Programming and Data Structures
Review Questions (Solutions)

Part 1: Comprehension Questions

1. Suppose **statement2** may throw an exception in the following code:

```
try{
    statement1;
    statement2;
    statement3;
}
catch(Exception ex1){
    System.exit(0);
}
finally{
   statement4;
}
statement5;
```

Answer the following questions:

a. If no exception occurs, will **statement3**, **statement4** or **statement5** be executed? **statement3**, **statement4**, and **statement 5** are executed.

b. If an exception of type **Exception** is thrown, will **statement3**, **statement4** or **statement5** be executed? None of the three statements is executed, because the program exits from the catch block.

c. If an exception that is not of type **Exception** is thrown, will **statement3**, **statement4** or **statement5** be executed? Only **statement4** is executed.

2. Evaluate the following postfix expression using a stack. Show all the steps.

20 6 * 6 5 * 31 + 1 - / 9 - *

		5		31		1					
6		6	30	30	61	61	60		9		
20	120	120	120	120	120	120	120	2	2	-7	ERROR

3. Determine the exact number of iterations executed by the following code and the time complexity of the code using Big-O notation? Show your work.

```
for (int i = n; i > 0; --i)      // n iterations
  for (int j = n/20; j > 0; j /= 2) // (n/20) (n/20)/2^1, (n/20)/2^2,
                                    // (log_2(n/20) + 1) iterations
     for(int k = 1; k < n; k *= 3) // k=1, 3, 9, 27,..., k=3^i=n,
                                   // log_3n iterations
            int product = i * k + j * (k-2) + k;
```

Total number of iterations: n * ($\log_2$(n/20)+1) * $\log_3$(n)
Big-O notation: O(n logn logn)

4. Assume the following list of values `{18, 9, 32, 22, 75, 83, 3}` to be sorted using **Merge sort**. Show all the steps to sort the list.

```
{18, 9, 32, 22, 75, 83, 3}
{18, 9 , 32}      {22, 75, 83, 3}          - split
{18}  {9, 32}     {22, 75}     {83, 3}     - split
{18}  {9}  {32}   {22}  {75}   {83}  {3}   - split

{18}  {9, 32}  {22, 75}  {3, 83}           - merge
{9, 18, 32}    {3, 22, 75, 83}             - merge
{3, 9, 18,22, 32, 75, 83}                  - merge
```

5. Assume the following list of values `{16, 80, 22, 55, 64, 95, 25}` to be sorted using **quicksort**. Use a pivot as the first element of the list. Show all the steps to sort the list.

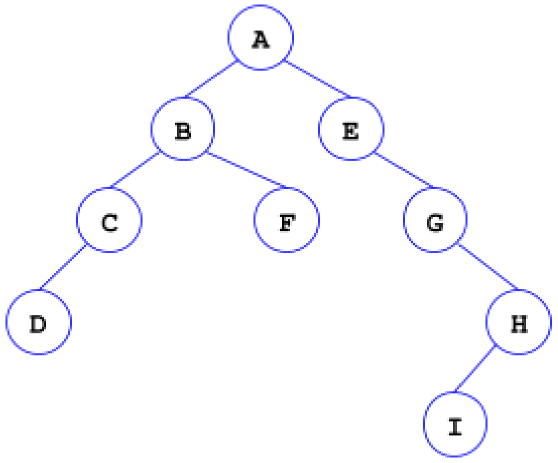
16	**80**	**22**	**55**	**64**	**95**	**25**	**-- pivot = 16**
16	**80**	**22**	**55**	**64**	**95**	**25**	**-- after partition()**
16	**80**	**22**	**55**	**64**	**95**	**25**	**-- pivot = 80**
16	**25**	**22**	**55**	**64**	**80**	**95**	**-- after partition()**
16	**25**	**22**	**55**	**64**	**80**	**95**	**-- pivot = 25**
16	**22**	**25**	**55**	**64**	**80**	**95**	**- - after partition()**
16	**22**	**25**	**55**	**64**	**80**	**95**	**-- pivot = 55**
16	**22**	**25**	**55**	**64**	**80**	**95**	**-- List sorted**

6. The nodes in a binary tree in pre-order and in-order sequences are as follows:

```
Pre-order: A B C D E F G H I
In-order: D C B A F E G I H
```

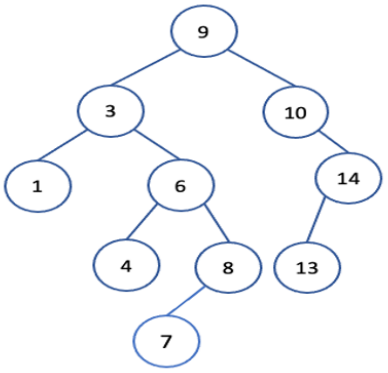
Draw the binary tree and write the post-order traversal of the tree.

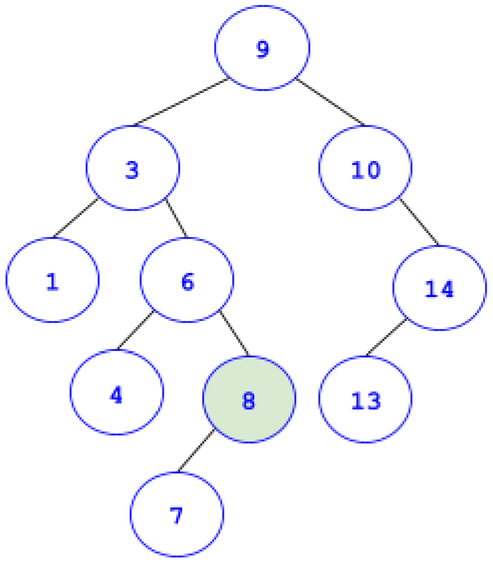
```
Pre-order: A B C D E F G H I  -- A is the root
In-order: D C B A F E G I H

LST(A) - preorder: B C D - B is the root
         inorder: D C B
         LST(B)- preorder: C D - C is the root
                 inorder: D C
                 LST(C)- D
                 RST(C)- null
         RST(B)- none


RST(A)- preorder: E F G H I - E is the root
        inorder: F E G I H
        LST(E)- F
        RST(E)- preorder: G H I - G is the root
                inorder: G I H
                LST(G)- none
                RST(G)- preorder:H I  - H is the root
                        inorder: I H
                        LST(H)- I
                        RST(H)- none

LST: Left SubTree, RST: Right SubTree
```

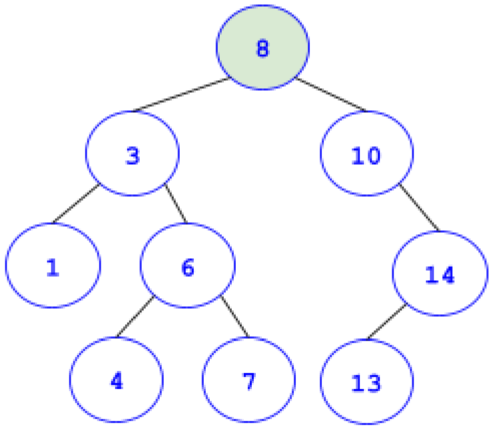


Postorder: D C B F I H G E A

7. Remove the node with value 9 from the following binary search tree. Write the in-order traversal of the BST after the deletion.







STEP 1: Find the largest node on the left of 9 (node with value 8)

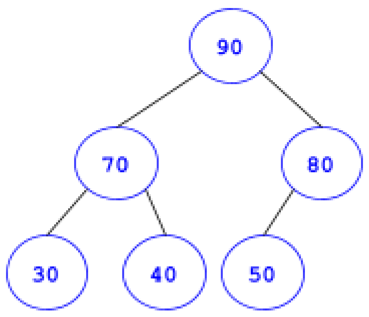
STEP 2: Replace the value 9 with the value 8

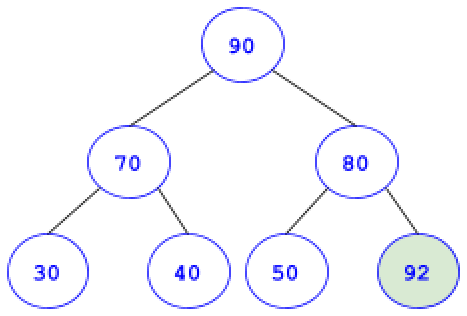
STEP 3: Remove the node with value 8

Inorder of the updated BST:

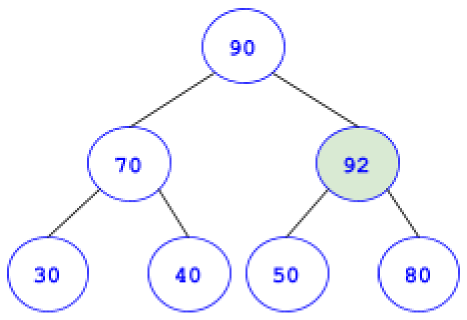
[1 3 4 6 7 8 10 13 14]

8. Redraw the following max heap after adding a node with value 92 and rebuilding the heap. Write the list of the nodes as stored in the heap's array list.

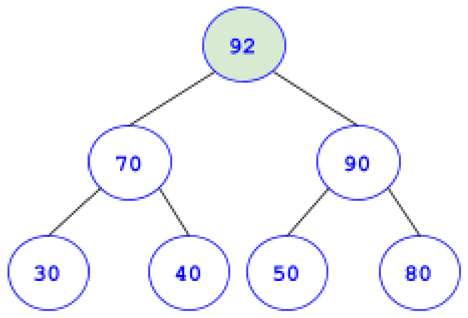




STEP 1: Add 92 at the end of the heap



STEP 2: swap 92 with 80



STEP 3: swap 92 with 90

The list of the heap nodes stored in the array list after adding node 92 is:

[92 70 90 30 40 50 80]

9. Assume a hash table has the initial size 4 and its load factor is 0.5, show the hash table after inserting the keys 34, 29, 53, 44, 120, 39, and 45, using linear probing.

```
put(34): hash(34) = 34%4 = 2, hashTable[2] = 34, size=1
put(29): hash(29) = 29%4 = 1, hashTable[1] = 29, size=2
put(53): size >= 2(4 * 0.5), rehash

put(29): hash(29) = 29%8 = 5, hashTable[5] = 29, size=2
put(34): hash(34) = 34%8 = 2, hashTable[2] = 34, size=1
put(53): hash(53) = 53%8 = 5, hashTable[6] = 53, size=3

put(44): hash(44) = 44%8 = 4, hashTable[4] = 44, size=4
put(120): size >= 4(8 * 0.5), rehash
put(34): hash(34) = 34%16 = 2, hashTable[2] = 34, size=1
put(44): hash(44) = 44%16 = 12, hashTable[12] = 44, size=2
put(29): hash(29) = 29%16 = 13, hashTable[13] = 29, size=3
put(53): hash(53) = 53%16 = 5, hashTable[5] = 53, size=4
put(120): hash(120) = 120%16 = 8, hashTable[8] = 120, size = 5
put(39): hash(39) = 39%16 = 7, hashTable[7] = 39, size=6
put(45): hash(45) = 45%16 = 13,hashTable[14] = 45, size=7
```

```
Initial table      After first rehash        After second rehash
                   (rehash when size=2)      (rehash when size=4)

capacity=4         capacity=8                capacity=16

0 null             0 null                    0  null
1 29               1 null                    1  null
2 34               2 34                      2  34
3 null             3 null                    3  null
                   4 44                      4  null
                   5 29                      5  53
                   6 53                      6  null
                   7 null                    7  39
                                             8  120
                                             9  null
                                             10 null
                                             11 null
                                             12 44
                                             13 29
                                             14 45
                                             15 null
```

10. Assume a hash table has the initial size 4 and its load factor is 0.9, show the hash table after inserting the keys 34, 29, 53, 44, 120, 39, and 45, using separate chaining.

```
put(34): hash(34)  = 34%4 = 2, hashTable[2] = {34}, size=1
put(29): hash(29)  = 29%4 = 1, hashTable[1] = {29}, size=2
put(53): hash(53)  = 53%4 = 1, hashTable[1] = {29, 53}, size=3
put(44): hash(44)  = 44%4 = 0, hashTable[0] = {44]}, size=4
put(120): (size >= 3.6 (4 *0.9), rehash

put(44): hash(44)  = 44%8 = 4, hashTable[4] = {44}, size=1
put(29): hash(29)  = 29%8 = 5, hashTable[5] = {29}, size=2
put(53): hash(53)  = 53%8 = 5, hashTable[5] = {29, 53}, size=3
put(34): hash(34)  = 34%8 = 2, hashTbale[2] = {34}, size=4
put(120): hash(120)  = 120%8 = 0, hashTable[0] = {120}, size=5
put(39): hash(39)  = 39%8 = 7, hashTable[7] = {39}, size=6
put(45): hash(45)  = 45%8 = 5, hashTable[5] = {29, 53, 45}, size=7
Initial table      After one rehash
                   (rehash when size = 4)

capacity=4         capacity=8

0 →  [44]          0 →[120]

1 →  [29, 53]      1 null

2 →  [34]          2 →[34]

3     null         3 null
```

```
4 →[44]
5 →[29, 53, 45]
6 null
7 →[39]
```

→ reference (pointer) to a linked list

Part 2: Programming Questions

1. Write a recursive method to compute the following series:

$$M(n) = 1 + \frac{1}{3} + \frac{2}{5} + \ldots + \frac{n}{2n+1}$$

Write a test program that displays *M(n)* for *n = 1, 2, . . ., 10*

```
public class Test{
     public static void main(String[] args) {
       int m = 10;
       for(int i=1; i<=m; i++)
         System.out.printf("M(%d) = %.2f\n", i, M(i));
     }
     public static double M(int n) {
          if(n==0)
             return 1;
          else
             return ((double)n/(2*n+1)) + M(n-1);
     }
}
```

2. Write a recursive method to find occurrences of a character c in a string s.

```
public static int occ(String s, char c) {
   return occ(s, c, 0); // start from index 0
}
// Helper recursive method
public static int occ(String s, char c, int index) {
   if (index >= s.length())
     return 0;
   else if(s.charAt(index) == c)
     return 1 + occ(s, c, index+1);
   else
     return occ(s, c, index+1);
}
```

3. Write a generic method **`removeDuplicates`** that removes duplicates from an **`ArrayList`**. The method returns a new **`ArrayList`** that contains the elements from the original list with no duplicates.

```
// The generic type E is not required to extend Comparable
// we do not need to order the elements in the list

public static <E> ArrayList<E>
                  removeDuplicates(ArrayList<E> list){
     ArrayList<E> newList = new ArrayList<>();
     for (int i=0; i<list.size(); i++) {
          int index = newList.indexOf(list.get(i));
          if (index == -1)
               newList.add(list.get(i));
     }
     return newList;
}
```

4. Write a generic method **`max`** that finds the maximum value in an **`ArrayList`.**

```
// The generic type E must extend type Comparable<E>
// finding the maximum requires comparing the elements in
// the list for order, not equality
public static <E extends Comparable<E>> E max(
                                      ArrayList<E> list) {
     if(size == 0)
       return null;
     else{
          E largest = list.get(0);
          for(int i=1; i<list.size(); i++) {
               if(largest.compareTo(list.get(i)) < 0)
                    largest = list.get(i);
          }
          return largest;
}
```

5. Add the methods **addAll(ArrayList<E>)**, **removeAll(ArrayList<E>)**, **retainAll(<ArrayList<E>)**, and **toArray()** to the class **ArrayList** and determine their time complexity.

[1, 2, 3].addAll([3, 4]) results in the first list equal to **[1, 2, 3, 3, 4]**
[1, 2, 3].removeAll([3, 4]) results in the first list equal to **[1, 2]**
[1, 2, 3].retainAll([3, 4]) results in the first list equal to **[3]**
[1, 2, 3].toArray() returns the array **{1, 2, 3}**

```
// Create the method indexOf() for linear search
// The method exists in the interface List which is
// implemented by the class ArrayList
public int indexOf(Object obj) { // O(n)
   for(int i=0; i<size; i++)
      if(elements[i].equals(obj))
         return i;
   return -1;
}
// Union of two array lists
public void addAll(ArrayBasedList<E> list) { // O(n)
   for(int i=0; i<list.size(); i++) {
      add(list.get(i));
   }
}
// Intersection of two array lists
public void retainAll(ArrayBasedList<E> list) { //O(n^2)
      for(int i=0; i<size; i++) {
      if(list.indexOf(elements[i]) == -1)
           remove(i);
      }
}
// Difference of two array lists
public void removeAll(ArrayBasedList<E> list){ // O(n^2)
      for(int i=0; i<size; i++) {
      if(list.indexOf(elements[i]) != -1)
           remove(i);
      }
}
// converts an array list into an array
public E[] toArray() { // O(n)
```

```
      E[] array = (E[]) new Object[size];
      for(int i=0; i<size; i++)
        array[i] = elements[i];
      return array;
}
```

6. Add the methods **`contains(E)`**, **`get(int index)`**, **`lastIndexOf(E value)`**, and **`set(int index, E value)`** to the class **`LinkedList`** and determine their time complexity.

```
public boolean contains(E item) { // O(n)
   Node current = head;
   while(current != null) {
     if(current.value.equals(item))
          return true;
     }
   return false;
}
public E get(int index) { // O(n)
     Node current = head;
     if (index < 0 || index >= size)
          throw new NoSuchElementException();
     for(int i=0; i<index; i++)
        current = current.next;
     return current.value;
}
public E set(int index, E newValue) { // O(n)
     Node current = head;
     if (index < 0 || index >= size)
       throw new NoSuchElementException();
     for(int i=0; i<index; i++)
       current = current.next;
     E old = current.value;
     current.value = newValue;
     return old;
}
```

```
public int lastIndexOf(E item) { // O(n^2)
      for(int i=size-1; i>=0; i--)
        if(get(i).equals(item))
            return i;
      return -1;
}
```

7. Add a method **`reverse()`** to the class **`LinkedList`** to print the elements of the list in reverse order (from the last element to the first element). Determine the time complexity of the method.

```
public void reverse() { // O(n^2)
     System.out.print("[");
     for(int i=size-1; i>=0; i--) { // O(n)
         System.out.print(get(i) + " ");// get() is O(n)
      }
      System.out.print("]");
}
```

8. Write a test program that stores 5 million integers in a linked list and test the time to traverse the list using an iterator vs. using the method **`get(index)`** from question 6.

```
LinkedList<Integer> llist = new LinkedList<>();
for(long i=0; i<5000000; i++)
   llist.add((int)(Math.random()*1000000));

      long time1 = System.nanoTime();
      for(long i=0; i<llist.size(); i++) {
        Integer n = llist.get(i);
      }
      long time2 = System.nanoTime();
      System.out.println("Time(get()):"+(time2-time1)+" ns");
      time1 = System.nanoTime();
      Iterator<Integer> lliter = llist.iterator();
      while(lliter.hasNext()) {
        Integer n = lliter.next();
      }
      time2 = System.nanoTime();
      System.out.println("Time(iterator):"+(time2-time1)+" ns");
```

9. The quicksort algorithm presented in class selects the first element in the list as the pivot. Revise it by selecting the median among the first, middle, and the last elements in the list. Compare the two quicksort algorithms on three arrays. The first array contains random integers, the second is sorted and the third is in reverse order.

```java
public static <E extends Comparable<E>>
       void quickSort(ArrayList<E> list) {
      quickSort(list, 0, list.size() - 1);
}

public static <E extends Comparable<E>>
  void quickSort(ArrayList<E> list, int first, int last) {
      if (last > first) {
        int pivotIndex = partition(list, first, last);
        quickSort(list, first, pivotIndex - 1);
        quickSort(list, pivotIndex + 1, last);
      }
}
public static <E extends Comparable<E>> int partition(
                ArrayList<E> list, int first, int last) {
      E pivot;
      int index, pivotIndex;
      pivotIndex = median(list, first, last);
      pivot = list.get(pivotIndex);
      swap(list, pivotIndex, first);
      pivotIndex = first;
      for (index = first + 1; index <= last; index++) {
           if (list.get(index).compareTo(pivot) < 0) {
                pivotIndex++;
                swap(list, pivotIndex, index);
           }
      }
      swap(list, first, pivotIndex);
      return pivotIndex;
}
public static <E extends Comparable<E>> int median(
              ArrayList<E> list, int first, int last) {
      E[] median = (E[]) new Object[3];
      int middle = (last+first)/2;
      median[0] = list.get(first);
      median[1] = list.get(middle);
      median[2] = list.get(last);
      java.util.Arrays.sort(median);
      if(median[1].compareTo(list.get(middle)) == 0)
           return middle;
      else if(median[1].compareTo(list.get(first)) == 0)
           return first;
      else
           return last;
}
public static <E extends Comparable<E>> void swap(
           ArrayList<E> list, int index1, int index2) {
      E temp = list.get(index1);
      list.set(index1, list.get(index2));
      list.set(index2, temp);
}
```

10. Add the method **`sort()`** to the class **`LinkedList`**. The method should sort the elements of the list using bubble sort. Determine the time complexity of the method.

```
/**
 * Method bubbleSort to sort the nodes of the linked list
 * @param comp Comparator object used to order the nodes
*/
public void bubbleSort(Comparator<E> comp) { // O(n^2)
   boolean sorted = false;
   for (int k=1; k < size && !sorted; k++) {
      sorted = true;
      Node node = head, previous = null;
      int max = size()-k; int index = 0;
      while(index < max) {
       previous = node; node = node.next; index++;
       if (comp.compare(previous.value, node.value) > 0) {
         // swap
         E temp = node.value;
         node.value = previous.value;
         previous.value = temp;
         sorted = false;
```

```
            }
          }
       }
   }
```

11. The heap data structure covered in class is a **`maxheap`** where every node is greater than its children. Modify the class to implement a **`min-heap`** where every node is less than its children. Use the **`min-heap`** to implement the priority queue data structure. Determine the time complexity of all the methods in the heap-based priority queue class. Write a program that offers one million integers in the priority queue and polls them from the queue. Display the average number of iterations of the methods **`offer()`** and **`poll()`**.

```java
import java.util.ArrayList;
// class MinHeap
public class MinHeap<E extends Comparable<E>> {
      private ArrayList<E> list;
      public MinHeap(){ // O(1)
            list = new ArrayList<>();
      }
      // added method getRoot() to implement peek() later
      public E getRoot() { // O(1)
        if (list.size() == 0)
           return null;
        else
           return list.get(0);
      }
      public void add(E item) { // O(log n)
            list.add(item);
            int currentIndex = list.size()-1;
            while(currentIndex > 0) {// O(log n)
                  int parentIndex = (currentIndex-1)/2;
                  E current = list.get(currentIndex);
                  E parent = list.get(parentIndex);
                  if(current.compareTo(parent) < 0) {
                        list.set(currentIndex, parent);
                        list.set(parentIndex, current);
```

```
                }
                else
                     break;
                currentIndex = parentIndex;
           }
      }
      public E remove() { // O(log n)
           if(list.size() == 0) return null;
           E removedItem = list.get(0);
           list.set(0, list.get(list.size()-1));
           list.remove(list.size()-1);
           int currentIndex = 0;
           while (currentIndex < list.size()) {
                int left = 2 * currentIndex + 1;
                int right = 2 * currentIndex + 2;
                //find the minimum of the two children
                if (left >= list.size())
                break;
                int minIndex = left;
                E min = list.get(minIndex);
                if (right < list.size())
                  if(min.compareTo(list.get(right)) > 0)
                          minIndex = right;
                 E current = list.get(currentIndex);
                 min = list.get(minIndex);
                 if(list.get(currentIndex).compareTo(min)>0){
                     list.set(minIndex, current);
                     list.set(currentIndex, min);
                     currentIndex = minIndex;
                }
                else
                     break;
           }
           return removedItem;
      }
      public int size() { // O(1)
           return list.size();
      }
```

```java
      public void clear() { // O(1)
            list.clear();
      }
      public boolean isEmpty() { // O(1)
            return list.isEmpty();
      }
      public String toString() { // O(n)
            return list.toString();
      }
}
// Class PriorityQueue using a MinHeap
public class PriorityQueue<E extends Comparable<E>> {
      private MinHeap<E> heap;
      public PriorityQueue() { // O(1)
          heap = new MinHeap<>();
      }
      public void offer(E item) { // O(log n)
          heap.add(item);
      }
      public E poll() { // O(log n)
         E value = heap.remove();
         return value;
      }
      public E peek() { // O(1)
         return heap.getRoot();
      }
      public String toString() { // O(n)
         return "Priority Queue: " + heap.toString();
      }
      public int size() { // O(1)
         return heap.size();
      }
      public void clear() { // O(1)
         heap.clear();
      }
      public boolean isEmpty() { // O(1)
         return heap.size() == 0;
      }
}
```

```
// Testing Heap based PriorityQueue
public class Test{
  public static void main(String[] args){
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      long startTime = System.nanoTime();
      for(int i=0; i<1000000; i++)
      pq.offer((int)(Math.random()*1000000));
      long endTime = System.nanoTime();
      System.out.println("Time (offer): " + (endTime -
startTime));
      startTime = System.nanoTime();
      for(int i=0; i<1000000; i++)
      pq.poll();
      endTime = System.nanoTime();
      System.out.println("Time (poll): " + (endTime -
startTime));
   }
}
```

12. Implement the **clone** and **equals** methods for the **BST** class. Two BST trees are equal if they contain the same nodes. The **clone** method returns a deep copy of the tree. Determine the time complexity of the two methods.

```
// Creating an iterator method for the BST to traverse
// the nodes in preorder-to be used by equals() and clone()
public Iterator<E> iterator() {
           return new BSTIterator();
}
private class BSTIterator implements Iterator<E>{
   int current=-1;
   ArrayList<E> preorderlist = new ArrayList<>();
   // Constructor to build an array list that contains
   // the nodes of the tree in preorder
   public BSTIterator(){
      preorderList(root, preorderlist);
      if(preorderlist.size() != 0)
         current = 0;
   }
   private void preorderList(TreeNode node, ArrayList<E> list){
```

```
            if (node !=null) {
                list.add(node.value);
                preorderList(node.left, list);
                preorderList(node.right, list);
            }
    }
    public boolean hasNext() {
       return (current>=0 && current<preorderlist.size());
    }
    public E next() {
       if (current < 0 || current >= preorderlist.size())
              throw new NoSuchElementException();
       E val = preorderlist.get(current);
       current++;
       return val;
    }
}
// Method equals() for BSTs
public boolean equals(Object o) {
       if (o instanceof BST) {
          BST<E> tree = (BST<E>) o;
          Iterator<E> iter1 = iterator();
          Iterator<E> iter2 = tree.iterator();
          if(size != tree.size())
             return false;
          while(iter1.hasNext() && iter2.hasNext()) {
             if(!iter1.next().equals(iter2.next()))
                return false;
          }
          return true;
       }
       return false;
}
// Method clone() for BST
public Object clone() {
       BST<E> tree = new BST<>();
       Iterator<E> iter = iterator();
       while(iter.hasNext()) {
            tree.add(iter.next());
```

```
     }
     return tree;
}
```

13. Add the following methods to the **BST** class.

```
// Iterative preorder traversal using a stack
// No recursion
public void iterativePreorder()

// Recursive search method
public boolean recursiveSearch(E item)
```

```
// Iterative preorder traversal
 public void iterativePreorder() {
      Stack<TreeNode> stack = new Stack<>();
      stack.push(root);
      TreeNode current;
      while(!stack.isEmpty()) {
           current = stack.pop();
           System.out.print(current.value + " ");
           if(current.right != null) {
                 stack.push(current.right);
           }
           if(current.left != null)
                 stack.push(current.left);
      }
 }

// recursive search
public boolean recursiveSearch(E item) {
     return recursiveSearch(root, item);
}
public boolean recursiveSearch(TreeNode node, E item) {
     if(node != null) {
          if(node.value.equals(item))
```

```
                return true;
            else if(node.value.compareTo(item) > 0)
                return recursiveSearch(node.left, item);
            else
                return recursiveSearch(node.right, item);
        }
        return false;
}
```

14. The class **HashMap** seen in class uses a **LinkedList** to store collisions. Replace **LinkedList** with **BST**. The generic type **K** must be a subtype of **Comparable**. Redefine **HashMap** and test it using the same test program from ALA 9.

```
public class HashMapEntry<K extends Comparable<K>, V>
                 implements Comparable<HashMapEntry<K,V>>{
      private K key;
      private V value;
      public HashMapEntry(K k, V v) {
            key = k;
            value = v;
      }
      public K getKey() {
            return key;
      }
      public V getValue() {
            return value;
      }
      public void setKey(K k) {
            key = k;
      }
      public void setValue(V v) {
            value=v;
      }
      public String toString() {
            return "(" + key + ", " + value + ")";
      }
      // Define compareTo to compare two HashMapEntry
      // objects using their keys
      public int compareTo(HashMapEntry<K, V> hme) {
```

```java
            return key.compareTo(hme.key);
      }
}

import java.util.ArrayList;
public class HashMap<K extends Comparable<K>, V> {
      private int size;
      private double loadFactor;
      private BST<HashMapEntry<K, V>>[] hashTable;
      // Constructors
      public HashMap() {
            this(100, 0.9);
      }
      public HashMap(int c) {
            this(c, 0.9);
      }
      public HashMap(int c, double lf) {
            hashTable = new BST[trimToPowerOf2(c)];
            loadFactor = lf;
            size = 0;
      }
      // private methods
      private int trimToPowerOf2(int c) {
            int capacity = 1;
            while (capacity < c)
                  capacity = capacity << 1; // * 2
            return capacity;
      }
      private int hash(int hashCode) {
            return hashCode & (hashTable.length - 1);
      }
      private void rehash() {
            ArrayList<HashMapEntry<K,V>> list = toList();
            hashTable = new BST[hashTable.length << 1];
            size = 0;
            for (HashMapEntry<K,V> entry : list)
                  put(entry.getKey(), entry.getValue());
      }
      // public interface
```

```
    public int size() {
          return size;
    }
    public void clear() {
          size = 0;
          for (int i = 0; i < hashTable.length; i++)
                if (hashTable[i] != null)
                      hashTable[i].clear();
    }
    public boolean isEmpty() {
          return (size == 0);
    }

    // search for key in the hash map
    public boolean containsKey(K key) {
          if (get(key) != null)
                return true;
          return false;
    }
    // returns the value of key if found, otherwise null
    public V get(K key) {
          int HTIIndex = hash(key.hashCode());
          iterations = 0;
          if (hashTable[HTIIndex] != null) {
           BST<HashMapEntry<K,V>> bst= hashTable[HTIIndex];
          // define E contains(E item) in the BST
            HashMapEntry<K,V> entry =
                bst.contains(new HashMapEntry(key, null));
            if(entry != null) {
                return entry.getValue();
            }
          }
          return null;
    }

    // remove a key if found
    public void remove(K key) {
      int HTIIndex = hash(key.hashCode());
      if (hashTable[HTIIndex] != null) {
          BST<HashMapEntry<K,V>> bst = hashTable[HTIIndex];
          bst.remove(new HashMapEntry(key, null));
      }
    }
    // adds a new key or modifies an existing key
    public V put(K key, V value) {
      if (get(key) != null) {
          int HTIndex = hash(key.hashCode());
          BST<HashMapEntry<K,V>> bst = hashTable[HTIIndex];
          HashMapEntry<K,V> entry = bst.contains(
                           new HashMapEntry(key, null));
          if(entry != null) {
                V old = entry.getValue();
                entry.setValue(value);
                return old;
          }
     }
     if (size >= hashTable.length * loadFactor)
           rehash();
     int HTIIndex = hash(key.hashCode());
          // create a new bst if bucket is empty
          if (hashTable[HTIIndex] == null) {
                hashTable[HTIIndex] = new BST<>();
          }
          hashTable[HTIndex].add(
                      new HashMapEntry<>(key, value));
          size++;
          return value;
    }
    // returns the elements of the hash map as a list
    public ArrayList<HashMapEntry<K,V>> toList() {
          ArrayList<HashMapEntry<K,V>> list =
                                new ArrayList<>();
          for (int i = 0; i < hashTable.length; i++) {
            if (hashTable[i] != null) {
                BST<HashMapEntry<K,V>> bst = hashTable[i];
          // Use the method iterator() for BST
          // question 22
                Iterator<HashMapEntry<K,V>> bstIter =
                                    bst.iterator();
                while (bstIter.hasNext())
                     list.add(bstIter.next());
                }
           }
           return list;
      }
      // returns the elements of the hash map as a string
      public String toString() {
         String out = "[";
         for (int i = 0; i < hashTable.length; i++) {
           if (hashTable[i] != null) {
                BST<HashMapEntry<K,V>> bst = hashTable[i];
                Iterator<HashMapEntry<K,V>> bstIter =
                                      bst.iterator();
                while (bstIter.hasNext())
                     out += bstIter.next().toString();
                 out += "\n";
            }
           }
           out += "]";
           return out;
      }
      public int collisions() {
           int max = 0;
           for(int i=0; i<hashTable.length; i++) {
                if(hashTable[i] != null) {
                     if(hashTable[i].size() > max)
                          max = hashTable[i].size();
                }
           }
           return max;
      }
}
```

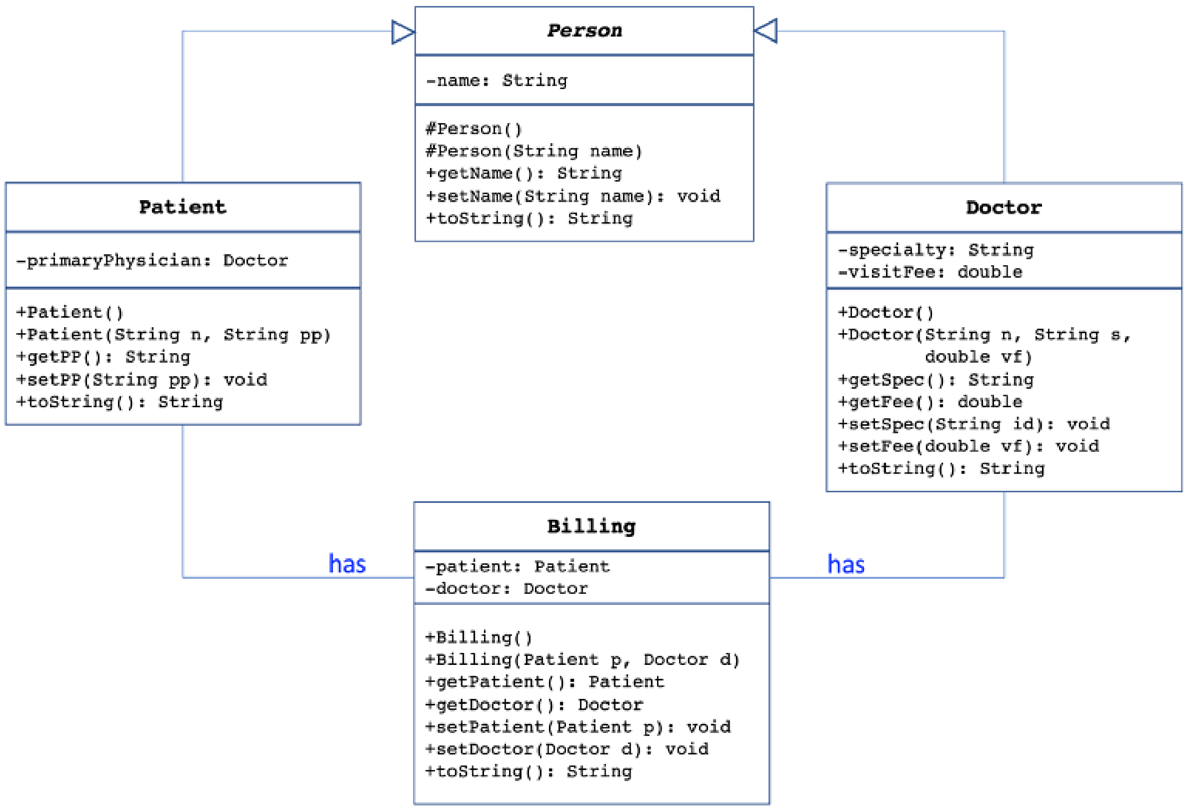
15. Add the following methods to the class **`HashMap`**:

a. **`public ArrayList<K> keys()`**: returns an array list with all the keys in the hash table.

b. **`public ArrayList<V> values()`**: returns an array list with all the values in the hash table.

```
public ArrayList<K> keys() {
      ArrayList<K> list = new ArrayList<>();
      for (int i = 0; i < hashTable.length; i++) {
        if (hashTable[i] != null) {
         LinkedList<HashMapEntry<K,V>> bucket=hashTable[i];
         for (HashMapEntry<K,V> entry : bucket)
           list.add(entry.getKey());
        }
      }
       return list;
}
public ArrayList<V> values() {
      ArrayList<V> list = new ArrayList<>();
      for (int i = 0; i < hashTable.length; i++) {
        if (hashTable[i] != null) {
         LinkedList<HashMapEntry<K,V>> bucket=hashTable[i];
         for (HashMapEntry<K,V> entry : bucket)
           list.add(entry.getValue());
        }
      }
      return list;
}
```

16. Implement the hierarchy of classes shown below. Write a program that reads the files `patients.txt`, `doctors.txt`, and `billings.txt`, and prints out the total income from the **`Billing`** records, the highest total income of doctors, and the highest total spending of patients. Note that **`Person`** is an abstract class.



```java
// Class Person
public abstract class Person {
   private String name;
   protected Person() { name =""; }
   protected Person(String name) { this.name = name; }
   public String getName() { return name; }
   public void setName(String n) { name = n; }
   public String toString() { return name; }
   public boolean equals(Object o) {
      if(o instance of Person){
      Person p = (Person) o;
      return p.name.equals(name)
      }
      return false;
   }
}
```

```
// Class Patient
public class Patient extends Person{
      private Doctor primaryPhysician;
     public Patient() {
      super();
      primaryPhysician = null;
     }
     public Patient(String n, Doctor d) {
      super(n);
      primaryPhysician = d;
     }
     public Doctor getPP() {
      return primaryPhysician;
     }
     public void setPP(Doctor d) {
      primaryPhysician = d;
     }
     public String toString() {
      return super.toString() + " " +
                 primaryPhysician.getName();
     }
}

// Class Doctor
public class Doctor extends Person{
   private String specialty;
   private double visitFee;
   public Doctor() {
      super();
      specialty=""; visitFee = 0.0;
   }
   public Doctor(String n, String s, double f) {
      super(n);
      specialty = s; visitFee = f;
   }
   public String getSpec() { return specialty;}
   public double getFee() { return visitFee;}
   public void setSpec(String s)  {specialty = s;}
   public void setFee(double f) { visitFee = f;}
   public String toString() {
     return super.toString() + " " + specialty + " " +
                                            visitFee;
   }
}
// Class Billing
public class Billing {
      private Patient patient;
      private Doctor doctor;
      public Billing() {
      patient = null; doctor = null;
      }
      public Billing(Patient p, Doctor d) {
      patient = p; doctor = d;
      }
      public Patient getPatient() { return patient; }
      public Doctor getDoctor() { return doctor; }
      public void setPatient(Patient p) { patient = p;}
      public void setDoctor(Doctor d) { doctor=d; }
      public String toString() {
      return "Patient: " + patient.toString() +
                      ", Doctor: " + doctor.toString();
      }
}

// Test program
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;
public class Test {
  // Main method
  public static void main(String[] args) {
   ArrayList<Patient> patientList = new ArrayList<>();
   ArrayList<Doctor> doctorList = new ArrayList<>();
   ArrayList<Billing> billingList = new ArrayList<>();
   readDoctorFile("doctors.txt", doctorList);
   readPatientFile("patients.txt",
                      patientList, doctorList);
   readBillingFile("billings.txt",  billingList,
                patientList, doctorList);
   double total = 0;
   for(int i=0; i<billingList.size(); i++)
       total += billingList.get(i).getDoctor().getFee();
   System.out.printf("Total Billing Amount: %.2f$\n",
                        total);
   doctorIncome(doctorList, billingList);
   patientSpending(patientList, billingList);
}
public static void doctorIncome(ArrayList<Doctor> dList,
                           ArrayList<Billing> bList) {
   double max = 0;
   int maxIndex = 0;
   ArrayList<Double> doctorIncome = new ArrayList<>();
   for(int i=0; i<dList.size(); i++)
      doctorIncome.add(0.0);
   for(int i=0; i<dList.size(); i++) {
     for(int j=0; j<bList.size(); j++) {
      if(bList.get(j).getDoctor().equals(dList.get(i))) {
       doctorIncome.set(i,
            doctorIncome.get(i)+dList.get(i).getFee());
      }
    }
  }
  for(int i=0; i<doctorIncome.size(); i++) {
      if(doctorIncome.get(i) > max) {
         maxIndex = i;
         max = doctorIncome.get(i);
      }
  }
  System.out.printf(
      "Doctor with the maximum income: %-15s ($%.2f)\n",
                dList.get(maxIndex).getName(), max);
}
public static void patientSpending(
                      ArrayList<Patient> pList,
                      ArrayList<Billing> bList) {
      double max = 0;
      int maxIndex = 0;
      ArrayList<Double> pSpending = new ArrayList<>();
      for(int i=0; i<pList.size(); i++)
         pSpending.add(0.0);
      for(int i=0; i<pList.size(); i++) {
        for(int j=0; j<bList.size(); j++) {
         if(bList.get(j).getPatient().equals(pList.get(i))){
           pSpending.set(i,pSpending.get(i) +
                      bList.get(j).getDoctor().getFee());
         }
        }
      }
      for(int i=0; i<pSpending.size(); i++) {
         if(pSpending.get(i) > max) {
            maxIndex = i;
            max = pSpending.get(i);
         }
      }
      System.out.printf(
      "Patient with the maximum spending: %-15s ($%.2f)\n",
                      pList.get(maxIndex).getName(), max);
}

public static void readPatientFile(String filename,
                           ArrayList<Patient> list,
                           ArrayList<Doctor> dList) {
    Scanner readFile = null;
    File file = new File(filename);
    try {
      readFile = new Scanner(file);
    }
    catch(FileNotFoundException e) {
      System.out.println("File not found");
      System.exit(0);
    }
    while (readFile.hasNext()) {
      String p = readFile.next();
      String dname = readFile.next();
      Doctor d = new Doctor(dname, "", 0);
      int index = dList.indexOf(d);
      if(index != -1) {
        list.add(new Patient(p, dList.get(index)));
      }
   }
   readFile.close();
}
public static void readDoctorFile(String filename,
                                  ArrayList<Doctor> list) {
   Scanner readFile = null;
   File file = new File(filename);
   try {
      readFile = new Scanner(file);
   }
   catch(FileNotFoundException e) {
      System.out.println("File not found");
      System.exit(0);
   }
   while (readFile.hasNext()) {
      Doctor d = new Doctor(readFile.next(),
                            readFile.next(),
                            readFile.nextDouble());
      list.add(d);
   }
   readFile.close();
}
public static void readBillingFile(String filename,
                                   ArrayList<Billing> list,
                                  ArrayList<Patient> pList,
                                  ArrayList<Doctor> dList) {
   Scanner readFile = null;
   File file = new File(filename);
   try {
      readFile = new Scanner(file);
   }
   catch(FileNotFoundException e) {
      System.out.println("File not found");
      System.exit(0);
  }
```

```
    while (readFile.hasNext()) {
        String pname = readFile.next();
        Patient p = new Patient(pname, null);
        String dname = readFile.next();
        Doctor d = new Doctor(dname, "", 0);
        int dindex = dList.indexOf(d);
        int pindex = pList.indexOf(p);
        if(pindex !=-1 && dindex!=-1)
          list.add(new Billing(pList.get(pindex),
                               dList.get(dindex)));
    }
    readFile.close();
  }
```

17. Suppose you have a collection of student records in the file `students.txt`. The records contain the name and grade of a student. The records are maintained in an array list. Write a program that fetches data from the text file and builds the array list of student records, then sorts the list by name, calculates the maximum and minimum grades, and the class average. Write a method that separates the students in the list into two lists, one containing records of passing students and one containing records of failing students (use a grade of 60 or more for passing). You are asked to do this in two ways:

a. Create a second list of passing students and a third list of failing students
b. Create a second list of failing students and copy the records of failing students to the new list and remove them from the original list.

Repeat the two solutions in **a** and **b** using a linked list.

Analyze the time and space complexity of the four solutions and compare them.

```
// Class Student
public class Student implements Comparable<Student>{
      private String name;
      private double grade;
      public Student() { name = ""; grade = 0.0; }
      public Student(String n, double g) {
            name = n; grade = g;
      }
      public String getName() { return name; }
      public double getGrade() { return grade; }
      public void setName(String n) { name = n; }
      public void setGrade(double g) { grade = g; }
      public String toString() {
            return name + " " + grade + "\n";
      }
      public int compareTo(Student s) {
            return name.compareTo(s.name);
      }
}
// Test program
public class Test {
   public static void main(String[] args) {
      ArrayList<Student> studentList = new ArrayList<>();
      LinkedList<Student> studentLL = new LinkedList<>();
      ArrayList<Student> passingList = new ArrayList<>();
      ArrayList<Student> failingList = new ArrayList<>();
      LinkedList<Student> passingLL = new LinkedList<>();
      LinkedList<Student> failingLL = new LinkedList<>();
      Scanner readFile = null;
      // Read file students.txt
      File file = new File("students.txt");
      try {
            readFile = new Scanner(file);
      }
      catch(FileNotFoundException e) {
            System.out.println("File not found");
            System.exit(0);
      }
      while (readFile.hasNext()) {
            String name = readFile.next();
            double grade = readFile.nextDouble();
            Student s = new Student(name, grade);
            studentList.add(s);
            studentLL.add(s);
      }
      readFile.close();
      // Sorting student records by name
      java.util.Collections.sort(studentList);
      System.out.println(studentList);
      double max, min, average;
      max = maxGrade(studentList);
      min = minGrade(studentList);
      average = avgGrade(studentList);
      System.out.println("Maximum Grade: " + max);
      System.out.println("Minimum Grade: " + min);
      System.out.println("Average Grade: " + average);

      // Splitting studentList using two additional lists
      splitArrayList1(studentList, passingList,
                                   failingList);
      // Splitting studentList using one additional list
      failingList.clear();
      splitArrayList2(studentList, failingList);
      //Splitting linked list using two additional lists
      splitLinkedList1(studentLL, passingLL, failingLL);
      // Splitting studentLL using one additional list
      failingLL.clear();
      splitLinkedList2(studentLL, failingLL);
}
// Time complexity: O(n)
// Space complexity: 3 * size of the original list
public static void splitArrayList1(ArrayList<Student> list,
   ArrayList<Student> passing, ArrayList<Student> failing) {
      long timeStart = System.nanoTime();
      for(int i=0; i<list.size(); i++) { // O(n)
           Student s = list.get(i); // O(1)
          if(s.getGrade() < 60.0)
                failing.add(s);//O(1)
           else
                passing.add(s);//O(1)
      }
     System.out.println("\nFailing AL size: " +failing.size());
     System.out.println("Passing AL size: "+ passing.size());
     long timeEnd = System.nanoTime();
     System.out.println("Time (ArrayList solution 1): " +
                                  (timeEnd - timeStart));
}
// Time complexity: O(n^2)
// Space complexity: 2 * size of the original list
public static void splitArrayList2(ArrayList<Student> list,
                                ArrayList<Student> failing) {
      long timeStart = System.nanoTime();
      int size = list.size();
      int i=0;
      for(int count=0; count<size; count++) { // O(n)
          Student s = list.get(i); // O(1)
          if(s.getGrade() < 60.0) {
               failing.add(s); // O(1)
               list.remove(s); // O(n)
          }
          else
             i++;
      }
      System.out.println("Failing AL size: "+ failing.size());
      System.out.println("Passing AL size: " + list.size());
      long timeEnd = System.nanoTime();
      System.out.println("Time (ArrayList solution 2): " +
                                    (timeEnd - timeStart));
}
// Time complexity: O(n)
// Space complexity: 2 * size of the original list
public static void splitLinkedList1(LinkedList<Student> list,
                                LinkedList<Student> passing,
                                LinkedList<Student> failing){
  long timeStart = System.nanoTime();
  Iterator<Student> LLIter = list.iterator();
  while(LLIter.hasNext()) { // O(n)
     Student s = LLIter.next(); // O(1)
     if(s.getGrade() < 60.0)
       failing.add(s); // O(1)
     else
       passing.add(s); // O(1)
  }
  System.out.println("\nFailing LL size: "+ failing.size());
  System.out.println("Passing LL size: " +  passing.size());
  long timeEnd = System.nanoTime();
  System.out.println("Time (LinkedList solution 1): " +
                                         (timeEnd - timeStart));
}
// Time complexity: O(n^2)
// Space complexity: size of the original list
public static void splitLinkedList2(LinkedList<Student> list,
                                  LinkedList<Student> failing) {
  long timeStart = System.nanoTime();
  Iterator<Student> LLIter = list.iterator();
  while(LLIter.hasNext()) { // O(n)
     Student s = LLIter.next(); // O(1)
     if(s.getGrade() < 60.0) {
        failing.add(s); // O(1)
        list.remove(s); // O(n)
     }
  }
  System.out.println("Failing LL size: " + failing.size());
  System.out.println("Passing LL size: " + list.size());
  long timeEnd = System.nanoTime();
  System.out.println("Time (LinkedList solution 2): " +
                 (timeEnd - timeStart));
}
// Method maxGrade() returns the highest grade
public static double maxGrade(ArrayList<Student> list) {
  double max = list.get(0).getGrade();
  for(Student s: list)
    if(s.getGrade() > max)
       max = s.getGrade();
  return max;
}
// Method minGrade() returns the lowest grade
public static double minGrade(ArrayList<Student> list) {
  double min = list.get(0).getGrade();
  for(Student s: list)
     if(s.getGrade() < min)
       min = s.getGrade();
  return min;
```

```
}
// Method avgGrade() returns the average grade
public static double avgGrade(ArrayList<Student> list) {
  double avg = 0;
  for(Student s: list)
     avg += s.getGrade();
  return avg/list.size();
}
```