```
//Array List

package MyList;

public class MyArrayList<T extends Comparable<T>> implements MyList<T> {

    private Object[] arr;
    private int length = 0;
    private int capacity = 10;

    public MyArrayList() {
        arr = new Object[capacity];
    }

    @Override
    public void add(T item) {
        if (length == capacity)
            increaseCapacity();

        arr[length++] = item;
    }

    @Override
    public void add(T item, int index) {
        if (index < 0 || index > length) {
            throw new IndexOutOfBoundsException("Index should be positive
and, less or equal to size");
        }
        if (length + 1 >= capacity) increaseCapacity();

        for (int i = length - 1; i >= index; i--){
            arr[i + 1] = arr[i];
        }
        arr[index] = item;
        length++;
    }

    @Override
    public boolean remove(T item) {
        int indexOfItem = indexOf(item);
        if (indexOfItem != -1) {
            remove(indexOfItem);
            return true;
        }
        return false;
    }

    @Override
    public T remove(int index) {
        if (index >= length || index < 0) {
            throw new IndexOutOfBoundsException("Index should be positive and
less than size");
        }
        T returnedItem = (T)arr[index];

        for (int i = index; i < length - 1; i++){
            arr[i] = arr[i + 1];
        }
        arr[length - 1] = null;
        length--;
        return returnedItem;
    }

    @Override
    public void clear() {
        for (Object o : arr) o = null;
        length = 0;
    }

    private void increaseCapacity() {
        capacity = 2 * capacity;
        Object[] old = arr;
        arr = new Object[capacity];

        for (int i = 0; i < old.length; i++)
            arr[i] = old[i];
    }

    @Override
    public T get(int index) {
        if (index < 0 || index > length) {
            throw new IndexOutOfBoundsException("Index should be positive and
less than size");
        }
        return (T)arr[index];
    }

    @Override
    public int indexOf(Object o) {
        for (int i = 0; i < length; i++){
            if (o.equals(arr[i])) return i;
        }
        return -1;
    }

    @Override
    public int lastIndexOf(Object o) {
        for (int i = length - 1; i >= 0; i--){
            if (o.equals(arr[i])) return i;
        }
        return -1;
    }

    @Override
    public void sort() {
        Comparable<T> first, second;
        for (int i = 0; i < length - 1; i++) {
            first = (Comparable<T>) arr[i];
            for (int j = i + 1; j < length; j++) {
                second = (Comparable<T>) arr[j];
                if (first.compareTo((T)second) > 0){
                    T temp = (T) arr[j];
                    arr[j] = arr[i];
                    arr[i] = temp;
                    first = second;
                }
            }
        }
    }

    @Override
    public int size() {
        return length;
    }
    @Override
    public boolean contains(Object o) {
        for (Object value : arr) {
            if (o.equals(value)) return true;
        }
        return false;
    }

    public void swap(int indexI, int indexJ){
        T temp = (T) arr[indexI];
        arr[indexI] = arr[indexJ];
        arr[indexJ] = temp;
    }

    public void printAll(){
        for (int i = 0; i < size(); ++i)  {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}



//MyHeap
import MyList.*;

public class MyHeap<T extends Comparable<T>> {
    private final MyArrayList<T> list;

    public MyHeap(){
        list = new MyArrayList<T>();
    }

    public T getMin(){
        return list.get(0);
    }

    public void add(T item){
        list.add(item);
        traverseUp(size() - 1);
    }


    public T removeRoot(){
        list.swap(0,size() - 1);
        T item = list.remove(size() - 1);
        heapify(0);
        return item;
    }

    private void heapify(int index) {
        int left = leftChildOf(index), right, minimum;
        if (left >= size()) return;
        minimum = left;
        right = left + 1;
        if (right < size()){
            if (list.get(left).compareTo(list.get(right)) > 0){
                minimum++;
            }
        }
        if (list.get(minimum).compareTo(list.get(index)) < 0){
            list.swap(index, minimum);
            heapify(minimum);
        }
        else return;
    }

    private void traverseUp(int index){
        int parent;
        while (index > 0){
            parent = parentOf(index);
            if(list.get(parent).compareTo(list.get(index)) > 0){
                list.swap(index, parent);
                index = parent;
            }
            else break;
        }
    }

    private int leftChildOf(int index) {
        return index * 2 + 1;
    }

    private int rightChildOf(int index){
        return 2 * (index + 1);
    }

    private int parentOf(int index){
        return (index - 1) / 2;
    }

    public void printAll(){
        list.printAll();
    }

    public int size(){
        return list.size();
    }

    public boolean isEmpty(){
        return size() == 0;
    }
}
```

MyQueue

```
import MyList.*;

public class MyQueue<T extends Comparable<T>> {
    private final MyList<T> list;

    public MyQueue() {
        list = new MyLinkedList<T>();
    }

    public T peek(){
        if (isEmpty()) {
            System.out.println("MyQueue is empty");
            return null;
        }
        return list.get(0);
    }

    public T enqueue(T item){
        list.add(item);
        return item;
    }

    public T dequeue(){
        if (isEmpty()) {
            System.out.println("MyQueue is empty");
            return null;
        }
        return list.remove(0);
    }

    public int size(){
        return list.size();
    }

    public boolean isEmpty(){
        return size() == 0;
    }
}
```

MyStack

```
import MyList.*;

public class MyStack<T extends Comparable<T>> {
    private final MyList<T> list;

    public MyStack() {
        list = new MyArrayList<T>();
    }

    public T push(T item){
        list.add(item);
        return item;
    }

    public T peek(){
        if (isEmpty()) {
            System.out.println("MyStack is empty");
            return null;
        }
        return list.get(size() - 1);
    }

    public T pop(){
        if (isEmpty()) {
            System.out.println("MyStack is empty");
            return null;
        }
        return list.remove(size() - 1);
    }

    public int size(){
        return list.size();
    }

    public boolean isEmpty(){
        return size() == 0;
    }
}
```