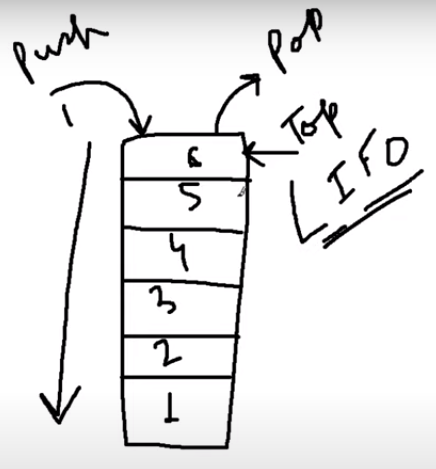
**Stack:**

* It is a child class of Vector.
* It is specially designed class for Last In First Out(LIFO).



**Stack Constructor:**

Stack s = new Stack();

Using the object we can access the methods of Stack.

**Methods in Stack:**

1. **Object push(Object obj);**

For inserting an object to the stack.

1. **Object pop();**

To remove and return top of the stack.

1. **Object peek();**

To return the top of the stack without removal of object.

1. **int search(Object obj);**

If the specified object is available it returns its offset from top of the stack.

If the object is not available then it returns -1.

Program:

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Stack<String> s = **new** Stack();

s.push("A");

s.push("B");

s.push("C");

System.***out***.println(s);

System.***out***.println(s.search("A"));

System.***out***.println(s.search("Z"));

s.pop();

System.***out***.println(s);

System.***out***.println(s.peek());

System.***out***.println(s);

}

}

Output:

[A, B, C]

3

-1

[A, B]

B

[A, B]

**Implement Stack using Array:**

Program – We are going to implement here

1. push
2. pop
3. isEmpty
4. isFull
5. peek

// Implement Stack using Array (fixed in size)

**public** **class** StackImplementation {

**int** size;

**int** arr[];

**int** top;

StackImplementation(**int** size){

**this**.size = size;

**this**.arr = **new** **int**[size];

**this**.top = -1;

}

**public** **boolean** isEmpty() {

**return**(top==-1);

}

**public** **boolean** isFull() {

**return**(size-1==top);

}

**public** **int** peek() {

**if**(!**this**.isEmpty()) {

**return** arr[top];

}

**else** {

System.***out***.println("Stack is empty");

**return** -1;

}

}

**public** **void** push(**int** element) {

**if**(!isFull()) {

top++;

arr[top] = element;

System.***out***.println("pushed element: " + element);

}

**else** {

System.***out***.println("Stack is full now");

}

}

**public** **int** pop() {

**if**(!isEmpty()) {

**int** returnedTop = top;

top--;

System.***out***.println("poped element: " + arr[returnedTop]);

**return** arr[returnedTop];

}

**else** {

System.***out***.println("Stack is empty");

**return** -1;

}

}

**public** **static** **void** main(String[] args) {

StackImplementation stackImp = **new** StackImplementation(10);

stackImp.pop();

System.***out***.println("---------------------");

stackImp.push(100);

stackImp.push(200);

stackImp.push(300);

stackImp.push(400);

System.***out***.println("-------------------------");

System.***out***.println(stackImp.peek());

stackImp.pop();

stackImp.pop();

stackImp.pop();

System.***out***.println(stackImp.isFull());

System.***out***.println(stackImp.isEmpty());

}

}

Output:

Stack is empty

---------------------

pushed element: 100

pushed element: 200

pushed element: 300

pushed element: 400

-------------------------

400

poped element: 400

poped element: 300

poped element: 200

false

false

**Implement Stack using LinkedList:**



**public** **class** StackUsingLinkedList {

Node head;

**class** Node{

**int** value;

Node next;

}

StackUsingLinkedList(){

head = **null**;

}

//push : add value to the beginning of the List

**public** **void** push(**int** value) {

Node extraHead = head;

head = **new** Node();

head.value = value;

head.next = extraHead;

}

**public** **int** pop() {

**if**(head==**null**) {

System.***out***.println("Stack is empty");

}

**int** value = head.value;

head = head.next;

**return** value;

}

**public** **static** **void** printStack(Node head) {

Node temp = head;

**while**(temp!=**null**) {

System.***out***.println(temp.value + " ");

temp = temp.next;

}

}

**public** **static** **void** main(String[] args) {

StackUsingLinkedList lls = **new** StackUsingLinkedList();

// System.out.println(lls.pop());

lls.push(10);

lls.push(20);

lls.push(30);

lls.push(40);

lls.push(50);

System.***out***.println("element poped out from stack: " + lls.pop());

System.***out***.println("element poped out from stack: " + lls.pop());

lls.push(55);

System.***out***.println("element poped out from stack: " + lls.pop());

*printStack*(lls.head);

}

}

Output:

element poped out from stack: 50

element poped out from stack: 40

element poped out from stack: 55

30

20

10

**Queue:**

* It is specially designed for First In First Out(FIFO).
* The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
* The Java Queue supports all methods of Collection interface including insertion, deletion etc.
* Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/" \t "_blank) in Java.

**Methods in Queue:**

1. add()- This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation.
2. peek()- This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
3. element()- This method is similar to peek(). It throws NoSuchElementException when the queue is empty.
4. remove()- This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty.
5. poll()- This method removes and returns the head of the queue. It returns null if the queue is empty.
6. size()- This method return the no. of elements in the queue.

Program:

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Queue<Integer> q = **new** PriorityQueue<>();

// Adds elements {0, 1, 2, 3, 4} to queue

**for** (**int** i=0; i<5; i++)

q.add(i);

// Display contents of the queue.

System.***out***.println("Elements of queue-"+q);

// To remove the head of queue.

**int** removedele = q.remove();

System.***out***.println("removed element-" + removedele);

System.***out***.println(q);

// To view the head of queue

**int** head = q.peek();

System.***out***.println("head of queue-" + head);

// Rest all methods of collection interface,

// Like size and contains can be used with this

// implementation.

**int** size = q.size();

System.***out***.println("Size of queue-" + size);

}

}

Output –

Elements of queue-[0, 1, 2, 3, 4]

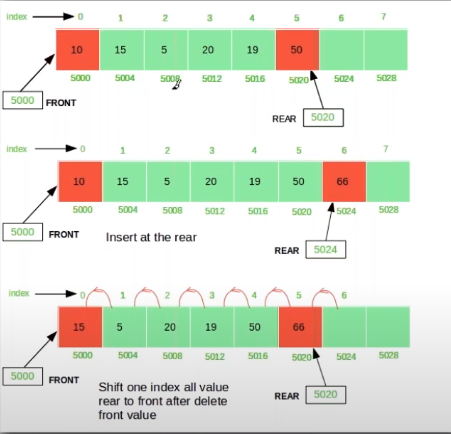
removed element-0

[1, 3, 2, 4]

head of queue-1

Size of queue-4

**Implement Queue using Array:**



**public** **class** QueueUsingArray {

**int** capacity;

**int** queueArr[];

**int** front;

**int** rear;

**int** currentSize;

**public** QueueUsingArray(**int** sizeOfQueue) {

**this**.capacity = sizeOfQueue;

front = 0;

rear = -1;

queueArr = **new** **int**[**this**.capacity];

}

**public** **boolean** isFull() {

**if**(currentSize == capacity) {

**return** **true**;

}

**return** **false**;

}

**public** **boolean** isEmpty() {

**if**(currentSize==0) {

**return** **true**;

}

**return** **false**;

}

/\*\*

\* this method is used to add element in the queue

\* **@param** data

\*/

**public** **void** enqueue(**int** data) {

**if**(isFull()) {

System.***out***.println("queue is full, can not insert the data");

}

**else** {

rear++;

**if**(rear == capacity) {

rear=0;

}

queueArr[rear] = data;

currentSize++;

System.***out***.println(data + "added to the queue");

}

}

/\*

\* this method is used to remove the element from the front of the Queue

\*/

**public** **void** dequeue() {

**if**(isEmpty()) {

System.***out***.println("Queue is empty, can not delete element");

}

**else** {

front++;

**if**(front == capacity) {

System.***out***.println(queueArr[front-1] + "removed from the queue");

front=0;

}

**else** {

System.***out***.println(queueArr[front-1] + "removed from the queue");

}

currentSize--;

}

}

**public** **static** **void** main(String[] args) {

QueueUsingArray obj = **new** QueueUsingArray(5);

obj.enqueue(10);

obj.enqueue(20);

obj.enqueue(430);

obj.enqueue(205);

obj.enqueue(125);

obj.dequeue();

obj.dequeue();

obj.dequeue();

obj.dequeue();

obj.dequeue();

}

}

Output –

10added to the queue

20added to the queue

430added to the queue

205added to the queue

125added to the queue

10removed from the queue

20removed from the queue

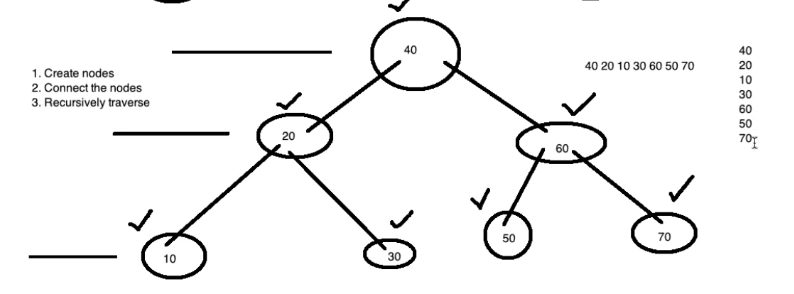
430removed from the queue

205removed from the queue

125removed from the queue

**Binary Tree:**

**PreOrder traversal**



Program:

**package** datastructures;

**public** **class** BTPreOrder {

//used to create the root node

**public** **static** **class** TreeNode{

**int** data;

TreeNode left;

TreeNode right;

TreeNode(**int** data){

**this**.data=data;

}

}

**public** **static** TreeNode createBinaryTree() {

TreeNode rootNode = **new** TreeNode(40);

TreeNode node20 = **new** TreeNode(20);

TreeNode node10 = **new** TreeNode(10);

TreeNode node30 = **new** TreeNode(30);

TreeNode node60 = **new** TreeNode(60);

TreeNode node50 = **new** TreeNode(50);

TreeNode node70 = **new** TreeNode(70);

rootNode.left = node20;

rootNode.right = node60;

node20.left = node10;

node20.right = node30;

node60.left = node50;

node60.right = node70;

**return** rootNode;

}

//pre order - recursive solution:

**public** **void** preOrderTraverse(TreeNode root) {

**if**(root != **null**) {

System.***out***.print(root.data+ " ");

preOrderTraverse(root.left);

preOrderTraverse(root.right);

}

}

**public** **static** **void** main(String[] args) {

BTPreOrder bt = **new** BTPreOrder();

TreeNode rootNode = *createBinaryTree*();

bt.preOrderTraverse(rootNode);

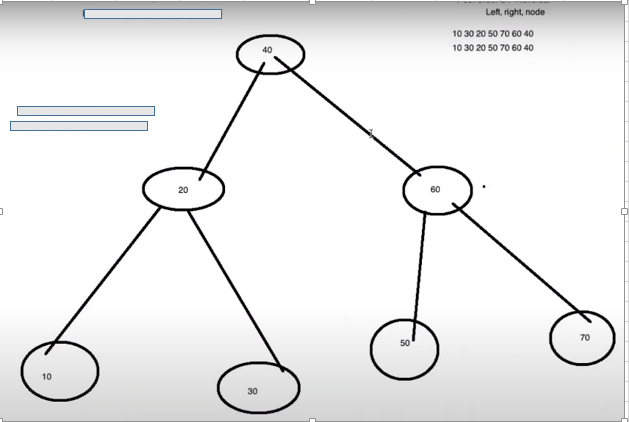
}

}

Output:

40 20 10 30 60 50 70

**PostOrder traversal**



Left, right, then Node

Program:

**package** datastructures;

**public** **class** BTPostOrder {

//used to create the root node

**public** **static** **class** TreeNode{

**int** data;

TreeNode left;

TreeNode right;

TreeNode(**int** data){

**this**.data=data;

}

}

**public** **static** TreeNode createBinaryTree() {

TreeNode rootNode = **new** TreeNode(40);

TreeNode node20 = **new** TreeNode(20);

TreeNode node10 = **new** TreeNode(10);

TreeNode node30 = **new** TreeNode(30);

TreeNode node60 = **new** TreeNode(60);

TreeNode node50 = **new** TreeNode(50);

TreeNode node70 = **new** TreeNode(70);

rootNode.left = node20;

rootNode.right = node60;

node20.left = node10;

node20.right = node30;

node60.left = node50;

node60.right = node70;

**return** rootNode;

}

//post order - recursive solution:

**public** **void** postOrderTraverse(TreeNode root) {

**if**(root != **null**) {

postOrderTraverse(root.left);

postOrderTraverse(root.right);

System.***out***.print(root.data+ " ");

}

}

**public** **static** **void** main(String[] args) {

BTPostOrder bt = **new** BTPostOrder();

TreeNode rootNode = *createBinaryTree*();

bt.postOrderTraverse(rootNode);

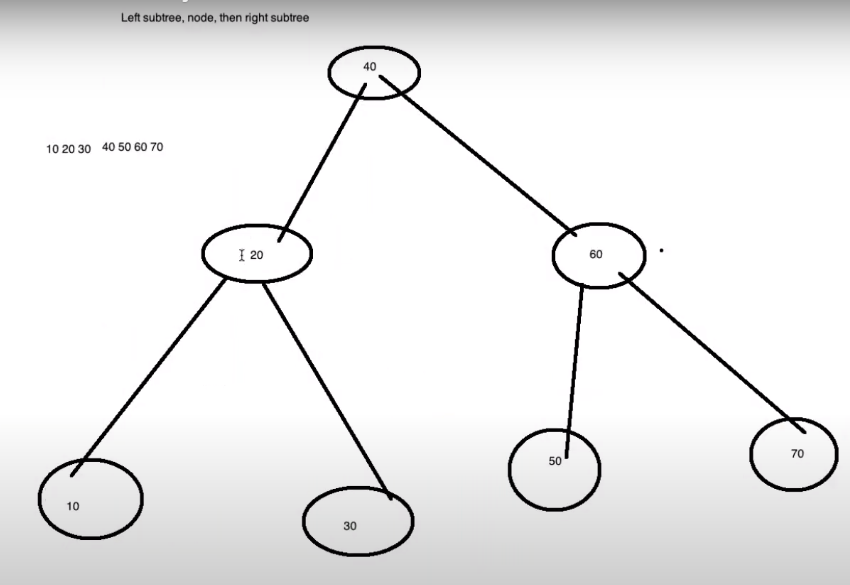
}

}

Output:

10 30 20 50 70 60 40

**InOrder traversal**



Left subtree, node, then right subtree

Program:

**package** datastructures;

**public** **class** BTInOrder {

//used to create the root node

**public** **static** **class** TreeNode{

**int** data;

TreeNode left;

TreeNode right;

TreeNode(**int** data){

**this**.data=data;

}

}

**public** **static** TreeNode createBinaryTree() {

TreeNode rootNode = **new** TreeNode(40);

TreeNode node20 = **new** TreeNode(20);

TreeNode node10 = **new** TreeNode(10);

TreeNode node30 = **new** TreeNode(30);

TreeNode node60 = **new** TreeNode(60);

TreeNode node50 = **new** TreeNode(50);

TreeNode node70 = **new** TreeNode(70);

rootNode.left = node20;

rootNode.right = node60;

node20.left = node10;

node20.right = node30;

node60.left = node50;

node60.right = node70;

**return** rootNode;

}

//In order - recursive solution:

**public** **void** inOrderTraversal(TreeNode root) {

**if**(root!=**null**) {

inOrderTraversal(root.left);

System.***out***.print(root.data + " ");

inOrderTraversal(root.right);

}

}

**public** **static** **void** main(String[] args) {

BTInOrder bt = **new** BTInOrder();

TreeNode rootNode = *createBinaryTree*();

bt.inOrderTraversal(rootNode);

}

}

Output:

10 20 30 40 50 60 70