = node.next;
        last = node;
    }
    Sout ("END")
    Sout ("print in Reverse");
    while (last != null) {
        Sout (last.val + " -> ");
        last = last.prev;
    }
    Sout ("To start");
}
public Node Find (int value) {
    Node node = head;
    while (node != null) {
        if (node.value = value) {
            return node;
        }
        node = node.next;
    }
    return null;
}
public void Insert (int after, int val) {
    Node p = Find (after);
    if (p == null) {
        Sout ("does not Exist");
        return;
    }
    Node node = new Node (val);
    node.next = p.next;
    p.next = node;
    node.prev = p;
    if (node.next != null) {
        node.next.prev = node;
    }
}
```

Stack:- It is data structure which the LIFO (or FILO formal)
Last In First out (or) First In Last out.
It have two functions :- (1) push (2) pop
push is used to push elements into stack.
pop is used to remove element.
Both operations have same Complexity of O(1).
Stacks are created using classes in the framework.
we can create Dynamic Stack by changing the size of stack.

Code of Implementation of Stacks without using Framework :-

```
public class Customstack {
    protected int[] data;
    private static final int Default_size = 10;
    int ptr = -1;
    public Customstack() {          // Constructor
        this(Default_size);
    }
    public Customstack(int size) {
        this.data = new int[size];
    }
    public boolean push(int item) {     // push function
        if (isFull()) {
            System.out.println("Stack is full!");
            return false;
        }
        ptr++;
        data[ptr] = item;
        return true;
    }
    public int pop() throws StackException {     // pop function
        if (isEmpty()) {
            throw new StackException("Cannot pop from Empty stack");
        }
        return data[ptr--];
    }
```

```
public int peek() throw StackException {
                                    // peek Function
    if(isEmpty()){
                    Stack;
        throw new^Exception(" Cannot peek from an Empty stack
        }
        return data[ptr];
        }
public boolean isFull(){
    return ptr = data.length-1;
        }
public boolean isEmpty(){
    return ptr == -1;
        }
}
```

Using Framework :-

```
import java.util.stacks
public class Main{
  p s v m (string[] args){
    Stack<Integer> stack = new Stack<>();
                    ↓
                Generic class type.
        stack.push(10);   // adds 10 Element
        stack.peek();     // Show last Element in stack which is 10
        stack.pop();      // removes the 10 Element.
  } } stack.isEmpty();    // Return true Stack is Empty.
```

Stack Usages :-

* Implementation the undo feature.
* Build Complier
* Evaluate Expression
* Build Navigation.

Code for reversing the String using stacks:-

```
public class Main {
    public String reverse (String input) {
        Stack <Character> stack = new Stack<>();
        for (int i = 0; i< input.length(); i++)
            stack.push (input.charAt(i));

        String s = new String();
        while (! stack.empty()) {
            s.append (stack.pop());
        return s;
        }
```

Edge cases:-

```
if (input == null) {
    throw new IllegalArgumentException();
```

Balancing of Brackets:-

Test case:-

input:- ((<>))

output: false. // beg different closing tags.

For my code:- Isbalanced_bracket_using_stacks.

Note:-

asList():- Used to convert elements into List of Element.

Ex:- List <character> x = Arrays.asList('(', '(', 'v')

↳ [')', '(', 'v'].

Queue's:

It is data structure type of FIFO or LILO
First In first Out or Last In Last Out

Queue's are done by two Operation
1. enqueue:- Adding of Elements to the queue.
2. dequeue:- Removing of Elements to the queue.

Queue's are Implemented by using Interfaces
The Operations of the enqueue, dequeue, peek, isEmpty, isfull have Complexition of O(1).

Queue is Created by
Queue <Integer> x = new Queue <Integer>();

In Array Queue :- The null Elements are not allowed.
- The thread is not Safe.
- Have no Capacity restrictions.

To used Queue framework we need to import java.util.Queue.

Custom Code for Implementation of Queue :-

```
public class CQueue{
    private int [] data;
    private Static final int Default_Size=10;
    int end = 0;
    public CQueue(){
        this(Default_Size);
    }
    public CQueue(int size){
        this.data = new data[size];
    }
    public boolean enqueue(int item){
        if (isFull){
            throw new IllegalStateException();
        }
        data[end++] = item;
        return true;
    }
```

```
public int dequeue() throws Exception {
    if (isEmpty()) {
        throw new Exception("Queue is Empty");
    }
    int removed = data[0];
    for (int i = 1; i < end; i++) {
        data[i-1] = data[i];
    }
    end--;
    return removed;
}
public boolean isFull(){
    return end == data.length;
}
public boolean isEmpty() {
    return end == 0;
}
}
```

Circular Queue:-

In linear Queue, insertion is done from the rear end & deletion is done from front end. In circular queue the insertion & deletion can take place from any end. Hence, Circular queue requires less memory than the linear queue.

Code for Circular Queue:-

```
class Circular_Queue {
    protected int[] data;
    private static final int Default_size=10;
    protected int end = 0;
    protected int front = 0;
    private int size=0;
    public Circular_Queue() {
        this(Default_size);
    }
    public Circular_Queue(int size){
        this.data = new data int[size];
    }
    public boolean isFull() {
        return size == data.length;
    }
    public boolean isEmpty() {
        return size == 0;
    }
```

```
public boolean insert(int item){
    if(isFull()){
        return false;
    }
    data[end++] = item;
    end = end % data.length;
    size++;
    return true;
}
public boolean remove() throws Exception {
    if(isEmpty()){
        throw new Exception("Queue is Empty");
    }
    int removed = data[front];
    front = front % data.length;
    size--;
    return removed;
}
}
```

HashMap :-

The HashMap Consist of the key-value pair which you can index them the values using keys.

* Hashmaps. Collection of key & value is Hashtable.
* HashMap is implemented using interfaces of the Collection framework.
* Hash tables used for Spell checkers, dictionaries, Compilers & Code Editor
* In Hashmaps the storing & retriving of value are done.
* Duplicate key are not Allowed.
* All the functions (insert, lookup, delete) have the Complexity of O(1).
* we can stored null values in both key & value.
* we can used different type of generics for both key & value pair.

```
import java.util.HashMap;
HashMap<Integer, String> x = new HashMap<Integer, String>();
```

put method is used to add the key-value pair.
get method is used to add value of value using key.
remove method is used to remove pair using key as parameter.
clear ⟶ remove all Elements
Size ⟶ lenght of Hashmap.
O(1) → containsKey ⟶ returns boolean type whether key is present or not
O(n) → ContainsValue → Returns boolean type whether value is present or not.
entrySet → return key=value pair.

Set is Collection of Element without duplicate numbers.

↳ Set<Integer> Set = new HashSet<>();

add ⟶ used to add Element
remove ⟶ to remove
clear ⟶ clear all Elements.

By using Hash Function the biggest values are stored in Hash table.

While Hashing we may get collisions which a key have. to store more than 2 values

To overcome the solution the chaining method & open addressing methods are used.

① Chaining method:- which stores two values at a key by chaining (or) linking each other.

② Open Addressing:- when the value is stored at a key, another value which try to store at same key then, we need search for empty slot which is called probing.

1. Linear probing:- If the current slot is full (or a value is stored at certain key then it will search for next slot.

$$hash(key) + i$$

2. Quadratic probing:- $hash(key) + i^2$

3. Double probing:-

Hash1 → Hash1(key) = key % table size.

Hash2 → Hash2(key) = prime − (key % prime)

prime = nearest lessen prime to table size.

index = (Hash1 + i * hash2) % Size.

Trees:

It is non-linear data structure.
which have Root node as the top Element & leaf as the bottom Element
It is used to represent hierarchical data & indexing in the Database. It is used in Autocompletion & Compilers & Compression types (JPEG etc)

Binary Tree:-

The node which have 2 Branches is Binary tree

Binary Search Tree:-

where, The BST is left branch < Root node < Right branch

lookup → O(logn) Delete → O(logn)
insert → O(logn)

Custom Code for building Trees:-

```
public class Tree {
    private class Node {
        private int value;
        private Node leftchild;
        private Node Rightchild;
        public Node (int value) {
            this.value = value;
        }
    }
    private Node root;
    public void insert (int value) {
        var node = New node(value);
        if (root = null) {
            root = node;
            return;
        }
        var current = root;
        while (true) {
            if (value < current.value) {
                if (current.leftchild = null) {
                    current.leftchild = node;
                    break;
                }
```

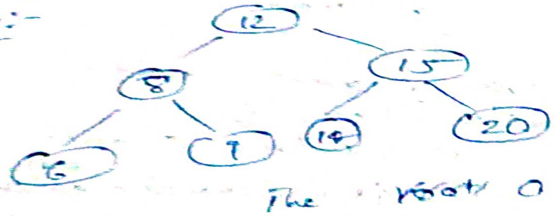
```
        Current = Current.leftChild;
    }
    else {
        if (Current.rightChild == null) {
            Current.rightChild = newNode;
            break;
        }
        Current = Current.rightChild;
    }
}
}
public boolean Find(int value) {
    var Current = root;
    while (Current != null) {
        if (value < Current.value) {
            Current = Current.leftChild;
        }
        else if (value > Current.value) {
            Current = Current.rightChild;
        } else
            return true;
    }
    return false;
}
}
```

Traversing Trees:-

1. Breadth First :- The roots are done first

Level order :-

Eg:-



The order of Traversing Tree is 12, 8, 15, 6, 1, 14, 20

2. Depth First :-

There are 3 ways :-

1. pre-order :- Root left right
2. post order left right Root
3. In order left Root right

These ways are done by using Recursion.

Code for all three ways of Depth First :-

```
public void tPreorder (Node node) {
    if (root = null) {
        return;
    }
    Sout (root.value);          // Using Recursion
    tPreorder (root.leftChild);
    tPostorder (root.rightchild);
}

public void tPostorder (Node root) {
    if (root = null) {
        return;
    }
    tPostorder (root.leftchild);
    tPostorder (root.rightchild);
    tPostorder (root.value);
    Sout
}

public void tInorder (Node root) {
    if (root = null) {
        return;
    }
    tInorder (root.leftchild);
    Sout (root.value);
    tInorder (root.rightchild);
}
```

Depth will increase while going root to leaves
Height will increase while going leaves to root.

Code for finding height :-

```
public int height (Node root) {
if (root.leftchild == null && root.rightchild == null){
    return 0;
    }
return 1 + Math.max (height (root.leftchild),
                     height (root.rightchild));
}
```

End point :-

```
if (root == null) {
    return -1;
}
```

Code for minimum value of Tree :-

```
public int min (Node root) {
    left = min (root.leftchild);
    right = min (root.rightchild);
return Math.min (Math.min (left, right), root.value);
}
```

O(n).

Edge case :-

```
if (root.leftchild == null && root.rightchild == null){
    return root.value;
}
```

Code for minimum value in Binary Search Tree :-

```
public int min ( ) {
    if (root == null)
        throw new Exception ( );
var c = root;  var last = c;
while (c != null) {
    last = c;
    c = c.leftchild;
}return last;
}
```

// O(log n)

Code for checking whether two trees are Equal (or) not :-

```
public boolean Equals(Node First, Node Second){
    if (First.value == null && Second == null){
        return true;
    }
    if (First != null && Second != null){
    return First.value == Second.value
        && Equals(First.leftchild, Second.leftchild)
        && Equals(First.rightchild, Second.rightchild);

    return false;
    }
}
```

Code for checking whether given tree is Binary Search Tree (or) not :-

```
public boolean isBinary(Node root, int min, int max){
    if (root == null){
        return true;
    }
    if (root.value < min || root.value > max)
        return false;
    return isBinary(root.leftchild, min, root.value - 1)
        && isBinary(root.rightchild, root.value + 1, max);
        // Calling function ↓.
    isBinary(root, Integer.MIN_value, Integer.max_value);
```

// min → min value of Integer $-\infty$
// max → max value of Integer $+\infty$

Code for node at k distances from root :-

```
public void knode(Node root, int distance){
    if (root == null)
        return;
    if (distance == 0){
        Sout(root.value);      // list.add(root.value);
        return;                   for next code
    }
    knode(root.leftchild, distance-1);
    knode(root.rightchild, distance-1);
}
```

// returns ↑ two nodes which of BST (left & right) values from

Code for Level Order Traversal (Breadth first):

get nodes from t - nodes in a array.

```
public void knodearray(Node root, int distance, ArrayList
public ArrayList<Integer> knodearray(int distance){
    ArrayList<Integer> list = new ArrayList<Integer>;
    knode(root, distance, list);
    return list;
}
```

for getting the knodes in arraylist.

```
public void OrderLevel(){
    for(int i=0; i<=height(); i++){
    ArrayList<Integer> list = knodearray(i);
    for(int value: list)
        Sout(value);
    }
}
}
```

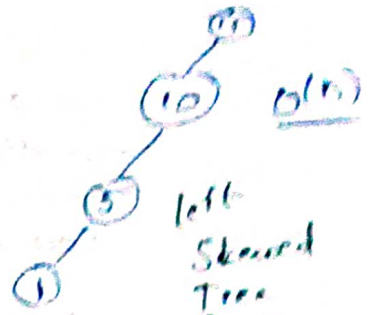
AVL Trees:

If the Binary Search Tree is not structured properly then it leads to O(n) from O(log n)

For Balanced Tree formula:-

-1 => height (left) - height (right) <= 1

height (left) - height (right) >= 1



Right - Skewed Tree:-

where the branches are done right side of roots

Left - skewed Tree:

where the branches are done left side of roots.

AVL (Adelson-velskey and Landis) Trees are self balancing trees

They balance the rotating the branches. There are 4 types of rotations:

1. Left (LL) 2. Right (RR) 3. left Right (LR)
4. Right left (RL)

After the rotation the height of trees will change.

Insert Node:-

we have done Inserting in tree using loop. here we will do using Recursion.

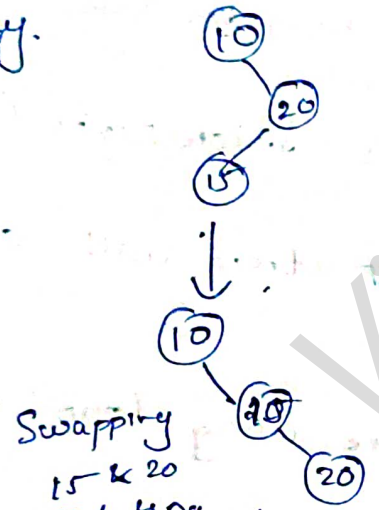
Code:-

```
private class AVLTree {
    private class AVTNode {
        private int val;
        private AVTNode leftchild;
        private AVTNode rightchild;
        public AVTNode (int value) {
            this.value = value;
        }
    }
    private AVTNode root;
    public void insert (int val) {
        root = insert (root, value);
    }
```

```
private AVTNode insert (AVTNode root, int value) {
    if (root == null)
        return new AVTNode (value);
    if (root.value > value)
root.leftchild = insert (root.leftchild, value);
    else
root.rightchild = insert (root.rightchild, value);
    return root;
}
}
```

AVL Tree are used to reduce Efficient O(n) to O(logn) By reducing the height of tree using Rotations

(1) way.



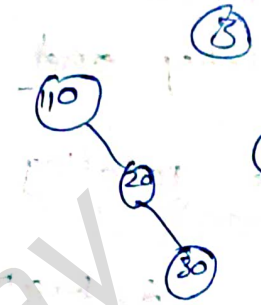
Swapping 15 & 20

(1) Rotating Rightside.

(2) Rotated 10 to left.

RL - Rotation

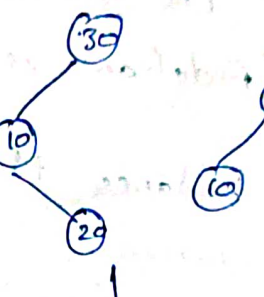
(2)



Rotating the 10 towards leftside.

LL - Rotation

(3)

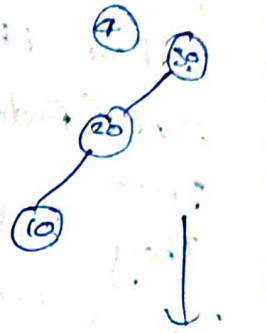


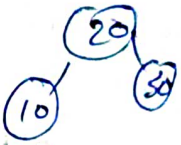
Rotating 20 towards left side

Rotating towards 30 to rightside.

LR - Rotation

(4)



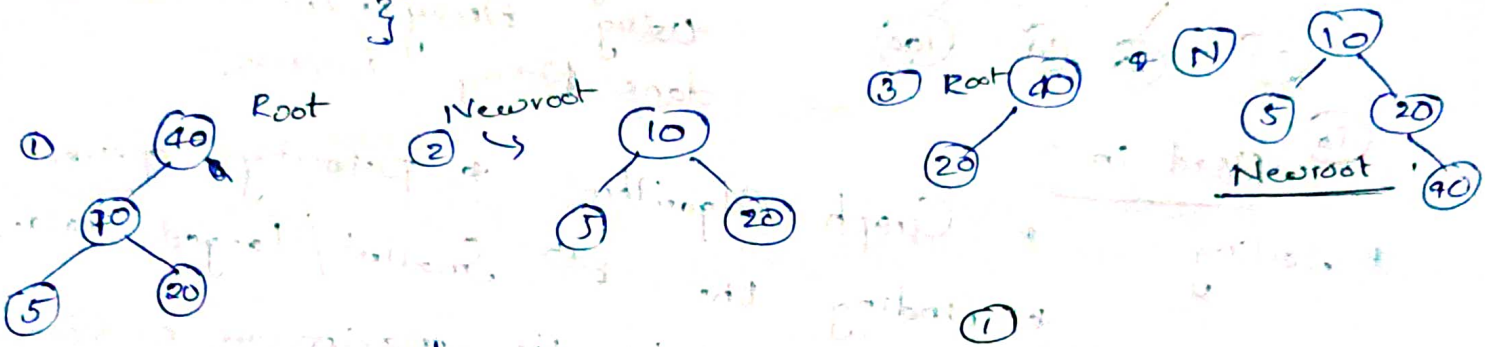


Rotate 30 towards Right

RR - Rotation

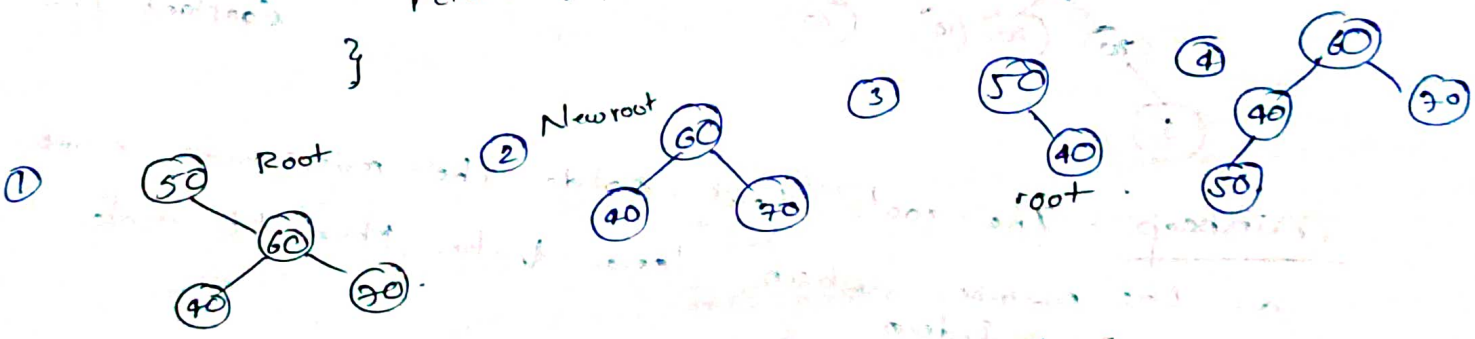
RotateRight :-

```
public AVLNode RotateRight (AVLNode root) {   // return type of AVLnode.
    AVLNode Newroot = root.leftchild;
    root.leftchild = newRoot.rightchild;
    newRoot.rightchild = root;
    Setheight (root);
    Setheight (Newroot);   // After rotation the height will change
    return Newroot;
}
```



RotateLeft :-

```
public AVLNode RotateLeft (AVLNode root) {
    AVLNode Newroot = root.rightchild;
    root.rightchild = Newroot.leftchild;
    Newroot.leftchild = root;
    Setheight (root);
    Setheight (Newroot);
    return Newroot;
}
```



isRightheavy :-

```
public boolean isRightheavy (AVLnode root) {
    if (height(left) - height(right) < -1)
        return true;
    return false;
}
```

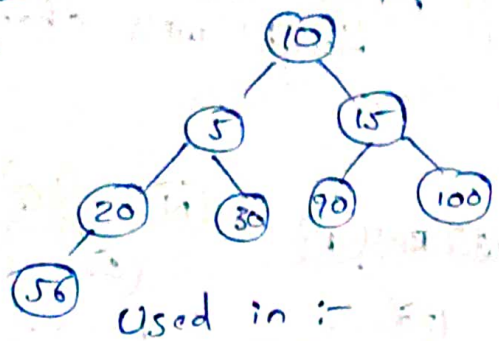
isLeftheavy :-

```
public boolean isLeftheavy (AVLnode root) {
    if (height(left) - height(right) > 1)
        return true;
    else false;
}
```

Heaps

It is type of data structure of form of Binary Tree which the elements are stride filled from left.

Ex:- 10, 5, 15, 20, 30, 90, 100, 56



① Filling the tree completely.

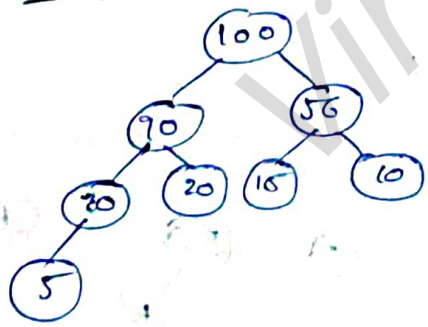
Heaps are Implemented using Arrays. The Nodes are done Using Indexes.

Used in :-

* Sorting
* Graph Algorithm
* priority queues
* finding the k^{th} Smallest / largest value.

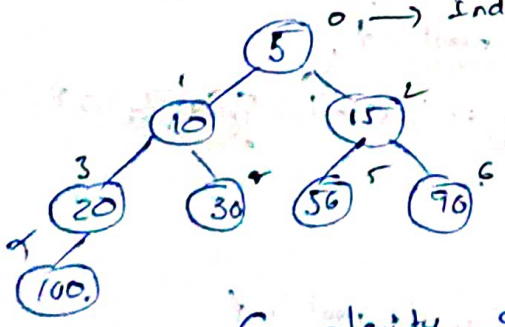
Maxheap:- The roots hold the maximum number (or) the number which is Greater than its nodes

Example for above problem:-



* From the Maxheap, the Complexity of Getting of Maximum no. is O(1) Constant Time.

Minheap:- The roots that holds the minimum number (or) the number which lesser Value than the nodes



The Complexity of removing & adding of Element is O(log n).

Indexes:-

Indexes of leftchild:- parent (Indexes) * 2 + 1

Indexes of Rightchild:- parent (Indexes) * 2 + 2.

Index of parent of node:-

(Node - 1) / 2

Example:- 1. Indexes of Element 100 using Element 20 Index.

Index (20) = 3.

Index (100) = 3 * 2 + 1 = 7.

2. Index of Element 15 using Element 90 Index

Index (90) = 6.

Index (15) = 6/2 + 1 = 3 - 1 = 2 //

Code for heaps:-

```
public class Heap {
    public int[] array = new int[10];
    public int size;
    public void insert(int val){
        if(isFull()){
            throw new IllegalStateException();
        }
        item[size++] = val;
        bubbleUp();
    }
public boolean isFull(){
    return size == item.length;
    }
public void bubbleUp(){
    var Index = size - 1; // Above size is done incremented (++size)
    while( Index > 0 && item[Index] > item[parent(Index)]){
        Swap(Index, parent);
        index = parent(Index);
    }
}
                                    // for value of parent
public int parent(int Index){
    return Index - 1/2;
    }
Swap(int Index, int Index2){    // Swapping of Indexes
    int temp = item[Index]; // item is array.
item[Index] = item[Index2];
item[Second] = temp;
}
```

```
public int remove() { // removes the max value.
                      (root val) &
                      poistion the next max
                      in root.
  if (isEmpty()) {
    throw new IllegalStateException();
  }
  int root = a[0];   // a is array (= mistake)
  a[0] = a[size];    // previous we have Incremental Size
                        by 1 than required
  bubbledown();
  return root;
}
public boolean isEmpty() {
    return size == 0;
  }
public int rightchild(int index) // for rightchild index
    return index * 2 + 2;
public int leftchild(int index) // for leftchild index
    return index * 2 + 1;
public int leftchildval(int index) {
    return item[leftchild(index)];
public int Rightchildval(int index)
    return item[rightchild(index)];
public boolean hasleftchild(int index) {
    return leftchild <= size;
public boolean hasrightchild(int index) {
    return rightchild <= size;
public boolean isvalidparent(int index) {
    if (! hasleftchild) {  // It means root has no child
        return true;          given in boolean value
      }
    boolean isvalid = item[index] >= rightchildval(index);
    isValid && item >= leftchildval(index);
    //isvalid = isvalid && item >= Rightchildval(index);
    return isValid;
}
```

```
private int largerchildIndex(int index){
    if(!hasleftchild)
        return index;
    if(!hasRightchild)
        return leftchildval(Index);
    return (leftchildval(index) > Rightchildval(index) ?
                leftchild(index) : rightchild(index));
    }  // return which child has bigger value of Index
privak void bubbledown(){
    var Index = 0;  // int is var
    while(index<=sizeA && isvalidparent(index)){
     // here not Equal is for (which we have to check up to root)
     int large = largerchildIndex(index);
       if(Swap(Index, largerchildIndex);
          index = large;
       }
    }
public int max(){
    if(isempty()).
        throw new IllegalstateException();
    return items[0];
    }
}
```

Heapify:- Rearranging the array into Heap property.

```
public class maxheap(int[]array){
    int lastparentIndex = array.length/2;
        // Gives the last parent of Index
                                for(int i= lastparentIndex;
                                    i>=0; i--){
        heapify(array, i);
    }; // End of for loop
}
```

The method which Heapify the array into maxheap. which max no. of array holds the root.

```
public static void heapify (int[] array, int index)
    int largerIndex = Index;
    int leftIndex = index * 2 + 1;
    if ( leftIndex < array.length && array[leftIndex]
                                        > array[largerIndex]
        largerIndex = leftIndex;
    int RightIndex = index * 2 + 2;
if ( RightIndex < array.lenght && array[leftIndex] > array[largerIndex]
        largerIndex = RightIndex;
    if ( Index == largerIndex)
        return;
        Swap (Index, largerIndex);
            array,
        heapify (array, largerIndex);
    }
public static void main (string args[]) {
    int x = array.lenght/2 - 1;
    // we Consider only parents without leaves
        child (which is efficient)
    for (int i = x; i >= 0; i--){
        heapify (array, i);
    }
}
public static int gthLargestNumber (int[] array, int k){
    if (k < 1 || k > array.length)
        throw new IllegalStateException();
    Heap heap = new Heap();
    for (int number : array)
        heap.insert (number);
    for (int i = 0; i < k-1; i++)
        heap.remove();
    return heap.max();
}
}
```