Exercise 8

Objectives

The aim of this exercise is to learn the concept of polymorphism.

A: OpenList [3 pt]

The following TreeTraverser class traverses a complete binary tree with N nodes. The traverse method of this class can implement various algorithms depending on the given data structure of OpenList.

```
class TreeTraverser{

    private int N = 0;

    public TreeTraverser(int N){
        this.N = N;
    }

    public void traverse(OpenList list){
        list.push(0);                    // set the starting point

        while(!list.isEmpty()){
            int u = list.pop();
            if ( u >= N ) continue;
            System.out.print(u + " "); // print the visited node
            list.push(2*u + 1);        // visit the left child
            list.push(2*u + 2);        // visit the right child
        }

        System.out.println();
    }
}
```

OpenList is an interface for implementing a data structure that holds the node numbers of a tree (0, 1, .., N-1), and the nodes can be added by the push method and retrieved (deleted) by the pop method.

Your task is to create an application by implementing a Stack class and a Queue class that implements OpenList and perform depth-first search and breadth-first search, respectively, on the complete binary tree.

If you run the TreeTraverseApplication shown below and get the following output, you have implemented the data structures correctly.

```
class TreeTraverseApplication{
   public static void main(String[] args){
      TreeTraverser traverser = new TreeTraverser(15);

      System.out.println("Depth First Search:");
      traverser.traverse(new Stack());
      System.out.println();
      System.out.println("Breadth First Search:");
      traverser.traverse(new Queue());
   }
}
```

```
Depth First Search:
0 2 6 14 13 5 12 11 1 4 10 9 3 8 7

Breadth First Search:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Submission Files	**Types**
Stack.java	Java Class
Queue.java	Java Class
OpenList.java	Java Interface
TreeTraverser.java	Java Class
TreeTraverseApplication.java	Java Class

B: Sorting Points by Comparator [4 pt]

For this assignment, you may extend or reuse the Point classe you previously created.

The following SortingPointApplication class is an application that aligns a given sequence of points with specified criteria. The number of points and their x, y coordinates are given as standard input, and they are aligned using an instance of the SortingPointMachine class.

```
import java.util.Scanner;

class SortingPointApplication{

   public SortingPointApplication(){
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      Point[] p = new Point[n];

      for ( int i = 0; i < n; i++ )
         p[i] = new Point(sc.nextInt(), sc.nextInt());

      SortingPointMachine machine = new SortingPointMachine(new XYComparator());
      machine.sort(p);
      System.out.println("Sorted by X-Y:");
      for ( int i = 0; i < p.length; i++ ) p[i].print();

      machine.setComparator(new YXComparator());
      machine.sort(p);
      System.out.println("Sorted by Y-X:");
      for ( int i = 0; i < p.length; i++ ) p[i].print();
   }

   public static void main(String[] args){
      new SortingPointApplication();
   }
}
```

The following is the SortingPointMachine class, which has a reference comparator to the comparison object, which can be set by the constructor or the setComparator method. The sort method is implemented in a simple selection sort algorithm, and the comparator is used to compare the positional relationship of the points.

```
class SortingPointMachine{
   PointComparator comparator;

   public SortingPointMachine(PointComparator comparator){
      this.comparator = comparator;
   }

   public void setComparator(PointComparator comparator){
      this.comparator = comparator;
   }

   public void sort(Point[] a){
      for ( int i = 0; i < a.length; i++ ){
         int mini = i;
         for ( int j = i+1; j < a.length; j++ ){
            if ( comparator.compare(a[j], a[mini]) < 0 ) mini = j;
         }
         Point t = a[mini];
         a[mini] = a[i];
         a[i] = t;
      }
   }
}
```

The following is the PointComparator interface.

```
public interface PointComparator{
   int compare(Point p1, Point p2);
}
```

Your task is to implement the following two classes that implement the PointComparator interface.

Class Name	**Criteria**	**Details of compare method**
XYComparator	Priority is given to x-coordinates in decreasing order.	If x of p1 is smaller than x of p2, return -1. If x of p1 is greater than x of p2, return 1. If x of p1 and p2 are the same, -1, 0, and 1 is return respectively if y of p1 is smaller than y of p2, y of p1 and y of p2 is equal, and y of p1 is greater than y of p2.
YXComparator	Priority is given to y-coordinates in decreasing order.	If y of p1 is smaller than y of p2, return -1. If y of p1 is greater than y of p2, return 1. If y of p1 and p2 are the same, -1, 0, and 1 is return respectively if x of p1 is smaller than x of p2, x of p1 and x of p2 is equal, and x of p1 is greater than x of p2.

Validate the SortingPointApplication with the following input/output data.

Sample Input

```
8
7 5
3 5
1 2
4 8
4 1
11 9
6 6
2 1
```

Sample Output

```
Sorted by X-Y:
(1, 2)
(2, 1)
(3, 5)
(4, 1)
(4, 8)
(6, 6)
(7, 5)
(11, 9)
Sorted by Y-X:
(2, 1)
(4, 1)
(1, 2)
(3, 5)
(7, 5)
(6, 6)
(4, 8)
(11, 9)
```

Submission Files	**Types**
`PointComparator.java`	`Java Interface`
`XYComparator.java`	`Java Class`
`YXComparator.java`	`Java Class`
`Point.java`	`Java Class`
`SortingPointMachine.java`	`Java Class`
`SortingPointApplication.java`	`Java Class`

C: Iterator [5 pt]

For this assignment, you may extend and reuse the SimpleList and SimpleNode classes you previously created.

条件

The concept of accessing and traversing the elements of a data collection in order by specific criteria is called Iterator. Let's build a simple application according to the following design technique.

Define the Aggregate interface as an abstract concept that represents a collection of data, and the Aggregate interface brings in methods to create Iterators. For this assignment, we will generate Iterators that manipulate elements in order and Iterators that manipulate elements in reverse order, as follows:

```
public interface Aggregate{
    Iterator forwardIterator();
    Iterator backwardIterator();
}
```

The Iterator interface is an abstract concept that represents "something to traverse" and has the following two methods:

- hasNext(): checks if the next element exists
- next(): retrieves the next element if it exists

取得する

So, the Iterator interface is implemented as follows

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

Here, as the concrete class of the Aggregate, you implement the SimpleList class, whose elements are objects of SimpleNode. That is, your SimpleList class implements the Aggregate interface as follows.

```
class SimpleList implements Aggregate{
    private SimpleNode nil;

    …

    public Iterator forwardIterator(){
        return new SimpleListForwardIterator(this);
    }
    public Iterator backwardIterator(){
        return new SimpleListBackwardIterator(this);
    }
}
```

That is, your SimpleList class must implement the forwardIterator and backwardIterator methods that creates and returns an Iterator object.

SimpleListForwardIterator, a concrete class of Iterator that accesses elements in order from the front, can be implemented as follows

```
public class SimpleListForwardIterator implements Iterator{
   private SimpleList simpleList;
   private SimpleNode cur;

   public SimpleListForwardIterator(SimpleList simpleList){
      this.simpleList = simpleList;
      cur = simpleList.getNil().getNext();
   }

   public boolean hasNext(){
      return cur != simpleList.getNil();
   }

   public Object next(){
      Object target = cur;
      cur = cur.getNext();
      return target;
   }
}
```

An objects generated from the concrete class of Iterator receives the target aggregate (in this case, a SimpleList object) and is created with a reference to it and the information needed to operate on it (in this case, *cur* representing the current position).

Your task is to complete the SimpleList and SimpleListBackwardIterator classes required for this application. The IteratorApplication class shown below is the application that tests your program.

```
import java.util.Scanner;

class IteratorApplication{
   public static void main(String[] args){
      new IteratorApplication().run();
   }

   void run(){
      Scanner sc = new Scanner(System.in);

      SimpleList list = new SimpleList();
      int Q = sc.nextInt();
      for ( int i = 0; i < Q; i++ ){
         String command = sc.next();
         int key = sc.nextInt();
         if ( command.equals("insert") ){
            list.insert(key);
         } else if ( command.equals("delete") ){
            list.delete(key);
         }
      }

      trace(list.forwardIterator());
      trace(list.backwardIterator());
   }

   private void trace(Iterator it){
      while( it.hasNext() ){
         SimpleNode node = (SimpleNode)it.next();
         System.out.print(node.getKey() + " ");
      }
      System.out.println();
   }
}
```

Check the following input/output examples for verification.

Sample Input	Sample Output
`13`	`31 29 23 13 11 7 5 2`
`insert 2`	`2 5 7 11 13 23 29 31`
`insert 3`	
`insert 5`	
`insert 7`	
`insert 11`	
`delete 3`	
`insert 13`	
`insert 17`	
`insert 23`	
`delete 17`	
`insert 29`	
`insert 31`	
`delete 17`	
`16`	`54 16 28 15 11 7`
`insert 8`	`7 11 15 28 16 54`
`insert 1`	
`delete 8`	
`insert 10`	
`insert 9`	
`insert 7`	
`insert 11`	
`delete 1`	
`delete 9`	
`insert 14`	
`insert 15`	
`insert 28`	
`delete 14`	
`insert 16`	
`insert 54`	
`delete 10`	

This design approach is called the Iterator pattern. The pattern features a flexible design that separates the class of aggregates from the class of algorithms for traversing those aggregates. In fact, it is not reasonable to implement all possible traversal algorithms in the SimpleList class. Also, the party traversing with Iterator does not need to be aware of how the aggregate is implemented (whether it is a list or an array). For example, if the aggregate is changed from the list implemented here to an array, the essential parts of the IteratorApplication would not need to be changed.

Submission Files	Types
`SimpleNode.java`	`Java Class`
`SimpleList.java`	`Java Class`
`Iterator.java`	`Java Interface`
`Aggregate.java`	`Java Interface`
`SimpleListForwardIterator.java`	`Java Class`
`SimpleListBackwardIterator.java`	`Java Class`
`IteratorApplication.java`	`Java Class`

Summary

This exercise deepened the understanding of the concept of polymorphism through a variety of applications. In particular, switching strategies using an interface such as Comparator is one of the most commonly used design patterns (this is called Strategy Pattern).