```
public class DoublyLinkedList<E> implements Iterable<E>, Cloneable {

    public DoublyLinkedList()
    public int size()
    public boolean isEmpty()
    public E first()
    public E last()

    public void addFirst(E e)
    public void addLast(E e)
    public E removeFirst()
    public E removeLast()

    public boolean equals(Object obj)
    public Object clone() throws CloneNotSupportedException

    //------- nested DoublyLinkedListIterator class ----------------

    private class DoublyLinkedListIterator implements ListIterator<E> {

        public DoublyLinkedListIterator

        final void  checkForComodification()
        public boolean hasNext()
        public E next() throws NoSuchElementException
        public boolean hasPrevious()
        public E previous() throws NoSuchElementException
        public int nextIndex()
        public int previousIndex()
        public void remove() throws NoSuchElementException
        public void set(E e) throws NoSuchElementException
        public void add(E e)
    }

//---------------- Iterable implementation ----------------
    public Iterator<E> iterator()

    public ListIterator<E> listIterator()

//---------------- nested Node class ----------------

    private static class Node<E> {

        private E element
        private Node<E> prev;
        private Node<E> next;

        public Node(E element, Node<E> prev, Node<E> next)
        public E getElement()
        public Node<E> getPrev()
        public Node<E> getNext()
        public void setElement(E element)
        public void setPrev(Node<E> prev)
        public void setNext(Node<E> next)
  }
}
```

```
public interface BasicGraph<V,E> {
  int numVertices();
  int numEdges();
  Iterable<V> vertices();
  Iterable<E> edges();
  int outDegree(V vertex);
  int inDegree(V vertex);
  Iterable<E> outgoingEdges(V vertex);
  Iterable<E> incomingEdges(V vertex);
  E getEdge(V va, V vb);
  V[] endVertices(E edge);
  boolean insertVertex(V newVertex);
  boolean insertEdge(V va, V vb, E newEdge);
  boolean removeVertex(V vertex);
  E removeEdge(V va, V vb);
}

public class AdjacencyMatrixGraph<V, E> implements BasicGraph<V, E>, Cloneable {
  int numVertices;
  int numEdges;
  ArrayList<V> vertices;
  E[][] edgeMatrix;
  . . .
}

public class GraphAlgorithms {

  public static<V,E> LinkedList<V> DFS(AdjacencyMatrixGraph<V,E> graph, V vertex)
  public static<V,E> LinkedList<V> BFS(AdjacencyMatrixGraph<V,E> graph, V vertex)
  public static<V,E> boolean existsPath(AdjacencyMatrixGraph<V,E> graph, V source,
                                                               V dest, LinkedList<V> path)
  public static<V,E> boolean allPaths(AdjacencyMatrixGraph<V,E> graph, V source, V dest,
                                                            LinkedList<LinkedList<V>> paths)
  public static<V,E> AdjacencyMatrixGraph<V,E> transitiveClosure(AdjacencyMatrixGraph<V,E> graph,
                                                                E dummyEdge)
}


public class EdgeAsDoubleGraphAlgorithms {

  public static<V> double shortestPath(AdjacencyMatrixGraph<V,Double> graph, V source, V dest,
                                       LinkedList<V> path)
  public static<V> AdjacencyMatrixGraph<V,Double> minDistGraph(AdjacencyMatrixGraph<V,Double>
                                                                                         graph)
}
```

```
public class Vertex<V, E> {

    private int key ;
    private V  element ;
    private Map<V, Edge<V,E>> outVerts;

    public Vertex ()
    public Vertex (int k, V vInf)
    public int getKey()
    public void setKey(int k)
    public V getElement()
    public void setElement(V vInf)
    public void addAdjVert(V vAdj, Edge<V,E> edge)
    public V getAdjVert(Edge<V,E> edge)
    public void remAdjVert(V vAdj)
    public Edge<V,E> getEdge(V vAdj)
    public int numAdjVerts()
    public Iterable<V> getAllAdjVerts()
    public Iterable<Edge<V,E>> getAllOutEdges()
    public boolean equals(Object otherObj)
    public Vertex<V,E> clone()
    public String toString()
}
```

```
public class Edge<V,E> implements Comparable {

   private E element;
   private double weight;
   private Vertex<V,E> vOrig;
   private Vertex<V,E> vDest;

   public Edge()
   public Edge(E eInf, double ew,
                 Vertex<V,E> vo, Vertex<V,E> vd)
   public E getElement()
   public void setElement(E eInf)
   public double getWeight()
   public void setWeight(double ew)
   public V getVOrig()
   public void setVOrig(Vertex<V,E> vo)
   public V getVDest()
   public void setVDest(Vertex<V,E> vd)
   public V[] getEndpoints()
   public boolean equals(Object otherObj)
   public int compareTo(Object otherObject)
   public Edge<V,E> clone()
   public String toString()
}
```

```
public interface GraphInterface<V,E> {
    int numVertices();
    Iterable<V> vertices();
    Iterable<V> adjVertices(V vert);
    int numEdges();
    Iterable<Edge<V,E>> edges();
    Edge<V,E> getEdge(V vOrig, V vDest);
    Object[] endVertices(Edge<V,E> edge);
    V opposite(V vert, Edge<V,E> edge);
    int outDegree(V vert) ;
    int inDegree(V vert) ;
    Iterable<Edge<V,E>> outgoingEdges (V vert);
    Iterable<Edge<V,E>> incomingEdges(V vert);
    boolean insertVertex(V newVert);
    boolean insertEdge(V vOrig, V vDest, E edge, double eWeight);
    boolean removeVertex(V vert);
    boolean removeEdge(V vOrig, V vDest);
}


public class Graph<V,E> implements GraphInterface<V,E> {

    private int numVert;
    private int numEdge;
    private boolean isDirected;
    private Map<V,Vertex<V,E>> vertices;  //all Vertices of the graph
    . . .
}
```

```
public class GraphAlgorithms {
   public static<V,E> LinkedList<V> BreadthFirstSearch(Graph<V,E> g, V vert)
   public static<V,E> LinkedList<V> DepthFirstSearch(Graph<V,E> g, V vert)
   public static<V,E> ArrayList<LinkedList<V>> allPaths(Graph<V,E> g, V vOrig, V vDest)
   public static<V,E> double shortestPath(Graph<V,E> g, V vOrig, V vDest, LinkedList<V> shortPath)
}


public class BST <E extends Comparable<E>> {

  public BST()
  protected Node<E> root()
  public boolean isEmpty()
  public int size()

  public void insert(E element)
  public E smallestElement()

  public int height()
  §public Map<Integer,List<E>> nodesByLevel()
  public Iterable<E> inOrder()
  public Iterable<E> preOrder()
  public Iterable<E> postOrder()


  //---------------- nested Node class ----------------

  protected static class Node<E> {

    private E element;
    private Node<E> left;
    private Node<E> right;

    public Node(E e, Node<E> leftChild, Node<E> rightChild)

    public E getElement()
    public Node<E> getLeft()
    public Node<E> getRight()

    public void setElement(E e)
    public void setLeft(Node<E> leftChild)
    public void setRight(Node<E> rightChild)
  }
}
```

```
public interface PriorityQueue<K,V> {

  int size();
  boolean isEmpty();
  Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
  Entry<K,V> min();
  Entry<K,V> removeMin();
}

public interface Entry<K,V> {
  K getKey();
  V getValue();
}

public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {

  //----------- Nested PQEntry class -----------

  protected static class PQEntry<K,V> implements Entry<K,V> {

    public PQEntry(K key, V value)

    public K getKey() { return k; }
    public V getValue() { return v; }

    protected void setKey(K key)
    protected void setValue(V value)
  }

  private Comparator<K> comp;

  protected AbstractPriorityQueue(Comparator<K> c)
  protected AbstractPriorityQueue()
  protected int compare(Entry<K,V> a, Entry<K,V> b)
  protected boolean checkKey(K key) throws IllegalArgumentException
  public boolean isEmpty()

}


public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {

  public HeapPriorityQueue()
  public HeapPriorityQueue(Comparator<K> comp)
  public HeapPriorityQueue(K[] keys, V[] values)

  protected void swap(int i, int j
  protected void percolateUp(int j)
  protected void buildHeap()

  public int size()
  public Entry<K,V> min()
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException
  public Entry<K,V> removeMin ()
  public HeapPriorityQueue<K,V> clone()

}
```

```
public interface Map<K,V> {
    void                  clear()
    boolean               containsKey(Object key)
    boolean               containsValue(Object value)
    Set<Map.Entry<K,V>>   entrySet()
    boolean               equals(Object o)
    V                     get(Object key)
    int                   hashCode()
    boolean               isEmpty()
    Set<K>                keySet()
    V                     put(K key, V value)
    void                  putAll(Map<? extends K,? extends V> m)
    V                     remove(Object key)
    int                   size()
    Collection<V>         values()
}

public interface Set<E>{
     boolean        add(E e)
     boolean        addAll(Collection<? extends E> c)
     void           clear()
     boolean        contains(Object o)
     boolean        containsAll(Collection<?> c)
     boolean        equals(Object o)
     int            hashCode()
     boolean        isEmpty()
     Iterator<E>    iterator()
     boolean        remove(Object o)
     boolean        removeAll(Collection<?> c)
     boolean        retainAll(Collection<?> c)
     int            size()
     Object[]       toArray()
}
```