```
interface IShape {
 <T> T accept(IShapeVisitor<T> func);
}

class Circle implements IShape {
 int radius;
 Circle(int r) {
   this.radius = r;
 }
 public <T> T accept(IShapeVisitor<T> func) {
   return func.visitCircle(this);
 }
}

class Rectangle implements IShape {
 int length;
 int width;
 Rectangle(int l, int w) {
   this.length = l;
   this.width = w;
 }
 public <T> T accept(IShapeVisitor<T> func) {
   return func.visitRectangle(this);
 }
}

interface IShapeVisitor<T> extends Function<IShape, T> {
 T visitCircle(Circle c);
 T visitRectangle(Rectangle r);
}

interface IList<T> {
 <R> R accept(IListVisitor<T, R> visitor);
}

class MtList<T> implements IList<T> {
 MtList() {
 }
 public <R> R accept(IListVisitor<T, R> visitor) {
   return visitor.visitMtList(this);
 }
}

class ConsList<T> implements IList<T> {
 T first;
 IList<T> rest;
 ConsList(T first, IList<T> rest) {
   this.first = first;
   this.rest = rest;
 }
 public <R> R accept(IListVisitor<T, R> visitor) {
   return visitor.visitConsList(this);
 }
}

interface IListVisitor<T, R> {
 R visitMtList(MtList<T> list);
 R visitConsList(ConsList<T> list);
}

class MapVisitor<T, R> implements IListVisitor<T,
IList<R>>, Function<IList<T>, IList<R>> {
 Function<T, R> func;
 MapVisitor(Function<T, R> func) {
   this.func = func;
 }
 public IList<R> apply(IList<T> element) {
   return element.accept(this);
 }
 public IList<R> visitMtList(MtList<T> list) {
   return new MtList<R>();
 }
 public IList<R> visitConsList(ConsList<T> list) {
   return new ConsList<R>(func.apply(list.first),
list.rest.accept(this));
 }
}

class FoldRVisitor<T, R> implements IListVisitor<T, R> {
 BiFunction<T, R, R> func;
 R base;
 FoldRVisitor(BiFunction<T, R, R> func, R base) {
   this.func = func;
   this.base = base;
 }
 public R visitMtList(MtList<T> mt) {
   return base;
 }
 public R visitConsList(ConsList<T> cons) {
   return func.apply(cons.first,
cons.rest.accept(this));
 }
}

class AppendVisitor<T> implements IListVisitor<T,
IList<T>> {
 IList<T> other;
 AppendVisitor(IList<T> other) {
   this.other = other;
 }
 public IList<T> visitMtList(MtList<T> list) {
   return other;
 }
 public IList<T> visitConsList(ConsList<T> list) {
   return new ConsList<T>(list.first,
list.rest.accept(this));
 }
}


class ArrayUtils {
 <T, U> ArrayList<U> map(ArrayList<T> arr, Function<T,
U> func){
   ArrayList<U> result = new ArrayList<U>();
   for (T item : arr) {
     result.add(func.apply(item));
   }
   return result;
 }
 <T> ArrayList<T> filter(ArrayList<T> array,
Predicate<T> predicate) {
   return customFilter(array, predicate, true);
 }
 <T> ArrayList<T> filterNot(ArrayList<T> array,
Predicate<T> predicate) {
   return customFilter(array, predicate, false);
 }
 <T> ArrayList<T> customFilter(ArrayList<T> array,
Predicate<T> predicate, boolean shouldAdd) {
   ArrayList<T> result = new ArrayList<T>();
   for (T element : array) {
     if (predicate.test(element) == shouldAdd) {
       result.add(element);
     }
   }
   return result;
 }
 <T> void removeFailing(ArrayList<T> array, Predicate<T>
predicate) {
   customRemove(array, predicate, false);
 }
 <T> void removePassing(ArrayList<T> array, Predicate<T>
predicate) {
   customRemove(array, predicate, true);
 }
 <T> void customRemove(ArrayList<T> array, Predicate<T>
predicate, boolean shouldRemove) {
   for (int i = array.size() - 1; i >= 0; i--) {
     if (predicate.test(array.get(i)) == shouldRemove) {
       array.remove(i);
     }
   }
 }
```

```
 <T> ArrayList<T> interweave(ArrayList<T> arr1,
ArrayList<T> arr2) {
   return customInterweave(arr1, arr2, 1, 1);
 }
 <T> ArrayList<T> customInterweave(ArrayList<T> arr1,
ArrayList<T> arr2, int getFrom1, int getFrom2) {
   ArrayList<T> result = new ArrayList<T>();
   for (int i = 0, j = 0; i < arr1.size() || j <
arr2.size();) {
     for (int k = 0; k < getFrom1 && i < arr1.size();
k++, i++) {
       result.add(arr1.get(i));
     }
     for (int k = 0; k < getFrom2 && j < arr2.size();
k++, j++) {
       result.add(arr2.get(j));
     }
   }
   return result;
 }
}

class PersonIterator implements Iterator<Person> {
 ArrayList<Person> allPeople;
 int index;
 PersonIterator(Person person) {
   this.allPeople = new ArrayList<Person>();
   this.allPeople.add(person);
   this.index = 0;
 }
 public boolean hasNext() {
   return index < this.allPeople.size();
 }
 public Person next() {
   if (!this.hasNext()) {
     throw new NoSuchElementException("No more people in
this family tree");
   }
   Person toReturn = this.allPeople.get(this.index);
   this.index++;
   ArrayList<Person> childrenToProcess = new
ArrayList<Person>(toReturn.listOfChildren);
   for (Person child : childrenToProcess) {
     this.allPeople.add(child);
   }
   return toReturn;
 }
}

class ListOfListsIterator<T> implements Iterator<T> {
 int firstIndex = 0;
 int secondIndex = 0;
 ArrayList<ArrayList<T>> lists;
 ListOfListsIterator(ArrayList<ArrayList<T>> lists) {
   this.lists = lists;
 }
 public boolean hasNext() {
   while (firstIndex < lists.size()) {
     if (secondIndex < lists.get(firstIndex).size()) {
       return true;
     }
     else {
       firstIndex++;
       secondIndex = 0;
     }
   }
   return false;
 }
 public T next() {
   if (!hasNext()) {
     throw new NoSuchElementException("No more
elements");
   }
   return lists.get(firstIndex).get(secondIndex++);
 }
}

class CycleIterator<T> implements Iterator<T> {
 Iterable<T> iterable;
 Iterator<T> current;
 CycleIterator(Iterable<T> iterable) {
   this.iterable = iterable;
   this.current = iterable.iterator();
 }
 public boolean hasNext() {
   return iterable.iterator().hasNext();
 }
 public T next() {
   if (!current.hasNext()) {
     current = iterable.iterator();
     if (!current.hasNext()) {
       throw new NoSuchElementException("There are no
more items");
     }
   }
   return current.next();
 }
}

class Student {
 String name;
 int age;

 Student(String name, int age) {
   this.name = name;
   this.age = age;
 }
 @Override
 public boolean equals(Object o) {
   if (this == o) {
     return true;
   }

   if (! (o instanceof Student)) {
     return false;
   }

   Student that = (Student) o;

   return this.name.equals(that.name) && this.age ==
that.age;
 }
 public int hashCode() {
   return this.name.hashCode() * 1000 + this.age;
 }
}
```

If we override equals such that objA.equals(objB) is true, then we must also override hashCode to ensure that objA.hashCode() == objB.hashCode().

If we override equals such that objA.equals(objB) is false, then objA.hashCode() and objB.hashCode() may or may not be the same.

If we override hashCode such that objA.hashCode() != objB.hashCode(), then we must also override equals to ensure that objA.equals(objB) is false.

If we override hashCode such that objA.hashCode() == objB.hashCode(), then objA.equals(objB) may or may not be true.

add(E e), add(int index, E element)
addAll(Collection<? extends E> c)
addAll(int index, Collection<? extends E> c)
clear()
contains(Object o)
get(int index)
indexOf(Object o) (-1 if doesn't exist)
isEmpty()
iterator()
remove(int index)
removeRange(int from, int to) inclusive, exclusive
set(int index, E element)
size()
subList(int from, int to)

```
abstract class ABST<T> {
 Comparator<T> order;
 ABST(Comparator<T> order) {
   this.order = order;
 }
 abstract ABST<T> insert(T item);
 abstract boolean present(T item);
 abstract T getLeftmost();
 abstract T getLeftmostHelper(T current);
 abstract ABST<T> getRight();
 abstract boolean sameTree(ABST<T> given);
 abstract boolean sameNode(Node<T> given);
 abstract boolean sameData(ABST<T> given);
 abstract boolean sameDataNode(Node<T> given);
 abstract boolean sameDataLeaf(Leaf<T> given);
 abstract IList<T> buildList();
}

class Leaf<T> extends ABST<T> {
 Leaf(Comparator<T> order) {
   super(order);
 }
 public ABST<T> insert(T item) {
   return new Node<>(this.order, item, new
Leaf<>(this.order), new Leaf<>(this.order));
 }
 public boolean present(T item) {
   return false;
 }
 public T getLeftmost() {
   throw new RuntimeException("No leftmost item of an
empty tree");
 }
 public T getLeftmostHelper(T current) {
   return current;
 }
 public ABST<T> getRight() {
   throw new RuntimeException("No right of an empty
tree");
 }
 public boolean sameTree(ABST<T> given) {
   return given.sameDataLeaf(this);
 }
 public boolean sameNode(Node<T> given) {
   return false;
 }
 public boolean sameData(ABST<T> given) {
   return given.sameDataLeaf(this);
 }
 public boolean sameDataNode(Node<T> given) {
   return false;
 }
 public boolean sameDataLeaf(Leaf<T> given) {
   return true;
 }
 public IList<T> buildList() {
   return new MtList<T>();
 }
}

class Node<T> extends ABST<T> {
 T data;
 ABST<T> left;
 ABST<T> right;
 Node(Comparator<T> order, T data, ABST<T> left, ABST<T>
right) {
   super(order);
   this.data = data;
   this.left = left;
   this.right = right;
 }
 public ABST<T> insert(T item) {
   if (order.compare(item, this.data) < 0) {
     return new Node<>(order, this.data,
this.left.insert(item), this.right);
   }
   else {
     return new Node<>(order, this.data, this.left,
this.right.insert(item));
   }
 }
 public boolean present(T item) {
   int comparison = order.compare(item, data);
   return comparison == 0 || left.present(item) ||
right.present(item);
 }
 public T getLeftmost() {
   return this.getLeftmostHelper(this.data);
 }
 public T getLeftmostHelper(T current) {
   return this.left.getLeftmostHelper(this.data);
 }
 public ABST<T> getRight() {
   int comparison = order.compare(this.data,
this.getLeftmost());
   if (comparison == 0) {
     return this.right;
   }
   else {
     return new Node<T>(this.order, this.data,
this.left.getRight(), this.right);
   }
 }
 public boolean sameTree(ABST<T> given) {
   return given.sameNode(this);
 }
 public boolean sameNode(Node<T> given) {
   return this.order.compare(this.data, given.data) == 0
&& this.left.sameTree(given.left)
       && this.right.sameTree(given.right);
 }
 public boolean sameData(ABST<T> given) {
   return given.sameDataNode(this);
 }
 public boolean sameDataNode(Node<T> given) {
   return this.order.compare(this.getLeftmost(),
given.getLeftmost()) == 0
       && this.getRight().sameData(given.getRight());
 }
 public boolean sameDataLeaf(Leaf<T> given) {
   return false;
 }
 public IList<T> buildList() {
   return new ConsList<T>(this.getLeftmost(),
this.getRight().buildList());
 }
}

class BooksByPrice implements Comparator<Book> {
 public int compare(Book o1, Book o2) {
   if (o1.price < o2.price) {
     return -1;
   }
   else if (o1.price == o2.price) {
     return 0;
   }
   else {
     return 1;
   }
 }
}

class MapVisitor<T, R> implements IListVisitor<T,
IList<R>> {
 Function<T, R> func;
 MapVisitor(Function<T, R> func) {
   this.func = func;
 }
```

```
 public IList<R> apply(IList<T> element) {
   return element.accept(this);
 }
 public IList<R> visitMtList(MtList<T> list) {
   return new MtList<R>();
 }
 public IList<R> visitConsList(ConsList<T> list) {
   return new ConsList<R>(func.apply(list.first),
list.rest.accept(this));
 }
}

class FoldRVisitor<T, R> implements IListVisitor<T, R> {
 BiFunction<T, R, R> func;
 R base;
 FoldRVisitor(BiFunction<T, R, R> func, R base) {
   this.func = func;
   this.base = base;
 }
 public R apply(IList<T> list) {
   return list.accept(this);
 }
 public R visitMtList(MtList<T> mt) {
   return base;
 }
 public R visitConsList(ConsList<T> cons) {
   return func.apply(cons.first,
cons.rest.accept(this));
 }
}

class Instructor {
 String name;
 IList<Course> courses;
 Instructor(String name) {
   this.name = name;
   this.courses = new MtList<Course>();
 }
 public void addCourse(Course c) {
   this.courses = new ConsList<Course>(c, this.courses);
 }
 boolean dejavu(Student s) {
   return new FoldRVisitor<Boolean, Integer>(new
AddOneIfTrue(), 0).apply(
       new MapVisitor<Course, Boolean>((Course c) ->
c.hasStudent(s)).apply(this.courses)) >= 2;
 }
}

class Student {
 String name;
 int id;
 IList<Course> courses;
 Student(String name, int id) {
   this.name = name;
   this.id = id;
   this.courses = new MtList<Course>();
 }
 public void enroll(Course c) {
   this.courses = new ConsList<Course>(c, this.courses);
   c.addStudent(this);
 }
 public boolean classmates(Student s) {
   return new FoldRVisitor<Boolean, Boolean>((b1, b2) ->
b1 || b2, false)
       .apply(new MapVisitor<Course, Boolean>((Course c)
-> c.hasStudent(s)).apply(this.courses));
 }
}

class Course {
 String name;
 Instructor prof;
 IList<Student> students;
 Course(String name, Instructor prof) {
   this.name = name;
   this.prof = prof;
   this.students = new MtList<Student>();
   this.prof.addCourse(this);
 }
 public void addStudent(Student s) {
   this.students = new ConsList<Student>(s,
this.students);
 }
 boolean hasStudent(Student s) {
   return new FoldRVisitor<Boolean, Boolean>((b1, b2) ->
b1 || b2, false)
       .apply(new MapVisitor<Student, Boolean>(student
-> student == s).apply(this.students));
 }
}

abstract class ANode<T> {
 ANode<T> next;
 ANode<T> prev;
 public ANode() {
   this.next = null;
   this.prev = null;
 }
 public abstract ANode<T> find(Predicate<T> pred);
 public abstract T removeFromHead();
 public abstract T removeFromTail();
 public void changeNext(ANode<T> next) {
   this.next = next;
 }
 public void changePrev(ANode<T> prev) {
   this.prev = prev;
 }
}

class Sentinel<T> extends ANode<T> {
 public Sentinel() {
   this.next = this;
   this.prev = this;
 }
 public ANode<T> find(Predicate<T> pred) {
   return this;
 }
 public T removeFromHead() {
   throw new RuntimeException("Cannot remove from an
empty deque.");
 }
 public T removeFromTail() {
   throw new RuntimeException("Cannot remove from an
empty deque.");
 }
}

class Node<T> extends ANode<T> {
 T data;
 public Node(T data) {
   this.data = data;
   this.next = null;
   this.prev = null;
 }
 public Node(T data, ANode<T> next, ANode<T> prev) {
   if (next == null || prev == null) {
     throw new IllegalArgumentException("Next or
previous node cannot be null.");
   }
   this.data = data;
   this.next = next;
   this.prev = prev;
   this.next.changePrev(this);
   this.prev.changeNext(this);
 }
 public ANode<T> find(Predicate<T> predicate) {
   if (predicate.test(this.data)) {
     return this;
   }
   else {
     return this.next.find(predicate);
   }
 }
 public T removeFromHead() {
   this.prev.changeNext(this.next);
   this.next.changePrev(this.prev);
   return this.data;
 }
 public T removeFromTail() {
   this.prev.changeNext(this.next);
   this.next.changePrev(this.prev);
   return this.data;
 }
}

class Deque<T> {
 Sentinel<T> header;
 public Deque() {
   this.header = new Sentinel<>();
 }
 public Deque(Sentinel<T> sentinel) {
   this.header = sentinel;
 }
 public int size() {
   int count = 0;
   ANode<T> current = this.header.next;
   while (current != this.header) {
     count++;
     current = current.next;
   }
   return count;
 }
 public void addAtHead(T value) {
   new Node<>(value, this.header.next, this.header);
 }
 public void addAtTail(T value) {
   new Node<>(value, this.header, this.header.prev);
 }
 public T removeFromHead() {
   return this.header.next.removeFromHead();
 }
 public T removeFromTail() {
   return this.header.prev.removeFromTail();
 }
 public ANode<T> find(Predicate<T> predicate) {
   return this.header.next.find(predicate);
 }
 public void specialAppend(Deque<T> given) {
   while (given.size() > 0) {
     this.addAtTail(given.removeFromHead());
   }
 }
}
```