```
public interface Stack {
        public int size();
        public boolean isEmpty();
        public Object top()  throws StackEmptyException;
        public void push (Object element);
         public Object pop() throws StackEmptyException; }
```

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public Object front() throws QueueEmptyException;
  public void enqueue(Object element);
  public Object dequeue() throws QueueEmptyException;
}
```

```
public interface Deque {
  public void insertFirst(Object o);
  public void insertLast(Object o);
  public Object removeFirst() throws DequeEmptyException;
  public Object removeLast() throws DequeEmptyException;
  public Object first() throws DequeEmptyException;
  public Object last() throws DequeEmptyException;
  public int size();
}
```

```
public interface Vector {
  public int size();
  public boolean isEmpty();
  public Object elemAtRank(int r) throws OutOfBoundaryException;
  public void replaceAtRank(int r, Object e) throws OutOfBoundaryException;
  public Object removeAtRank(int r) throws OutOfBoundaryException;
  public void insertAtRank(int r, Object e) throws OutOfBoundaryException;
}
```

```
public interface Position {
    public Object element();
}
```

```
public interface List {
  public int size();
  public boolean isEmpty();
  public boolean isFirst(Position p) throws InvalidPositionException;
  public boolean isLast(Position p) throws InvalidPositionException;
  public Position first() throws EmptyContainerException;
  public Position last() throws EmptyContainerException;
  public Position prev(Position p) throws InvalidPositionException,
        BoundaryViolationException;
  public Position next(Position p) throws InvalidPositionException,
        BoundaryViolationException;
  public Position insertBefore(Position p, Object element) throws InvalidPositionException;
  public Position insertAfter(Position p, Object element) throws InvalidPositionException;
  public Position insertFirst(Object element);
  public Position insertLast(Object element);
  public Object remove(Position p) throws InvalidPositionException;
  public void replaceElement(Position p, Object element) throws InvalidPositionException;
  public void swapElements(Position a, Position b) throws  InvalidPositionException;
  public Iterator positions();
  public Iterator elements();
}

public interface Sequence extends List, Vector {
  public Position atRank(int rank) throws BoundaryViolationException;
  public int rankOf(Position position) throws InvalidPositionException;
}


public interface Iterator {
        public boolean hasNext();
        public Object next();
}

public interface Tree {
  public int size();
  public boolean isEmpty();
  public Iterator elements();
  public Iterator positions();
  public void swapElements(Position v, Position w) throws InvalidPositionException;
  public Object replaceElement(Position v, Object e) throws InvalidPositionException;
  public Position root() throws EmptyTreeException;
  public Position parent(Position v) throws InvalidPositionException,
                                                BoundaryViolationException;
  public Iterator children(Position v) throws InvalidPositionException;
  public boolean isInternal(Position v) throws InvalidPositionException;
  public boolean isExternal(Position v) throws InvalidPositionException;
  public boolean isRoot(Position v) throws InvalidPositionException;
}
```

```
public interface BinaryTree  extends Tree {
  public Position leftChild(Position v) InvalidPositionException,
                                                    BoundaryViolationException;
   public Position rightChild(Position v) InvalidPositionException,
                                                    BoundaryViolationException;
  public Position sibling(Position v) InvalidPositionException;
}

public interface Entry {
  public Object key();
  public Object value();
}

public interface Comparator {
   public int compare(Object a, Object b) throws ClassCastException;
}

public interface PriorityQueue {
  public int size();
  public boolean isEmpty();
  public Entry min() throws EmptyPriorityQueueException;
  public Entry insert(Object key, Object value) throws InvalidKeyException;
  public Entry removeMin() throws EmptyPriorityQueueException;
}

public interface CompleteBinaryTree extends BinaryTree {
  public Position add(Object elem);
  public Object remove();
}

public interface AdaptablePriorityQueue extends PriorityQueue {
  public Entry remove(Entry e);
  public Object replaceKey(Entry e, Object key) throws InvalidKeyException;
  public Object replaceValue(Entry e, Object value);
}

public interface Dictionary {
  public int size();
  public boolean isEmpty();
  public Entry find(Object key) throws InvalidKeyException;
  public Iterator findAll(Object key) throws InvalidKeyException;
  public Entry insert(Object key, Object value) throws InvalidKeyException;
  public Entry remove(Entry e) throws InvalidEntryException;
  public Iterator entries();
}

public interface EqualityTester {
  public boolean isEqualTo(Object x, Object y) throws ClassCastException;
}
```

```
public interface OrderedDictionary extends Dictionary {
        public Entry first();
        public Entry last();
        public Iterator successors(Object key) throws InvalidKeyException;
        public Iterator predecessors(Object key) throws InvalidKeyException;
}

public interface Map {
  public int size();
  public boolean isEmpty();
  public Object put(Object key, Object value) throws InvalidKeyException;
  public Object get(Object key) throws InvalidKeyException;
  public Object remove(Object key) throws InvalidKeyException;
  public Iterator keys();
  public Iterator values();
}

public interface Set {
  public int size();
  public boolean isEmpty();
  public Set union(Set B);
  public Set intersect(Set B);
  public Set subtract(Set B);
}

public interface Element {
  public Set set();
  public Object element();
}

public interface Partition {
  public int size();
  public boolean isEmpty();
  public Set makeSet(Object x);
  public Set union(Set A, Set B);
  public Set find(Object x);
}

public interface PartitionAware {
  public int size();
  public boolean isEmpty();
  public Set makeSet(Element x);
  public Set union(Set A, Set B);
  public Set find(Element x);
}

public interface Vertex extends Position { }

public interface Edge extends Position { }
```

```
public interface InspectableContainer {
  public int size();
  boolean isEmpty();
  Iterator elements();
}

public interface PositionalContainer extends InspectablePositionalContainer {
  public void swapElements(Position v, Position w);
  public Object replaceElement(Position v, Object e);
}

public interface InspectablePositionalContainer extends InspectableContainer {
  public Iterator positions();
}

public interface InspectableGraph extends InspectablePositionalContainer {
  public int numVertices();
  public int numEdges();
  public VertexIterator vertices();
  public Vertex aVertex();
  public EdgeIterator edges();
  public Edge anEdge();
  public boolean areAdjacent(Vertex v1, Vertex v2);
  public int degree(Vertex v);
  public Iterator adjacentVertices(Vertex v);
  public Iterator incidentEdges(Vertex v);
  public Vertex[] endVertices(Edge e);
  public Vertex opposite(Vertex v, Edge e);
}

public interface Graph extends PositionalContainer, InspectableGraph {
  public Vertex insertVertex(Object element);
  public Edge insertEdge (Vertex v1, Vertex v2, Object element);
  public void removeVertex(Vertex v);
  public void removeEdge(Edge e);
  public void removeEdge(Vertex v1, Vertex v2);
  public void replace(Vertex v, Object x);
 public void replace(Edge e, Object x);
}

public interface InspectableDirectedGraph extends InspectableGraph{
  public Vertex origin (Edge e);
  public Vertex destination (Edge e);
  public int inDegree(Vertex v);
  public int outDegree(Vertex v);
  public EdgeIterator outIncidentEdges(Vertex v);
  public EdgeIterator inIncidentEdges(Vertex v);
  public VertexIterator outAdjacentVertices(Vertex v);
  public VertexIterator inAdjacentVertices(Vertex v);
}
```

```
public interface UndirectedGraph extends Graph{}

public interface DirectedGraph  extends Graph, InspectableDirectedGraph {
  public void reverseDirection(Edge e);
  public void setDirectionFrom(Edge e, Vertex v);
  public void setDirectionTo(Edge e, Vertex v);
}
```