```
class Stack {

  // store elements of stack
  private int arr[];
  // represent top of stack
  private int top;
  // total capacity of the stack
  private int capacity;

  // Creating a stack
  Stack(int size) {
    // initialize the array
    // initialize the stack variables
    arr = new int[size];
    capacity = size;
    top = -1;
  }

  // push elements to the top of stack
  public void push(int x) {
    if (isFull()) {
      System.out.println("Stack OverFlow");

      // terminates the program
      System.exit(1);
    }

    // insert element on top of stack
    System.out.println("Inserting " + x);
    arr[++top] = x;
  }

  // pop elements from top of stack
  public int pop() {

    // if stack is empty
    // no element to pop
    if (isEmpty()) {
      System.out.println("STACK EMPTY");
      // terminates the program
      System.exit(1);
    }

    // pop element from top of stack
    return arr[top--];
```

```
  }

  // return size of the stack
  public int getSize() {
    return top + 1;
  }

  // check if the stack is empty
  public Boolean isEmpty() {
    return top == -1;
  }

  // check if the stack is full
  public Boolean isFull() {
    return top == capacity - 1;
  }

  // display elements of stack
  public void printStack() {
    for (int i = 0; i <= top; i++) {
      System.out.print(arr[i] + ", ");
    }
  }

  public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(1);
    stack.push(2);
    stack.push(3);

    System.out.print("Stack: ");
    stack.printStack();

    // remove element from stack
    stack.pop();
    System.out.println("\nAfter popping out");
    stack.printStack();

  }
}
```

```
public class Queue {
  int SIZE = 5;
  int items[] = new int[SIZE];
  int front, rear;

  Queue() {
    front = -1;
    rear = -1;
  }

  // check if the queue is full
  boolean isFull() {
    if (front == 0 && rear == SIZE - 1) {
      return true;
    }
    return false;
  }

  // check if the queue is empty
  boolean isEmpty() {
    if (front == -1)
      return true;
    else
      return false;
  }

  // insert elements to the queue
  void enQueue(int element) {

    // if queue is full
    if (isFull()) {
      System.out.println("Queue is full");
    }
    else {
      if (front == -1) {
        // mark front denote first element of queue
        front = 0;
      }

      rear++;
      // insert element at the rear
      items[rear] = element;
      System.out.println("Insert " + element);
```

```
    }
  }

  // delete element from the queue
  int deQueue() {
    int element;

    // if queue is empty
    if (isEmpty()) {
      System.out.println("Queue is empty");
      return (-1);
    }
    else {
      // remove element from the front of queue
      element = items[front];

      // if the queue has only one element
      if (front >= rear) {
        front = -1;
        rear = -1;
      }
      else {
        // mark next element as the front
        front++;
      }
      System.out.println( element + " Deleted");
      return (element);
    }
  }

  // display element of the queue
  void display() {
    int i;
    if (isEmpty()) {
      System.out.println("Empty Queue");
    }
    else {
      // display the front of the queue
      System.out.println("\nFront index-> " + front);

      // display element of the queue
      System.out.println("Items -> ");
      for (i = front; i <= rear; i++)
        System.out.print(items[i] + "   ");
```

```
      // display the rear of the queue
      System.out.println("\nRear index-> " + rear);
    }
  }

  public static void main(String[] args) {

    // create an object of Queue class
    Queue q = new Queue();

    // try to delete element from the queue
    // currently queue is empty
    // so deletion is not possible
    q.deQueue();

    // insert elements to the queue
    for(int i = 1; i < 6; i ++) {
      q.enQueue(i);
    }

    // 6th element can't be added to queue because queue is full
    q.enQueue(6);

    q.display();

    // deQueue removes element entered first i.e. 1
    q.deQueue();

    // Now we have just 4 elements
    q.display();

  }
}
```

```
import java.util.*;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
```

```
    {

        // Creating an empty HashMap
        Map<String, Integer> map = new HashMap<>();

        // Inserting entries in the Map
        // using put() method
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        // Iterating over Map
        for (Map.Entry<String, Integer> e : map.entrySet())

            // Printing key-value pairs
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }
}
```

```
import java.util.*;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty LinkedHashMap
        Map<String, Integer> map = new LinkedHashMap<>();

        // Inserting pair entries in above Map
        // using put() method
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        // Iterating over Map
        for (Map.Entry<String, Integer> e : map.entrySet())

            // Printing ket-value pairs
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }
```

```
}
import java.util.*;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty TreeMap
        Map<String, Integer> map = new TreeMap<>();

        // Inserting custom elements in the Map
        // using put() method
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        // Iterating over Map using for each loop
        for (Map.Entry<String, Integer> e : map.entrySet())

            // Printing key-value pairs
            System.out.println(e.getKey() + " "
                               + e.getValue());
    }
}
```