Ho Chi Minh City International University

HCMIU CVIP

Nguyen Qui Vinh Quang

2021-04-09

List (1)

ArrayList.java

209 lines

```
package list;
public class MyArrayList<E> implements java.util.List<E> {
    private static enum MoveType {
        NEXT, PREV
    }
    private static int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    private E[] elements;
    private int size;
    //Constructor:
    public MyArrayList(int capacity) throws
         IllegalArgumentException {
        if ((capacity < 0) || (capacity > MAX_CAPACITY)) {
            String message = String.format("Invalid capacity
                 (=%d)", capacity);
            throw new IllegalArgumentException(message);
        }

        this.elements = (E[]) new Object[capacity];
        this.size = 0;
    }
    public MyArrayList() throws IllegalArgumentException {
  this(10);}
    //Utitilies
    private void checkValidIndex(int index, int min, int max) {
        if ((index < min) || (index > max)) {
            String message = String.format("Invalid index (=%d)
                 ", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    @Override
    public int size() {return this.size;}
    @Override
    public boolean isEmpty() {return this.size == 0;}
    @Override
    public boolean contains(Object o) {
        boolean found = false;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) {
                found = true; break;
            }
        }
        return found;
    }
    @Override
    public Iterator<E> iterator() {return new MyIterator();}
    @Override
    public Object[] toArray() {return Arrays.copyOf(elements,
         size);}
    @Override
    public boolean add(E e) {
        if (e == null) throw new NullPointerException("Can't
             add null pointer");
        checkCapacity(this.size + 1);
        this.elements[this.size++] = e;
        return true;
    }
    private void checkCapacity(int minCapacity) {
        if ((minCapacity < 0) || (minCapacity > MAX_CAPACITY))
            throw new OutOfMemoryError("Not enough memory to
                 store the array");
        //newCapacity maybe a negative becuase of the overflow
        if (minCapacity >= this.elements.length) {
            //grow: oldCapacity >> 1 (= oldCapacity/2)
            int oldCapacity = this.elements.length;
            int newCapacity = oldCapacity + (oldCapacity / 2);
            if (newCapacity < 0) newCapacity = MAX_CAPACITY;
            this.elements = Arrays.copyOf(this.elements,
                 newCapacity);}
    }
    @Override
    public boolean remove(Object o) {
        int index = indexOf(o);
        if (index < 0) return false;
        remove(index);
        return true;
    }

    @Override
    public void clear() {
        while (isEmpty() == false) remove(0);
    }
    @Override
    public E get(int index) {
        checkValidIndex(index, 0, size - 1);
        return this.elements[index];
    }
    @Override
    public E set(int index, E element) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        this.elements[index] = element;
        return oldElement;
    }
    @Override
    public void add(int index, E element) {
        checkValidIndex(index, 0, size);
        if (element == null)
            throw new NullPointerException("Can not add null
                 pointer");
        checkCapacity(this.size + 1);
        int copyCount = (this.size - 1) - index + 1;
        System.arraycopy(this.elements, index, this.elements,
             index + 1, copyCount);
        this.elements[index] = element; this.size++;
    }
    @Override
    public E remove(int index) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        int copyCount = (this.size - 1) - (index + 1) + 1;
        System.arraycopy(this.elements, index + 1, this.
             elements, index,
                 copyCount);
        this.size--;
        return oldElement;
    }
    @Override
    public int indexOf(Object o) {
        int foundIdx = -1;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) { //== not
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public int lastIndexOf(Object o) {
        int foundIdx = -1;
        for (int idx = this.size - 1; idx >= 0; idx--) {
            if (this.elements[idx].equals(o)) {
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public ListIterator<E> listIterator() {return new
         MyListIterator();}
    @Override
    public ListIterator<E> listIterator(int index) {
return new MyListIterator(index);}
    @Override
    public String toString() {
        String desc = "[";
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            E e = it.next();
            desc += String.format("%s,", e);
        }
        if (desc.endsWith(","))
            desc = desc.substring(0, desc.length() - 1);
        return desc + "]";
    }
    //Definition of Inner Class
    public class MyIterator implements Iterator<E> {
        int cursor = 0;
        MoveType moveType = MoveType.NEXT;
        boolean afterMove = false;
        @Override
        public boolean hasNext() {
            return this.cursor != MyArrayList.this.size; }
        @Override
        //Move cursor to next + return preivous element
        public E next() {
            cursor += 1;
            moveType = MoveType.NEXT;
            afterMove = true;
            return MyArrayList.this.elements[cursor - 1];
        }
        @Override
        public void remove() {
            if (!afterMove) return;
            MyArrayList.this.remove(cursor - 1);
            cursor -= 1;
            afterMove = false;
        }
    }//End of MyIterator
    public class MyListIterator extends MyIterator implements
         ListIterator<E> {
        public MyListIterator(int index) {cursor = index;}
        public MyListIterator() {}
        @Override
        public boolean hasPrevious() {return this.cursor != 0;}
        @Override
        public void remove() {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) super.remove();
             else {
                MyArrayList.this.remove(cursor);
                afterMove = false;
```

```
            }}
        @Override
        public E previous() {
            cursor -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return MyArrayList.this.elements[cursor];
        }
        @Override
        public int nextIndex() { return cursor;}
        @Override
        public int previousIndex() { return cursor - 1;}
        @Override
        public void set(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) MyArrayList.this.set
                 (cursor - 1, e);
            else MyArrayList.this.set(cursor, e);
        }
        @Override
        public void add(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) {
                MyArrayList.this.add(cursor - 1, e);
            } else MyArrayList.this.add(cursor, e);
            cursor += 1;
            afterMove = false;
        }
    }
}
```

SLinkedList.java — 208 lines

```
package list;
public class MyArrayList<E> implements java.util.List<E> {
    private static enum MoveType {
        NEXT, PREV
    }
    private static int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    private E[] elements;
    private int size;
    //Constructor:
    public MyArrayList(int capacity) throws
         IllegalArgumentException {
        if ((capacity < 0) || (capacity > MAX_CAPACITY)) {
            String message = String.format("Invalid capacity
                 (=%d)", capacity);
            throw new IllegalArgumentException(message);
        }

        this.elements = (E[]) new Object[capacity];
        this.size = 0;
    }
    public MyArrayList() throws IllegalArgumentException {
  this(10);}
    //Utitilies
    private void checkValidIndex(int index, int min, int max) {
        if ((index < min) || (index > max)) {
            String message = String.format("Invalid index (=%d)
                 ", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    @Override
    public int size() {return this.size;}
    @Override
    public boolean isEmpty() {return this.size == 0;}
    @Override
    public boolean contains(Object o) {
        boolean found = false;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) {
                found = true; break;
            }
        }
        return found;
    }
    @Override
    public Iterator<E> iterator() {return new MyIterator();}
    @Override
    public Object[] toArray() {return Arrays.copyOf(elements,
         size);}
    @Override
    public boolean add(E e) {
        if (e == null) throw new NullPointerException("Can't
             add null pointer");
        checkCapacity(this.size + 1);
        this.elements[this.size++] = e;
        return true;
    }
    private void checkCapacity(int minCapacity) {
        if ((minCapacity < 0) || (minCapacity > MAX_CAPACITY))
            throw new OutOfMemoryError("Not enough memory to
                 store the array");
        //newCapacity maybe a negative becuase of the overflow
        if (minCapacity >= this.elements.length) {
            //grow: oldCapacity >> 1 (= oldCapacity/2)
            int oldCapacity = this.elements.length;
            int newCapacity = oldCapacity + (oldCapacity / 2);
            if (newCapacity < 0) newCapacity = MAX_CAPACITY;
            this.elements = Arrays.copyOf(this.elements,
                 newCapacity);}
    }
    @Override
    public boolean remove(Object o) {
        int index = indexOf(o);
        if (index < 0) return false;
        remove(index);
        return true;
    }
    @Override
    public void clear() {
        while (isEmpty() == false) remove(0);
    }
    @Override
    public E get(int index) {
        checkValidIndex(index, 0, size - 1);
        return this.elements[index];
    }
    @Override
    public E set(int index, E element) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        this.elements[index] = element;
        return oldElement;
    }
    @Override
    public void add(int index, E element) {
        checkValidIndex(index, 0, size);
        if (element == null)
            throw new NullPointerException("Can not add null
                 pointer");
        checkCapacity(this.size + 1);
        int copyCount = (this.size - 1) - index + 1;
        System.arraycopy(this.elements, index, this.elements,
             index + 1, copyCount);
        this.elements[index] = element; this.size++;
    }
    @Override
    public E remove(int index) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        int copyCount = (this.size - 1) - (index + 1) + 1;
        System.arraycopy(this.elements, index + 1, this.
             elements, index,
                 copyCount);
        this.size--;
        return oldElement;
    }
    @Override
    public int indexOf(Object o) {
        int foundIdx = -1;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) { //== not
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public int lastIndexOf(Object o) {
        int foundIdx = -1;
        for (int idx = this.size - 1; idx >= 0; idx--) {
            if (this.elements[idx].equals(o)) {
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public ListIterator<E> listIterator() {return new
         MyListIterator();}
    @Override
    public ListIterator<E> listIterator(int index) {
return new MyListIterator(index);}
    @Override
    public String toString() {
        String desc = "[";
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            E e = it.next();
            desc += String.format("%s,", e);
        }
        if (desc.endsWith(","))
            desc = desc.substring(0, desc.length() - 1);
        return desc + "]";
    }
    //Definition of Inner Class
    public class MyIterator implements Iterator<E> {
        int cursor = 0;
        MoveType moveType = MoveType.NEXT;
        boolean afterMove = false;
        @Override
        public boolean hasNext() {
            return this.cursor != MyArrayList.this.size; }
        @Override
        //Move cursor to next + return preivous element
        public E next() {
            cursor += 1;
            moveType = MoveType.NEXT;
            afterMove = true;
            return MyArrayList.this.elements[cursor - 1];
        }
        @Override
        public void remove() {
            if (!afterMove) return;
            MyArrayList.this.remove(cursor - 1);
```

```
            cursor -= 1;
            afterMove = false;
        }
    }//End of MyIterator
    public class MyListIterator extends MyIterator implements
         ListIterator<E> {
        public MyListIterator(int index) {cursor = index;}
        public MyListIterator() {}
        @Override
        public boolean hasPrevious() {return this.cursor != 0;}
        @Override
        public void remove() {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) super.remove();
             else {
                MyArrayList.this.remove(cursor);
                afterMove = false;
            }}
        @Override
        public E previous() {
            cursor -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return MyArrayList.this.elements[cursor];
        }
        @Override
        public int nextIndex() { return cursor;}
        @Override
        public int previousIndex() { return cursor - 1;}
        @Override
        public void set(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) MyArrayList.this.set
                 (cursor - 1, e);
            else MyArrayList.this.set(cursor, e);
        }
        @Override
        public void add(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) {
                MyArrayList.this.add(cursor - 1, e);
            } else MyArrayList.this.add(cursor, e);
            cursor += 1;
            afterMove = false;
        }
    }
}
```

DLinkedList.java

390 lines

```
public class DLinkedList<E> implements List<E> {
    private static enum MoveType { NEXT, PREV }
    private Node<E> head;
    private Node<E> tail;
    private int size;
    /*
    Initialize the double-linked list as shown in Figure 32 in
         the tutorial.
    The following values should be initialized:
    * head, head.prev, head.next
    * tail, tail.prev, tail.next
    * size
    */
    public DLinkedList() {
        head = new Node<>(null, null, null);
        tail = new Node<>(null, null, null);
        head.next = tail;
        tail.prev = head;
        size = 0;
    }
    //checkValidIndex: assert that "index" inside of [min, max]
    private void checkValidIndex(int index, int min, int max) {
        if ((index < min) || (index > max)) {
            String message = String.format("Invalid index (=%d)
                 ", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    /*
    getNode(Object o): get node containing data (not applied
         for meta-nodes: ie., head and tail)
    * search and return the node containing object "o".
    * return "null" if not found
    */
    private Node<E> getNode(Object o) {
        Node<E> curNode = head.next;
        Node<E> foundNode = null;
        while (curNode != tail) {
            if (curNode.element.equals(o)) {
                foundNode = curNode;
                break;
            }
            curNode = curNode.next;
        }
        return foundNode;
    }
    //getNode(int index, int min, int max): get node containing
         either data or head/tail.
    private Node<E> getNode(int index, int min, int max) {
        checkValidIndex(index, min, max);
        Node<E> curNode;
        int curIndex;
        if ((size - index) < (size / 2)) {
            curNode = this.tail;
            curIndex = size;
            while (curIndex > index) {
                curIndex -= 1;
                curNode = curNode.prev;
            }
        } else {
            curNode = head;
            curIndex = -1;
            while (curIndex < index) {
                curIndex += 1;
                curNode = curNode.next;
            }
        }
        return curNode;
    }
    /*
    insertLnewR(Node<E> left, Node<E> newNode, Node<E> right):
    insert newNode to the double-linked list.
    after insertion: left<->newNode<->right
    */
    private void insertLnewR(Node<E> left, Node<E> newNode,
         Node<E> right) {
        Node<E> curNode = head;
        while (curNode.next != null) {
            if (curNode.next == right && curNode == left) {
                newNode.next = curNode.next;
                curNode.next.prev = newNode;
                newNode.prev = curNode;
                curNode.next = newNode;
                size += 1;
                return;
            }
            curNode = curNode.next;
        }
    }
    /*
    removeNode(Node<E> removedNode):
    remove "removedNode" from the double-linked list.
    */
    private void removeNode(Node<E> removedNode) {
        Node<E> preNode = removedNode.prev;
        Node<E> nextNode = removedNode.next;
        preNode.next = nextNode;
        nextNode.prev = preNode;
        size -= 1;
    }
    @Override
    public int size() { return this.size;}
    @Override
    public boolean isEmpty() { return size == 0;}
    @Override
    public boolean contains(Object o) {
        Node<E> curNode = head.next;
        while (curNode != tail) {
            if (curNode.element.equals(o)) return true;
            curNode = curNode.next;
        }
        return false;
    }
    @Override
    public Iterator<E> iterator() {return new FBWDIterator();}

    private void checkValidIndex(int index) {
        if ((index < 0) || (index >= size)) {
            String message = String.format("Invalid index (=%d)
                 ", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    @Override
    public boolean add(E e) {
        Node<E> newNode = new Node(null, null, e);
        Node<E> lastNode = tail;
        Node<E> preLastNode = tail.prev;
        preLastNode.next = newNode;
        lastNode.prev = newNode;
        newNode.prev = preLastNode;
        newNode.next = lastNode;
        size += 1 ;
        return true;
    }
    @Override
    public boolean remove(Object o) {
        Node<E> curNode = head.next;
        while (curNode != tail) {
            if (curNode.element.equals(o)) {
                removeNode(curNode);
                return true;
            }
            curNode = curNode.next;
        }
        return false;
    }
    @Override
    public void clear() {
        Node<E> curNode = head.next;
        Node<E> nextNode;
        while (curNode!= tail) {
            nextNode = curNode.next;
            removeNode(curNode);
            curNode = nextNode;
        }
        head.next = tail;
```

```
        tail.prev = head;
        size = 0;
    }
    @Override
    public E get(int index) {
        checkValidIndex(index);
        Node<E> curNode = head.next;
        for (int i = 1; i <= index; i++) {
            curNode = curNode.next;
        }
        return curNode.element;
    }
    @Override
    public E set(int index, E element) {
        checkValidIndex(index);
        Node<E> curNode = head.next;
        for (int i = 1; i <= index; i++) curNode = curNode.next
            ;
        curNode.element = element;
        return curNode.element;
    }
    @Override
    public void add(int index, E element) {
        if (index < 0 || index > size) {
            String message = String.format("IndexOutOfBounds
                (>=size)", index);
            throw new IndexOutOfBoundsException(message);
        }
        Node<E> curNode = head.next;
        int curIndex = 0;
        Node<E> newNode = new Node(null, null, element);
        if (index == 0) {
            insertLnewR(head, newNode, tail);
            return;
        }

        while (curNode != tail) {
            if (curIndex == index) {
                insertLnewR(curNode.prev, newNode, curNode);
                return;
            }
            curNode = curNode.next;
            curIndex++;
        }
    }
    @Override
    public E remove(int index) {
        checkValidIndex(index);
        Node<E> curNode = head.next;
        int curIndex = 0;
        while (curNode != tail) {
            if (curIndex == index) {
                Node<E> temp = curNode;
                removeNode(curNode);
                return temp.element;
            }
            curNode = curNode.next;
            curIndex += 1;
        }
        return null;
    }
    @Override
    public int indexOf(Object o) {
        Node<E> curNode = head.next;
        int foundIdx = -1;
        int index = 0;
        while (curNode != tail) {
            if (curNode.element.equals(o)) {
                foundIdx = index;
                break;
            }
            index += 1;
            curNode = curNode.next;
        }
        return foundIdx;
    }
    @Override
    public int lastIndexOf(Object o) {
        Node<E> curNode = head.next;
        int foundIdx = -1;
        int index = 0;
        while (curNode != tail) {
            if (curNode.element.equals(o)) foundIdx = index;
            index += 1;
            curNode = curNode.next;
        }
        return foundIdx;
    }
    @Override
    public ListIterator<E> listIterator() {
        return new FBWDIterator();
    }
    @Override
    public ListIterator<E> listIterator(int index) {
        return new FBWDIterator(index);
    }
    @Override
    public String toString() {
        String desc = "[";
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            E e = it.next();
            desc += String.format("%s,", e);
        }
        if (desc.endsWith(","))
            desc = desc.substring(0, desc.length() - 1) + ']';
        return desc;
    }
    //BEGIN OF INNER CLASSES
    private class Node<E> {
        E element;
        Node<E> next;
        Node<E> prev;
        Node(Node<E> prev, Node<E> next, E element) {
            this.prev = prev;
            this.next = next;
            this.element = element;
        }
        void update(Node<E> prev, Node<E> next, E element) {
            this.prev = prev;
            this.next = next;
            this.element = element;
        }
    }
    private class FWDIterator implements Iterator<E> {
        Node<E> curNode;
        boolean afterMove;
        FWDIterator() {
            curNode = DLinkedList.this.head.next;
            afterMove = false;
        }
        @Override
        public boolean hasNext() {
            return curNode != DLinkedList.this.tail;
        }
        @Override
        public E next() {
            E element = curNode.element;
            curNode = curNode.next;
            afterMove = true;
            return element;
        }
        @Override
        public void remove() {
            if (!afterMove) return;
            Node<E> prevNode = curNode.prev;
            removeNode(prevNode);
            afterMove = false;
        }
    }//End of MyIerator
    private class FBWDIterator extends FWDIterator implements
         ListIterator<E> {
        int curIndex;
        MoveType moveType;
        FBWDIterator() {
            super();
            curIndex = 0;
            moveType = MoveType.NEXT; //default
            afterMove = false;
        }
        FBWDIterator(int index) {
            moveType = MoveType.NEXT;
            if ((index < 0) || (index > DLinkedList.this.size))
                  {
                String message = String.format("Invalid index
                    (=%d)", index);
                throw new IndexOutOfBoundsException(message);
            }
            //Assign values to curIdex and curNode
            if ((DLinkedList.this.size - index) < DLinkedList.
                 this.size / 2) {
                curNode = DLinkedList.this.tail;
                curIndex = DLinkedList.this.size;
                while (curIndex > index) {
                    curIndex -= 1;
                    curNode = curNode.prev;
                }
            } else {
                curNode = DLinkedList.this.head;
                curIndex = -1;
                while (curIndex < index) {
                    curIndex += 1;
                    curNode = curNode.next;
                }
        }}
        @Override
        public E next() {
            E e = super.next();
            curIndex += 1;
            moveType = MoveType.NEXT;
            return e;
        }
        @Override
        public void remove() {
            if (!afterMove) return;
            Node<E> removedNode;
            if (moveType == MoveType.NEXT) {
                removedNode = curNode.prev;
            } else {
                removedNode = curNode;
                curNode = curNode.next;
            }
            removeNode(removedNode);
            afterMove = false;
        }
        @Override
        public boolean hasPrevious() {
```

```
            return curNode.prev != DLinkedList.this.head;
        }
        @Override
        public E previous() {
            curNode = curNode.prev;
            curIndex -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return curNode.element;
        }
        @Override
        public int nextIndex() {return this.curIndex;}
        @Override
        public int previousIndex() {return curIndex - 1;}
        @Override
        public void set(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) {
                Node<E> prevNode = curNode.prev;
                prevNode.element = e;
            } else
                curNode.element = e;
        }
        @Override
        public void add(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) {
                Node<E> prevNode = curNode.prev; //go to prev
                Node<E> newNode = new Node<>(null, null, e);
                insertLnewR(prevNode.prev, newNode, prevNode);
            } else {
                Node<E> newNode = new Node<>(null, null, e);
                insertLnewR(curNode.prev, newNode, curNode);
                curNode = curNode.prev; // to new node
            }
        }
    }//End of FBWDIterator
}//End of DLinkedList
```

sorting (2)

ISort.java

5 lines

```
package sorting;
import java.util.Comparator;
public interface ISort<E> {
    public void sort(E[] array, Comparator<E> comparator);
}
```

ShellSort.java

49 lines

```
package sorting;
public class ShellSort<E> implements ISort<E> {
    //Sort segement k:
    /* To call this method,
  int[] num_segment = {1, 3, 7}; shell_sort(list, num_segment);
       */
    private int[] num_segment;
    public ShellSort(int[] num_segment){
        this.num_segment = num_segment;
    }
    public static <E> void sortSegment(E[] array, int
         segment_idx, int num_segment,
            Comparator<E> comparator){
        int current;
        int walker;
        E temp;
        current = segment_idx + num_segment;
        while(current < array.length){
            temp = array[current];
            walker = current - num_segment;
            while((walker >= 0) &&  comparator.compare(temp,
                 array[walker]) < 0 ){
                array[walker + num_segment] = array[walker]; //
                     shift to right
                walker -= num_segment;
            }
            array[walker + num_segment] = temp;
            current += num_segment;
        }
    }
    public void sort(E[] array, Comparator<E> comparator){
        for(int k= num_segment.length - 1; k >= k > 0; k--){
            int nsegment = num_segment[k];
            for(int segment_idx = 0; segment_idx < k;
                 segment_idx++)
                ShellSort.sortSegment(array, segment_idx,
                     nsegment, comparator);
        }

    // Another shell sort method, shell sort with segments
         divided / 2 after each itearations
    @Override
    public void sortSegmentMethod2(E[] array, Comparator<E>
         comparator) {
        int n = array.length;
        int interval = n;
        while ((interval /= 2) > 0) {
            for(int i = interval, j; i < n; i++){
                E temp = array[i];
                for(j = i; j >= interval && comparator.compare(
                     temp, array[j - interval]) < 0; j -=
                     interval){
                    array[j] = array[j - interval];
                }
                array[j] = temp;
            }
        }
    }
}
```

BubbleSort.java

22 lines

```
package sorting;
import java.util.Comparator;
public class BubbleSort<E> implements ISort<E> {
    public static void sort(E arr[], Comparator<E> comparator){
        int current, walker;
        boolean flag;
        current = 0;
        flag = false;
        while((current < arr.length-1) && (flag == false)){
            walker = arr.length - 1; //start from the last and
                 backward
            flag = true; //for testing if the input already in
                 ascending order
            while(walker > current){
                if(comparator.compare(arr[walker],arr[walker
                     -1]) < 0){
                    flag = false;
        E temp = arr[walker]; arr[walker] = arr[walker-1]; arr[
             walker-1] = temp;
                }
                walker -= 1;
            }
            current += 1;
        }
    }
}
```

StraightInsertionSort.java

23 lines

```
package sorting;
public class StraightInsertionSort<E> implements ISort<E> {
    /* Method: sort
    Objective: Sort an array of arbitrary type according to a
         comparator:
    Computational complexity: O(n^2)
        + Traversing through the array takes O(n), and in each
             iteration,
        we compare the current element with the elements after
             it in the array,
        which takes at most O(n) for each iteration. Thus, O(n^
             2). */
    @Override
    public void sort(E[] array, Comparator<E> comparator) {
        int current, walker;
        E temp; current = 1;
        while(current < array.length) {
            temp = array[current];
            walker = current - 1;
            while((walker >= 0) && comparator.compare(temp,
                 array[walker]) < 0) {
                array[walker + 1] = array[walker]; //shift to
                     right
                walker -= 1;
            }
            array[walker + 1] = temp;
            current += 1;
        }
}}
```

StraightSelectionSort.java

31 lines

```
package sorting;
public class StraightSelectionSort < E > implements ISort < E >
     {
    Boolean isAsc = true;
    public StraightSelectionSort() {}
    public StraightSelectionSort(Boolean isAsc) {
        this.isAsc = isAsc;
    }
    @Override
    public void sort(E[] array, Comparator < E > comparator) {
        int current, smallest, walker;
        current = 0;
        while (current < array.length - 1) {
            smallest = current;
            walker = current + 1;
            int factor = 1;
            if (!isAsc) factor = -1;
            while (walker < array.length) {
                if (comparator.compare(array[walker], array[
                     smallest]) * factor < 0)
                    smallest = walker;
                walker++;
            }
            if (smallest != current) {
                // swap:
                E temp = array[smallest];
                array[smallest] = array[current];
                array[current] = temp;
            }
            current++;
        }
    }
}
```

```
}
```

Searching (3)

BinarySearh.java

15 lines

```
import java.util.Comparator;
class BinarySearh<E> {
  public int search(E[] arr,int n,  Comparator<E> comparator, E
       val) {
      int l = 1, r = n;
      while (l <= r) {
            int mid = (l + r) >> 2;
            if (comparator.compare(arr[mid], val) == 0)
      return mid;
            if (comparator.compare(arr[mid], val) < 0)
      l = mid + 1;
            else r = mid - 1;
        }
      return -1;
  }
}
```

TernarySearch.java

39 lines

```
class TernarySearh {
  public static void main(String[] args) {
    System.out.println("Hello Codiva");

  }
  public double f(double x) {
    // implement function
    return 0;
  }
  public double golden_section_search(double a, double b) {
    double r = (Math.sqrt(5) - 1) / 2, eps = 0.01;
    double x1 = b - r * (b - a), x2 = a + r * (b - a);
    double f1 = f(x1), f2 = f(x2);
    while (Math.abs(b - a) > eps) {
        // cout << x1 << " " << f1 << "\n";
        if (f1 > f2) {
            b = x2; x2 = x1; f2 = f1;
            x1 = b - r * (b - a);
            f1 = f(x1);
        } else {
            a = x1; x1 = x2; f1 = f2;
            x2 = a + r * (b - a);
            f2 = f(x2);
        }
    }
    return a;
  }
  // vnoi ternary search
  double max_f(double left, double right) {
  int N_ITER = 100;
  for (int i = 0; i < N_ITER; i++) {
    double x1 = left + (right - left) / 3.0;
    double x2 = right - (right - left) / 3.0;
    if (f(x1) > f(x2)) right = x2;
    else left = x1;
  }
  return f(left);
}
}
```

Stuff (4)

NoteStupid.java

20 lines

```
import java.util.Comparator;
public class NoteStupid.java {
    public String toString() {
  // 6 la so ki tu hien thi, 2 la so sau dau thap phan
        return String.format("P(%6.2f, %6.2f)", this.x, this.y)
            ;
    }
  //Generate trong doan min max
    public static Double[] generate(int N, double min, double
        max){
        Random generator = new Random();
        Double[] list = new Double[N];
        for(int idx=0; idx < N; idx++){
            double x = min + generator.nextDouble()*(max - min)
                ;
      //nextInt: Integer
        }
        return list;
    }
    Integer[] arr = new Integer[]{ 1,2,3}; // create defaut
        array
    sortAlg.sort(points, new PointComparator(true)); // cach
        goi ham sort
    Math.sqrt(); // cac thao tac toan dung Math
}
```

PointComparator.java

20 lines

```
import java.util.Comparator;
public class PointComparator implements Comparator<Point2D> {
    enum sortTyoe {
        ASC(1), DES(-1);
        private int constant;
        sortTyoe(int constant) {this.constant = constant;}
        int getConstant() {return this.constant;}
    }
    private sortTyoe direction;
    public PointComparator(boolean isAscending) {
        if (isAscending) this.direction = sortTyoe.ASC;
        else this.direction = sortTyoe.DES;
    }
    @Override
    public int compare(Point2D o1, Point2D o2) {
        if (Math.abs(o1.getX() - o2.getX()) < 1e-7) return 0;
        else if (o1.getX() < o2.getX()) return -1 * direction.
            getConstant();
        else return 1 * direction.getConstant();
    }
}
```