Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Exam Programming Practicum 31/08/2015, 09.00 - 12.00

- You cannot use any material you bring yourself, but all relevant code is provided after the three questions.
- Please start every question on a new sheet.
- Good luck!

Question 1: Linked Lists? (6 points)

A typical implementation of a Double Linked List keeps for each element x two pointers, *prev*[x] and *next*[x], i.e. the addresses of the previous and next element in the list. However, it is possible to implement a Smart Double Linked List using only one pointer value *sp*[x] per element instead of two. The simplified* principle works as follows:

- 0 represents NIL
- *sp*[x] = *prev*[x] + *next*[x]
- The method address(x) returns the address pointer of element x

The following code provides the basis implementation:

```
public class SmartDoubleLinkedList {
  class ListElement {
    private int value;
    private Pointer smartPointer;

    public ListElement(int value, Pointer smartPointer) { ... }
    public int getValue() { ... }
    public int setValue(int s) { ... }
    public Pointer getSmartPointer() { ... }
    public Pointer setSmartPointer(Pointer p) { ... }
  }

  private ListElement head;
  public SmartDoubleLinkedList() {head = null};

}
```

For a given element e and its predecessor element p, the successor element s of e is then determined as follows*:

```
ListElement s = (ListElement) e.smartPointer() - address(p).
```

For example, a list could be represented as follows:

```
e.getValue()         1 2 3 4 5 6
address(e)           5 3 4 2 1 6 (random memory addresses)
e.getSmartPointer() 3 9 5 5 8 1
```

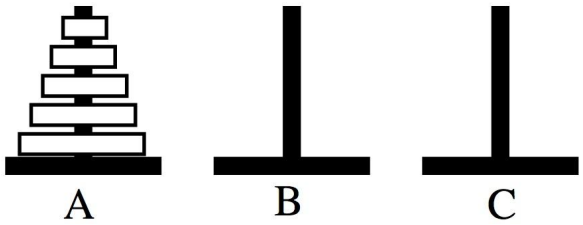
Provide an implementation of the following operations:

- addLast(int e)

- delete(int position)

* the proposed method will not actually work in Java because in Java the pointer address cannot be derived from an object. We're adopting the Java syntax for simplicity and assume a Pointer to an address of a ListElement object can be casted to the ListElement.

Question 2: Recursion (6 points)



The "Towers of Hanoi" is a puzzle that consists of three rods (labeled A, B and C) and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in ascending order of size on one rod, the smallest at the top, thus making a conical shaped "tower".

The objective of the puzzle is to move the entire tower to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the rods and placing it on top of another rod. I.e. a disk can only be moved if it is the uppermost disk on a rod.
- No disk may be placed on top of a smaller disk.

Write a class that contains appropriate data structures to represent the 3 rods. The disks can simply be represented by integer numbers corresponding to the size of the disk, from 1 (smallest, topmost) to n (largest, bottommost).

To puzzle can be recursively solved as follows, move n discs from rod A to rod C:

- move n−1 discs from A to B. This leaves disc n alone on peg A.
- move disc n from A to C
- move n−1 discs from B to C so they sit on disc n

The above is a recursive algorithm, to carry out steps 1 and 3, apply the same algorithm again for n−1. Provide an implementation of this algorithm.

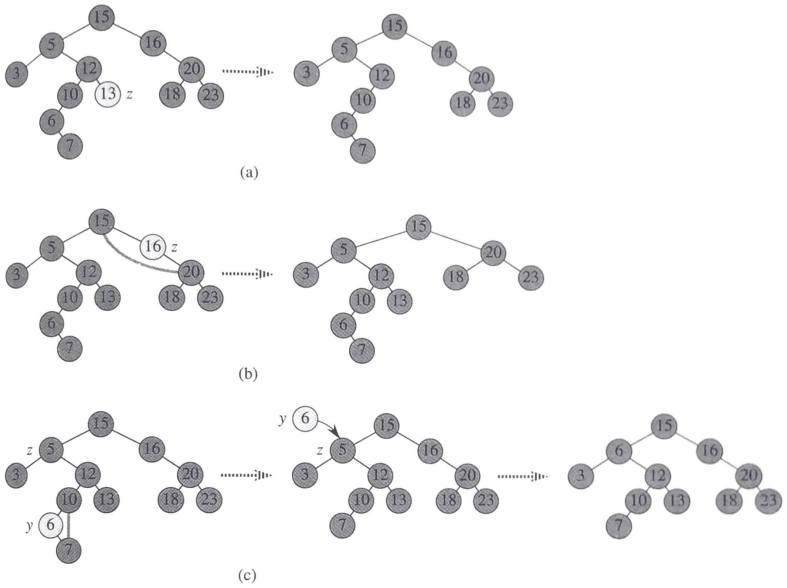
What is the amount of steps required to solve the puzzle?

Question 3: Binary Search Tree (4 points)

The figure belows illustrates the deletion operation of a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded.

A. If z has no children, we just remove it.
B.
C. If z has only one child, we splice out z.

D. If z has two children, we splice out its successor y, which can only have one child, and then replace z's key and satellite data with y's key and satellite data.



1. The procedure relies on the fact that if a node in a binary search tree has two children its successor has no left child and its predecessor has no right child. Explain why.
2. Is the delete operation on a Binary Search Tree commutative? I.e. does deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give an illustrated counterexample.

Question 4: Graphs (4 points)

Write a method that removes a node from a graph in adjacency list representation. You may assume the Linked List class provides a method remove(Comparable e) that removes the element e from the list. What is the time complexity of this method? Explain why.

Question 1/4: Linked Lists / Recursion (5 points)

Write a **recursive** implementation of the reverse method on a LinkedList. The code should only make a single pass over the list.

```
public void reverse() {
      // make call to recursive method


}

private ListElement reverseRec(ListElement e) {
```

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Stacks / Queues (4 points)

Write a method that checks if a s string is a palindrome (word remains the same when reversed; e.g. “racecar”) using only the operations on stacks and queues. What is the time complexity of your method using the provided Stack and Queue implementations? Explain.

```
public static boolean isPalindrome(String word) {
      // word.length() will give you the length of the String
      // word.charAt(0) will give you the character at position 0
```

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4: Binary (Search) Trees (6 points)

1) Write a method that checks whether a binary tree is a binary search tree. What it the time complexity of your method? (3)
2) Write a method that checks if a binary search tree contains two integer numbers that sum to a given integer number *x*. What is the time complexity? (3)

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (5 points)

Using the adjacency list graph representation, write a method that returns a (linked) list of predecessors of a given node. You also have to provide the implementation of helper methods on underlying data structures. What is the time complexity of this operation?

Question 1/4 (4 points)

Write a method **LinkedList extractOdds()** on the LinkedList class that traverses the list and extracts the odd elements (in-place) and returns a new linked list with the extracted elements.

For example, for the following list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11

the original list would have following elements:

2 -> 4 -> 6 -> 8 -> 10

and the newly created and returned list would contain:

1 -> 3 -> 5 -> 7 -> 9 -> 11

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

¶

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

In a binary tree (**not** a binary search tree), the elements are not ordered. It means that for a given node, both children can be larger or smaller. It is possible that the same element even occurs more than once in the tree. Let's suppose you need to extend the implementation of this binary tree (*not binary search tree*) storing numbers, but you don't have access to other methods of the tree, only the root node.
Hint: For some questions you could use a Vector or a LinkedList implementation that has the addSorted(…) method.

a) Write a method **boolean equals(TreeNode root1, TreeNode root2)** that checks if the two binary trees are structurally identical - they contain nodes with the same values arranged in the same way.
b) What is the time complexity of the method provided in *a)*?
c) Write a method **boolean valuesEqual(TreeNode root1, TreeNode root2)** that checks if values of the binary trees are equal regardless the structure.
d) What would be the time complexity following the strategy provided in *c)*?

¶

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Write a method **void permute(Vector objects)** that uses a stack S and/or a queue Q to print all possible permutations of an n-element set T non-recursively. Since the parameters are objects, you should use the toString() method.

Permutations for objects ‘A’, ‘A’, ‘C’ are:

[AAC, ACA, CAA]

Permutations for ‘A’, ‘B’, ‘C’ are:

[ACB, ABC, BCA, CBA, CAB, BAC]

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

¶

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (6 points)

In a directed graph, the in-degree of a vertex is defined as the number of incoming edges in the vertex. Similarly, the out-degree of a vertex is defined as the number of outgoing edges of the vertex.

Extend the edge list representation of a graph to compute the in-degree and out-degree of a given vertex. Do this by Implementing the following methods on the graph class

int outDegree(string nodeLabel)

int inDegree(string nodeLabel)

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
```

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

```
    return result;
  }
}
```

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Queue

```
public class Queue
{
        private Vector data;

        public Queue ()
        {
           data = new Vector();
        }

        public void push(Comparable item)
        {
           data.addLast(item);
        }

        public Comparable pop()
        {
           Comparable element = data.getFirst();
           data.removeFirst();
           return element;
        }

        ….
}
```

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }

    public TreeNode getRightTree()
    {
      return rightNode;
    }

    public Comparable getValue()
    {
      return value;
    }

  }

  protected TreeNode root;

  public Tree()
  {
    root = null;
  }

  …

}
```

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
```

```
  }
}
```

Question 1/4 (4 points)

Write a method **`int addAfterEach(object x, object y)`** on the LinkedList class that inserts an object **`x`** after each occurrence of the element **`y`**. The method should work in place and use only a single traversal. The method will return number of insertions.

For example, for the following list:

```
1 -> 2 -> 3 -> 4 -> 3 -> 2 -> 1
```

after calling method **`addAfterEach(3, 42)`**, the list will be modified to:

```
1 -> 2 -> 3 -> 42 -> 4 -> 3 -> 42 -> 1
```

and the method will return number **`2`**.

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

Write a method **`int balanced(string code)`** that traverses a string (a sequence of characters) from left to right and determines whether its parentheses are "balanced". The method will return the position of the first error in the string or `-1` otherwise. What is the time complexity of the method provided?
Hint: Use stack

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

If we keep a pointer to the largest element in a BST, we can trivially make a method to retrieve the largest element in O(1). However, it requires an update of the insert and remove method to make sure the pointer to the largest element remains correct.

1. Give an insert method on the BST which updates the pointer to the largest element.

2. What is the time complexity of this method?

3. If the BST is a red-black tree, how should the insert method of red black tree be updated to handle this pointer? (No code is required, but explain the idea in a few sentences + a drawing.)

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (6 points)

For a directed graph, makes a method that returns the path (a list of nodes) of a cycle in case it exists. In case there is no cycle, return false.

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }
    public ListElement(Comparable el)
    {
      this(el,null);
    }
    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;
  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

   public class TreeNode implements Comparable {
     protected Comparable   value;
     protected TreeNode   leftNode;
     protected TreeNode   rightNode;

     public TreeNode(Comparable v)
     {
       value = v;
       leftNode = null;
       rightNode = null;
     }

     public TreeNode(Comparable v, TreeNode left, TreeNode right)
     {
       value = v;
       leftNode = left;
       rightNode = right;
     }

     public TreeNode getLeftTree()
     {
       return leftNode;
     }
     public TreeNode getRightTree()
     {
       return rightNode;
```

```
    }

        public Comparable getValue()
        {
                return value;
        }
}

        protected TreeNode root;

        public Tree()
        {
                root = null;
        }


        public void insert(Comparable element)
        {
                insertAtNode(element,root,null);
        }

        private void insertAtNode(Comparable element,TreeNode current,TreeNode parent)
        {
                if(current == null)
                {
                        TreeNode newNode = new TreeNode(element);
                        if(parent != null)
                        {
                                if(element.compareTo(parent.value) < 0)
                                {
                                        parent.leftNode = newNode;
                                }
                                else
                                {
                                        parent.rightNode = newNode;
                                }
                        }
                        else root = newNode;
                }
                else if(element.compareTo(current.value) == 0)
                {
                        // if the element is already in the tree, what to do?
                }
                else if(element.compareTo(current.value) < 0)
                {
                        insertAtNode(element,current.getLeftTree(),current);
                }
                else insertAtNode(element,current.getRightTree(),current);
        }

        ....
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;
      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;
   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Datastructures 25/01/2015, 09.00 - 12.00 Name:

You cannot use any material you bring yourself, but all relevant code is provided after the three questions. Good luck!

Question 1/4: Linked Lists / Recursion (5 points)

Write a **recursive** implementation of the reverse method on a LinkedList. The code should only make a single pass over the list.

```
public void reverse() {
      // make call to recursive method


}

private ListElement reverseRec(ListElement e) {
```

Exam Algorithms and Datastructures 25/01/2015, 09.00 - 12.00 Name:

You cannot use any material you bring yourself, but all relevant code is provided after the three questions. Good luck!

Question 2/4: Stacks / Queues (4 points)

Write a method that checks if a s string is a palindrome (word remains the same when reversed; e.g. “racecar”) using only the operations on stacks and queues. What is the time complexity of your method using the provided Stack and Queue implementations? Explain.

```
public static boolean isPalindrome(String word) {
      // word.length() will give you the length of the String
      // word.charAt(0) will give you the character at position 0
```

Exam Algorithms and Datastructures 25/01/2015, 09.00 - 12.00 **Name:**

You cannot use any material you bring yourself, but all relevant code is provided after the three questions. Good luck!

Question 3/4: Binary (Search) Trees (6 points)

1) Write a method that checks whether a binary tree is a binary search tree. What it the time complexity of your method? (3)
2) Write a method that checks if a binary search tree contains two integer numbers that sum to a given integer number *x*. What is the time complexity? (3)

Exam Algorithms and Datastructures 25/01/2015, 09.00 - 12.00 **Name:**

You cannot use any material you bring yourself, but all relevant code is provided after the three questions. Good luck!

Question 4/4: Graphs (5 points)

Using the adjacency list graph representation, write a method that returns a (linked) list of predecessors of a given node. You also have to provide the implementation of helper methods on underlying data structures. What is the time complexity of this operation?

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4 (4 points)

Write a method **LinkedList extractOdds()** on the LinkedList class that traverses the list and extracts the odd elements (in-place) and returns a new linked list with the extracted elements.

For example, for the following list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11

the original list would have following elements:

2 -> 4 -> 6 -> 8 -> 10

and the newly created and returned list would contain:

1 -> 3 -> 5 -> 7 -> 9 -> 11

¶

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

In a binary tree (**not** a binary search tree), the elements are not ordered. It means that for a given node, both children can be larger or smaller. It is possible that the same element even occurs more than once in the tree. Let's suppose you need to extend the implementation of this binary tree (*not binary search tree*) storing numbers, but you don't have access to other methods of the tree, only the root node.
Hint: For some questions you could use a Vector or a LinkedList implementation that has the addSorted(…) method.

a) Write a method **boolean equals(TreeNode root1, TreeNode root2)** that checks if the two binary trees are structurally identical - they contain nodes with the same values arranged in the same way.
b) What is the time complexity of the method provided in *a)*?
c) Write a method **boolean valuesEqual(TreeNode root1, TreeNode root2)** that checks if values of the binary trees are equal regardless the structure.
d) What would be the time complexity following the strategy provided in *c)*?

¶

Exam Algorithms and Data Structures 21/08/2018, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Write a method **void permute(Vector objects)** that uses a stack S and/or a queue Q to print all possible permutations of an n-element set T non-recursively. Since the parameters are objects, you should use the toString() method.

Permutations for objects ‘A’, ‘A’, ‘C’ are:

[AAC, ACA, CAA]

Permutations for ‘A’, ‘B’, ‘C’ are:

[ACB, ABC, BCA, CBA, CAB, BAC]

¶

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (6 points)

In a directed graph, the in-degree of a vertex is defined as the number of incoming edges in the vertex. Similarly, the out-degree of a vertex is defined as the number of outgoing edges of the vertex.

Extend the edge list representation of a graph to compute the in-degree and out-degree of a given vertex. Do this by Implementing the following methods on the graph class

int outDegree(string nodeLabel)

int inDegree(string nodeLabel)

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Queue

```
public class Queue
{
        private Vector data;

        public Queue ()
        {
           data = new Vector();
        }

        public void push(Comparable item)
        {
           data.addLast(item);
        }

        public Comparable pop()
        {
           Comparable element = data.getFirst();
           data.removeFirst();
           return element;
        }

        ….
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }

    public TreeNode getRightTree()
    {
      return rightNode;
    }

    public Comparable getValue()
    {
      return value;
    }

  }

  protected TreeNode root;

  public Tree()
  {
    root = null;
  }

  …

}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 28/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4 (4 points)

Write a method **`int addAfterEach(object x, object y)`** on the LinkedList class that inserts an object **`x`** after each occurrence of the element **`y`**. The method should work in place and use only a single traversal. The method will return number of insertions.

For example, for the following list:

```
1 -> 2 -> 3 -> 4 -> 3 -> 2 -> 1
```

after calling method **`addAfterEach(3, 42)`**, the list will be modified to:

```
1 -> 2 -> 3 -> 42 -> 4 -> 3 -> 42 -> 1
```

and the method will return number **`2`**.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

Write a method **`int balanced(string code)`** that traverses a string (a sequence of characters) from left to right and determines whether its parentheses are "balanced". The method will return the position of the first error in the string or `-1` otherwise. What is the time complexity of the method provided?
Hint: Use stack

Question 3/4 (4 points)

If we keep a pointer to the largest element in a BST, we can trivially make a method to retrieve the largest element in O(1). However, it requires an update of the insert and remove method to make sure the pointer to the largest element remains correct.

1. Give an insert method on the BST which updates the pointer to the largest element.

2. What is the time complexity of this method?

3. If the BST is a red-black tree, how should the insert method of red black tree be updated to handle this pointer? (No code is required, but explain the idea in a few sentences + a drawing.)

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (6 points)

For a directed graph, makes a method that returns the path (a list of nodes) of a cycle in case it exists. In case there is no cycle, return false.

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }
    public ListElement(Comparable el)
    {
      this(el,null);
    }
    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;
  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }
    public TreeNode getRightTree()
    {
      return rightNode;
    }

        public Comparable getValue()
        {
```

```
		return value;
	}
}

	protected TreeNode root;

	public Tree()
	{
		root = null;
	}


	public void insert(Comparable element)
	{
		insertAtNode(element,root,null);
	}

	private void insertAtNode(Comparable element,TreeNode current,TreeNode parent)
	{
		if(current == null)
		{
			TreeNode newNode = new TreeNode(element);
			if(parent != null)
			{
				if(element.compareTo(parent.value) < 0)
				{
					parent.leftNode = newNode;
				}
				else
				{
					parent.rightNode = newNode;
				}
			}
			else root = newNode;
		}
		else if(element.compareTo(current.value) == 0)
		{
			// if the element is already in the tree, what to do?
		}
		else if(element.compareTo(current.value) < 0)
		{
			insertAtNode(element,current.getLeftTree(),current);
		}
		else insertAtNode(element,current.getRightTree(),current);
	}

	....
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;
      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;
   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 24/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4 (4 points)

Write a method **`int addBeforeEach(object x, object y)`** on the LinkedList class that inserts an object **`x`** before each occurrence of the element **`y`**. The method should work in place and use only a single traversal. The method will return number of insertions.

For example, for the following list:

```
1 -> 2 -> 3 -> 4 -> 3 -> 2 -> 1
```

after calling method **`addBeforeEach(3, 42)`**, the list will be modified to:

```
1 -> 2 -> 42 -> 3 -> 4 -> 42 -> 3 -> 1
```

and the method will return number **`2`**.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

Write a method **`int balanced(string code)`** that traverses a string (a sequence of characters) from left to right and determines whether its parentheses are "balanced". The method will return the position of the first error in the string or `-1` otherwise. What is the time complexity of the method provided?
Hint: Use stack

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Write a method **`TreeNode next(TreeNode root, TreeNode current)`** that returns the successor of a node in a binary search tree (the tree stores integer keys, and the successor is defined as a node with the smallest key that is larger than the key of the **`current`** node). What is the time complexity of the method? (use the big o notation, and define the meaning of n)

Exam Algorithms and Data Structures 24/01/2019, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (6 points)

A bridge in a graph is an edge that, if removed, would separate a connected graph into two disjoint subgraphs. A graph that has no bridges is said to be two-edge connected. Write a method that determines whether a given graph is two-edge connected. What is the time complexity of the method?

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Queue

```
public class Queue
{
        private Vector data;

        public Queue ()
        {
           data = new Vector();
        }

        public void push(Comparable item)
        {
           data.addLast(item);
        }

        public Comparable pop()
        {
           Comparable element = data.getFirst();
           data.removeFirst();
           return element;
        }

        ….
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }

    public TreeNode getRightTree()
    {
      return rightNode;
    }

    public Comparable getValue()
    {
      return value;
    }

  }

  protected TreeNode root;

  public Tree()
  {
    root = null;
  }

  …

}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 27/01/2020, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Run Length Encoding (RLE) is a simple compression method in which each sequence of the same data value in consecutive data elements is stored as a single data value and count.
E.g. A binary image containing only black and white pixels, represented in a vector of “B” and “W” elements - BBBWWWBWWWWWW - is encoded into B3W3B1W6.

1. Write a method which implements RLE on a vector containing only “B” and “W” elements. **Only traverse the given vector once**!
2. Define and motivate the time complexity of this method.

Exam Algorithms and Data Structures 27/01/2020, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 Binary Search Trees (5 points)

The range of a Binary Search Tree is defined as the difference between the largest and the smallest element. Write a method which computes the range of a BST **without** traversing the entire tree. Define and motivate the time complexity of this method.

Exam Algorithms and Data Structures 27/01/2020, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 Queues (6 points)

A **double-ended priority queue** (**DEPQ**) is a data structure, similar to priority queue but DEPQ has methods to remove the maximum and minimum priority element (the priority is defined as a comparable or a number). Implement the following methods of the DEPQ.

1. Implement the popMax() and popMin() methods such that their time complexity is O(1).
2. Implement the push(Object o, int priority) method that inserts an element to the DEPQ. What is the time complexity of your push(…) method?

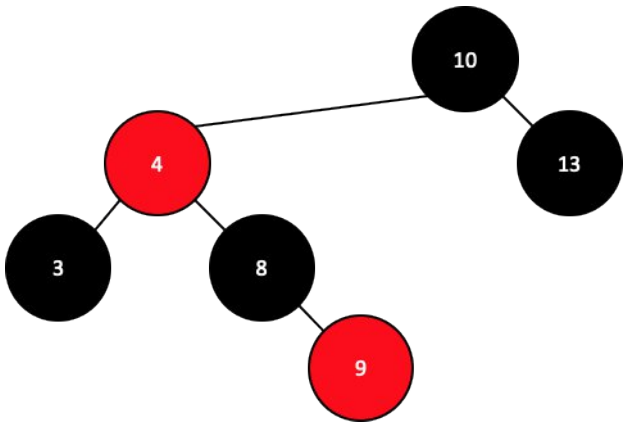
Exam Algorithms and Data Structures 27/01/2020, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4 Red Black Trees (4 points)

For the following statements, explain whether the statements are correct or not and provide a clear motivation. No score will be given for answers without motivation.

1. Is it true that traversing a red-black tree with n nodes in in-order takes O(n log n) time?
2. Is it true that a red-black tree with n elements, the elements can be sorted in at most O(n log n)?
3. Is it true that the time complexity of building a red-black tree with n elements is $O(n^2)$?
4. Is it true that the insertion of the elements 13, 10, 8, 3, 4 and 9 in an empty red black tree can result in the following tree?



Provided Code

Vector

```
public class Vector {
	protected Object data[];
	protected int count;

	public Vector(int capacity) {
		data = new Object[capacity];
		count = 0;
	}

	public int size() {
		return count;
	}

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

	public Object get(int index) {
		return data[index];
	}

	public void set(int index, Object obj) {
		data[index] = obj;
	}

	public boolean contains(Object obj) {
		for(int i=0;i<count;i++) {
			if(data[i] == obj) return true;
		}
		return false;
	}

	public void addFirst(Object item) { … }

	public void addLast(Object o) {
		data[count] = o;
		count++;
	}

	public Object getFirst() { … }

	public Object getLast() { … }

	public void removeLast() { .. }

	public void removeFirst() { … }

}
```

Binary Tree

```
public class Tree {
      public class TreeNode implements Comparable {
        protected Comparable   value;
        protected TreeNode   leftNode;
        protected TreeNode   rightNode;
        public TreeNode(Comparable v) {
          value = v;
          leftNode = null;
          rightNode = null;
        }
        public TreeNode(Comparable v, TreeNode left, TreeNode right) {
          value = v;
          leftNode = left;
          rightNode = right;
        }
        public TreeNode getLeftTree() {
          return leftNode;
        }
        public TreeNode getRightTree() {
          return rightNode;
        }
        public Comparable getValue() {
           return value;
        }
    }
    protected TreeNode root;
    public Tree() {
        root = null;
    }

    public void insert(Comparable element) {
        insertAtNode(element,root,null);
    }

    private void insertAtNode(Comparable element,TreeNode current,TreeNode parent)              {
          if(current == null) {
              TreeNode newNode = new TreeNode(element);
              if(parent != null) {
                    if(element.compareTo(parent.value) < 0) {
                        parent.leftNode = newNode;
                    }
                    else {
                        parent.rightNode = newNode;
                    }
              }
              else root = newNode;
          }
          else if(element.compareTo(current.value) == 0) { // if the element is already in the tree, what to do? }
          else if(element.compareTo(current.value) < 0) {
                    insertAtNode(element,current.getLeftTree(),current);
          }
          else insertAtNode(element,current.getRightTree(),current);
  }
  ....
}
```

PriorityQueue

```
public class PriorityQueue {
	public class PriorityPair implements Comparable {
		public Object element;
		public Comparable priority;
		public PriorityPair(Object element, Object priority) {
			this.element = element;
			this.priority = priority;
		}
		public int compareTo(Object o) {
			PriorityPair p2 = (PriorityPair) o;
			return priority.compareTo(p2.priority);
		}
	}

	private LinkedList data;

	public PriorityQueue() {
		data = new LinkedList();
	}

	public void push(Object o, int priority) { ... }

	public Object pop() { ... }

	public Object top(){ ... }
}
```

CircularVector

```
public class CircularVector {
	private Vector data;
	private int first;
	private int count;

	public CircularVector() {
		count = 0;
		first = 0;
		data = new Vector(5);
	}

	public int size() {  return count; }
	public void addFirst(Object element) { ... }
	public void addLast(Object element) { ... }
	public Object getFirst() { ... }
	public Object getLast() { ... }
	public void removeFirst() {
		if(count > 0) {
			first = (first+1)%data.capacity();
			count--;
		}
	}

	public void removeLast() { ... }
}
```

DoubleLinkedList

```
public class DoubleLinkedList {
	private class DoubleLinkedListElement {
		private Object data;
		private DoubleLinkedListElement nextElement;
		private DoubleLinkedListElement previousElement;
		public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous) {
			data = v;
			nextElement = next;
			previousElement = previous;
			if(nextElement != null) nextElement.previousElement = this;
			if(previousElement != null) previousElement.nextElement = this;
		}

		public DoubleLinkedListElement(Object v) { this(v,null,null); }

		public DoubleLinkedListElement previous() { return previousElement; }

		public Object value() { return data; }

		public DoubleLinkedListElement next() { return nextElement; }

		public void setNext(DoubleLinkedListElement value) { nextElement = value; }

		public void setPrevious(DoubleLinkedListElement value) { previousElement = value; }
	}
	private int count;
	private DoubleLinkedListElement head;
	private DoubleLinkedListElement tail;

	public DoubleLinkedList()  {
		head = null;
		tail = null;
		count = 0;
	}

	public Object getFirst() { return head.value(); }

	public Object getLast() { return tail.value(); }

	public int size() {  return count; }

	public void addFirst(Object value) {
		head = new DoubleLinkedListElement(value,head,null);
		if(tail == null) tail = head;
		count++;
	}
	public void addLast(Object value) {
		tail = new DoubleLinkedListElement(value,null,tail);
		if(head == null) head = tail;
		count++;
	}
}
```

Exam Algorithms and Data Structures 25/01/2021, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Write a method on Vector class that transforms the given vector such that all odd numbers are placed first (before) the even numbers. You must preserve the ordering from the original vector within the group of odd and even numbers E.g., for vector [4, 5, 1, 3, 11, 25, 2, 30, 13] the final vector would be [5, 1, 3, 11, 25, 13, 4, 2, 30].

- What is the time complexity of your method?
- If you would have used a Linked List instead of the Vector, would the time complexity change? Why?

Exam Algorithms and Data Structures 25/01/2021, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 Binary Search Trees (5 points)

~~The range of a Binary Search Tree is defined as the difference between the largest and the smallest element. Write a method which computes the range of a BST **without** traversing the entire tree. Define and motivate the time complexity of this method.~~

Assume that a BinarySearchTree has nodes extended ~~by~~ with a property childrenCount that accurately represents the total number of children under the node. Write a method median() on the tree with an algorithm that finds the median element and returns its value without traversing all elements.¶

¶

$$simplified\_median(X) = \begin{cases} X\left[\frac{n}{2}\right] & if\ \boldsymbol{n}\ is\ even \\ X\left[\frac{n+1}{2}\right] & if\ \boldsymbol{n}\ is\ odd \end{cases}$$

¶

(Just an idea, perhaps too easy?)
Write the find method for our BinarySearchTree class binary search not using recursion but using a loop.

Exam Algorithms and Data Structures 25/01/2021, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 Queues (6 points)

Assuming a priority queue can only have ~~only~~ Integer priorities~~y~~. Write a method getAverage() (and update other methods if necessary) that returns the average of all priorities in the priority queue with a time complexity O(1) AND make sure it will preserve the time complexity of any updated method.

Exam Algorithms and Data Structures 25/01/2021, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4 Graphs (4 points)

Write a method that receives a vector containing N graphs and returns a new graph that is the intersection of all N graphs. Assume the edge-list representation of the of graphs. The intersection of two graphs is a graph containing only nodes and edges present in both graphs.

Write a method which for each edge computes how many times that edge is part of the shortest path from each node to all other nodes. Edges are undirected.

- E.g. on a graph with nodes A, B, C and edges from each node to all others the shortest paths like this:

  A->B A->B->C

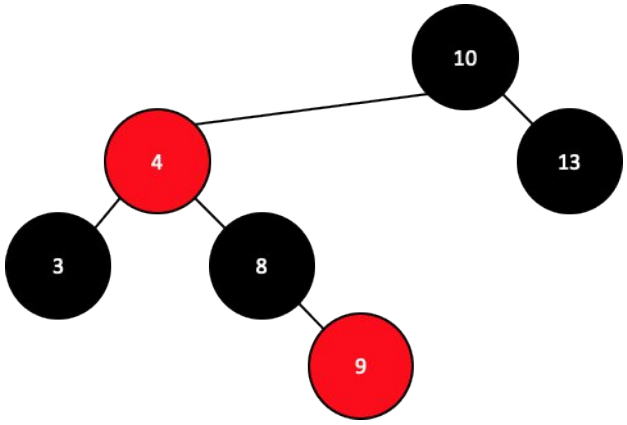
  B->A B->C

  C->B->A

- the edge A->B is "used" 4 times (in 4 out of these 5 paths), edge B->C is "used" 3x, edge A->C 0x

~~For the following statements, explain whether the statements are correct or not and provide a clear motivation. No score will be given for answers without motivation.~~

1. ~~Is it true that traversing a red-black tree with n nodes in in-order takes O(n log n) time?~~
2. ~~Is it true that a red-black tree with n elements, the elements can be sorted in at most O(n log n)?~~
3. ~~Is it true that the time complexity of building a red-black tree with n elements is $O(n^2)$?~~
4. ~~Is it true that the insertion of the elements 13, 10, 8, 3, 4 and 9 in an empty red-black tree can result in the following tree?~~



Provided Code

Vector

```
public class Vector {
	protected Object data[];
	protected int count;

	public Vector(int capacity) {
		data = new Object[capacity];
		count = 0;
	}

	public int size() {
		return count;
	}

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

	public Object get(int index) {
		return data[index];
	}

	public void set(int index, Object obj) {
		data[index] = obj;
	}

	public boolean contains(Object obj) {
		for(int i=0;i<count;i++) {
			if(data[i] == obj) return true;
		}
		return false;
	}

	public void addFirst(Object item) { … }

	public void addLast(Object o) {
		data[count] = o;
		count++;
	}

	public Object getFirst() { … }

	public Object getLast() { … }

	public void removeLast() { .. }

	public void removeFirst() { … }

}
```

Binary Tree

```
public class Tree {
      public class TreeNode implements Comparable {
        protected Comparable   value;
        protected int  childrenCount;
        protected TreeNode   leftNode;
        protected TreeNode   rightNode;
        public TreeNode(Comparable v) {
          value = v;
          leftNode = null;
          rightNode = null;
        }
        public TreeNode(Comparable v, TreeNode left, TreeNode right) {
          value = v;
          leftNode = left;
          rightNode = right;
        }
        public TreeNode getLeftTree() {
          return leftNode;
        }
        public TreeNode getRightTree() {
          return rightNode;
        }
        public Comparable getValue() {
           return value;
        }
    }
    protected TreeNode root;
    ....
}
```

PriorityQueue

```
public class PriorityQueue {
	public class PriorityPair implements Comparable {
		public Object element;
		public Comparable priority;
		public PriorityPair(Object element, Object priority) {
			this.element = element;
			this.priority = priority;
		}
		public int compareTo(Object o) {
			PriorityPair p2 = (PriorityPair) o;
			return priority.compareTo(p2.priority);
		}
	}

	private LinkedList data;

	public PriorityQueue() {
		data = new LinkedList();
	}

	public void push(Object o, int priority) { ... }

	public Object pop() { ... }

	public Object top(){ ... }
}
```

CircularVector

```
public class CircularVector {
	private Vector data;
	private int first;
	private int count;

	public CircularVector() {
		count = 0;
		first = 0;
		data = new Vector(5);
	}

	public int size() {  return count; }
	public void addFirst(Object element) { ... }
	public void addLast(Object element) { ... }
	public Object getFirst() { ... }
	public Object getLast() { ... }
	public void removeFirst() {
		if(count > 0) {
			first = (first+1)%data.capacity();
			count--;
		}
	}

	public void removeLast() { ... }
}
```

DoubleLinkedList

```
public class DoubleLinkedList {
	private class DoubleLinkedListElement {
		private Object data;
		private DoubleLinkedListElement nextElement;
		private DoubleLinkedListElement previousElement;
		public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous) {
			data = v;
			nextElement = next;
			previousElement = previous;
			if(nextElement != null) nextElement.previousElement = this;
			if(previousElement != null) previousElement.nextElement = this;
		}

		public DoubleLinkedListElement(Object v) { this(v,null,null); }

		public DoubleLinkedListElement previous() { return previousElement; }

		public Object value() { return data; }

		public DoubleLinkedListElement next() { return nextElement; }

		public void setNext(DoubleLinkedListElement value) { nextElement = value; }

		public void setPrevious(DoubleLinkedListElement value) { previousElement = value; }
	}
	private int count;
	private DoubleLinkedListElement head;
	private DoubleLinkedListElement tail;

	public DoubleLinkedList()  {
		head = null;
		tail = null;
		count = 0;
	}

	public Object getFirst() { return head.value(); }

	public Object getLast() { return tail.value(); }

	public int size() {  return count; }

	public void addFirst(Object value) {
		head = new DoubleLinkedListElement(value,head,null);
		if(tail == null) tail = head;
		count++;
	}
	public void addLast(Object value) {
		tail = new DoubleLinkedListElement(value,null,tail);
		if(head == null) head = tail;
		count++;
	}
}
```

Question 1

Write a method **`void customInsert(int number)`** on the LinkedList class that accepts one number as an argument. If the given number is already present in the list, insert the new element at the end of the first sequence of this number. If the number is not in the list, insert it in the beginning of the list. The method **`customInsert(…)`** should traverse the elements only once and should not copy all elements to a new list.

A. What is the time complexity of your method?
B. If you would have used a Vector instead of the Linked List, would the time complexity change? Why?

A template of the method:

```
void customInsert(int number){

  // Write your answer here

}
```

Question 2

On a BinarySearchTree write a method `secondSmallest()` that returns the second smallest element that is contained in the tree. The method should work for all cases of BST, while avoiding traversal of all elements (except the worst-case scenario).

A template of the method:

```
Comparable secondSmallest(){

 // Write your answer here

}
```

Question 3

Assuming a priority queue can only have Integer priorities. Write a method `uniquePriorities()` (and update other methods if necessary) that returns the number of unique priorities in the priority queue with a time complexity $O(1)$ AND make sure it will preserve the time complexity of any updated method.

A template of the method:

```
int uniquePriorities(){

  // Write your answer here

}
```

Question 4

Write a method **int cycles()** on that will count all cycles on a directed graph in the adjacency list representation. Don't forget to provide implementations of missing methods of underlying data structures if needed.

A template of the method:

```
int cycles(){

  // Write your answer here

}
```

Question 1

Write a method **`void customDelete(int number)`** on the LinkedList class that accepts one number as an argument. LinkedList is ordered in a sense, that the same numbers are kept as a sequence, e.g. 1-1-1-7-4-4-4-9-3.

If the given number is present in the list, remove the element at the end of the sequence of this number. If the number is not in the list, do not change the list. The method **`customDelete(int number)`** should traverse the elements only once and should not copy all elements to a new list.

A. What is the time complexity of your method?
B. If you would have used a Vector instead of the Linked List, would the time complexity change? Why?

A template of the method:

```
void customDelete(int number){

  // Write your answer here

}
```

Example: in given LinkedList customDelete(4) should delete element on index 6

index	0	1	2	3	4	5	**6**	7	8
element	1	1	1	7	4	4	**4**	9	3

Question 2 (middle Element)

On a BinarySearchTree write a **`Comparable method middle()`** that returns the median or an element closest to the median contained in the tree. The method should work for all cases of BST (also for unbalanced trees).

A template of the method:

```
Comparable median(){

 // Write your answer here
```

```
}
```

Question 3 (return most represented priority)

Assuming a priority queue can only have Integer priorities. Write a method `int mostFrequentPriority()` (and update other methods if necessary) that returns the priority that is most represented in the priority queue. Make sure it will preserve the time complexity of any updated method.

A template of the method:

```
int mostFrequentPriority(){
  // Write your answer here
}
```

Example: If a Priority Queue contains priorities as follows: 1-1-2-3-4-5-6-6-6-7-7, your method should return "6".

Question 4 (remove all nodes of degree N)

Write a method **void removeNthDegree(int n)** on Graph that will remove all vertices of Nth degree (i.e. nodes with N edges) on a directed graph in the adjacency list representation. Don't forget to provide implementations of missing methods of underlying data structures if needed.

A template of the method:

```
void removeNthDegree(int n){
  // Write your answer here
}
```

Exam Algorithms and Data Structures 31/01/2022, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4 (5 points)

Write a method **`void removeEveryNth(int n)`** on the LinkedList class that traverses the list and (in-place) removes every n-th element .

For example, after calling **`removeEveryNth(3)`** on the following list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11

will result in a list containing following elements:

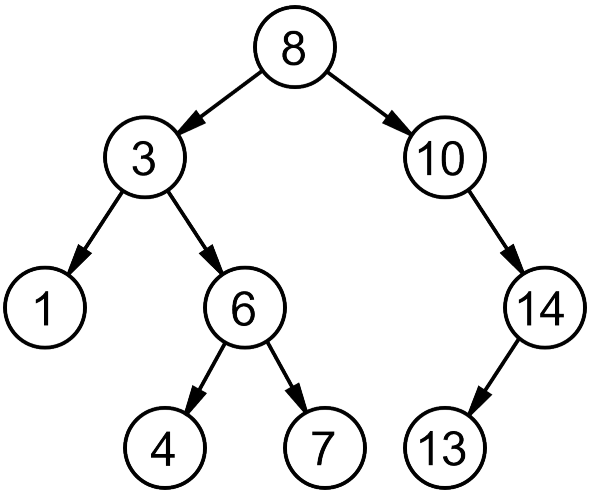
1 -> 2 -> 4 -> 5 -> 7 -> 8 -> 10 -> 11

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4 (6 points)

Write a method **`void minMaxTraverse(TreeAction action)`** on the binary search tree that traverses all elements in the tree interleaving the minimum and maximum elements, converging all the way to the median.

For example: for the given tree:



The method will traverse the elements in this order: 1, 14, 3, 13, 4, 10, 6, 8, 7
What is the time complexity of the method?

Hint: *Check supplementary material,* **`traverse()`** *and* **`traverseInOrder()`** *methods might help as an inspiration.*

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Assume a stack data structure contains only numbers and has push, pop, and top operations implemented in O(1). Write a method **`int getMax()`** that returns the largest element on stack in O(1) while preserving the time complexity of the other methods. You will need to change the other methods too, so please provide the updated implementation of all the methods you need to change.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: (5 points)

A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles. Write a **`boolean`** `isDAG()` method on the graph to check whether a given graph is a DAG or not. You can assume the edge list representation.

What is the time complexity of the method?

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode;}

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n,TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Write a method **`void removeDuplicates()`** on the Vector class that traverses the vector and (in-place) removes all duplicates in it without creating or using an additional vector (or any other collections).
For example, after calling **`removeDuplicates()`** on the following vector:

[1, 2, 3, 3, 2, 4, 1]

will result in a vector containing the following elements:

[1, 2, 3, 4]

What is the time complexity of your algorithm?

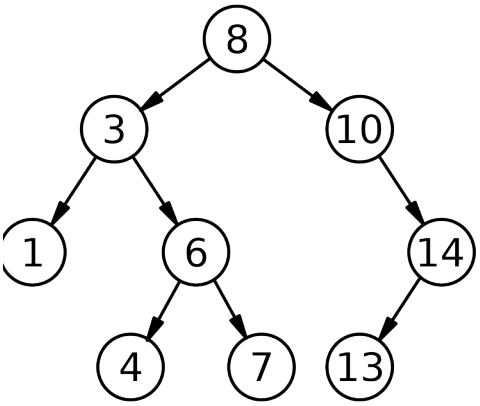
Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Trees (6 points)

A different way to store a binary search tree than the method we have seen in class, is to store all the data in a vector. Every element in the tree corresponds to a position in the array, according to a fixed rule. The root is stored in the array on position 1, and for every element stored at position i, the left child is stored at position 2*i and the right child is stored at position 2*i+1.

For instance, for the tree on the left, the corresponding vector is given on the right



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/	8	3	10	1	6		14			4	7			13

a. Implement a method SetLeftChild(int index, Comparable value) which takes the index in the vector of the current node and assigns the given value to the left child.
b. A method “swap” on a binary search tree mirrors the tree, i.e the left and right children are swapped. For instance, in the picture below, the tree and its swapped version are shown. Implement a method swap on a binary search tree stored in a vector as explained above. Use SetLeftChild and SetRightChild (you can assume this method exists).
c. What is the time complexity of your swap method. Motivate.

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Assume a circular structure called ***Circular double linked list***. This structure is similar to Double linked list, but the elements are chained in a circle, meaning the predecessor of the first element is the last element and the successor of the last element is the first element. Write the methods addFirst(...), addLast(), removeFirst() and removeLast().

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: (5 points)

Consider a directed acyclic graph. A “transpose” of a graphs is the same graph, with all edges transposed, i.e. an edge from node i to node j is transposed into an edge from node j to node i. Write a transpose method on a graph in adjacency matrix representation. Motivate the time complexity of the method.

Provided Code

Vector

```
public class Vector {
	protected Object data[];
	protected int count;

	public Vector(int capacity) {
		data = new Object[capacity];
		count = 0;
	}

	public int size() {
		return count;
	}

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

	public Object get(int index) {
		return data[index];
	}

	public void set(int index, Object obj) {
		data[index] = obj;
	}

	public boolean contains(Object obj) {
		for(int i=0;i<count;i++) {
			if(data[i] == obj) return true;
		}
		return false;
	}

	public void addFirst(Object item) { … }

	public void addLast(Object o) {
		data[count] = o;
		count++;
	}

	public Object getFirst() { … }

	public Object getLast() { … }

	public void removeLast() { .. }

	public void removeFirst() { … }

}
```

DoubleLinkedList

```
public class DoubleLinkedList {
	private class DoubleLinkedListElement {
		private Object data;
		private DoubleLinkedListElement nextElement;
		private DoubleLinkedListElement previousElement;
		public DoubleLinkedListElement(Object v,
			DoubleLinkedListElement next,
			DoubleLinkedListElement previous) {
				data = v;
				nextElement = next;
				previousElement = previous;
				if (nextElement != null)
					nextElement.previousElement = this;
				if (previousElement != null)
					previousElement.nextElement = this;
		}
		public DoubleLinkedListElement(Object v) {
			this(v, null, null);
		}
		public DoubleLinkedListElement previous() {
			return previousElement;
		}
		public Object value() {
			return data;
		}
		public DoubleLinkedListElement next() {
			return nextElement;
		}
		public void setNext(DoubleLinkedListElement value) {
			nextElement = value;
		}
		public void setPrevious(DoubleLinkedListElement value) {
			previousElement = value;
		}
	}
	private int count;
	private DoubleLinkedListElement head;
	private DoubleLinkedListElement tail;
	public DoubleLinkedList() {
		head = null;
		tail = null;
		count = 0;
	}
	public Object getFirst() {
		return head.value();
	}
	public Object getLast() {
		return tail.value();
	}
	public int size() {
		return count;
```

```
        }
        public void addFirst(Object value) {...}
        public void addLast(Object value) {...}
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode; }

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n, TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Write a method **`void removeDoubles()`** on the Vector class that traverses removes all duplicates in a vector. A duplicate is an element that occurs exactly twice in the vector.

For example, after calling **`removeDoubles()`** on the following vector:

[1, 2, 3, 3, 2, 4, 1, 5, 5, 5, 7]

will result in a vector containing the following elements:

[4, 5, 7]

The elements 1, 2 and 3 are completely removed as they occur exactly two times. The element 5 is kept as it occurs more than 3 times. The element 7 is kept as it occurs less than two times.

What is the time complexity of your algorithm?

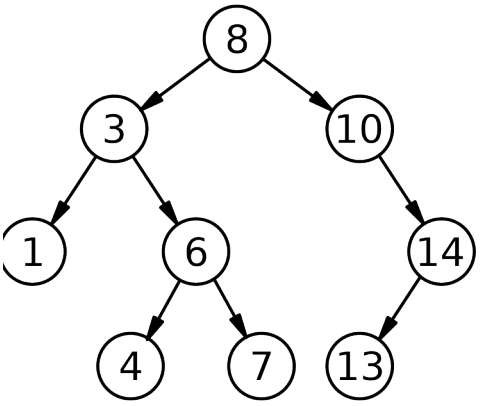
Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Trees (6 points)

A different way to store a binary search tree than the method we have seen in class, is to store all the data in a vector. Every element in the tree corresponds to a position in the array, according to a fixed rule. The root is stored in the array on position 1, and for every element stored at position i, the left child is stored at position 2*i and the right child is stored at position 2*i+1.

For instance, for the tree on the left, the corresponding vector is given on the right



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/	8	3	10	1	6		14			4	7			13

a. Implement a method SetLeftChild(int index, Comparable value) which takes the index in the vector of the current node and assigns the given value to the left child.
b. A method “swap” on a binary search tree mirrors the tree, i.e the left and right children are swapped. For instance, in the picture below, the tree and its swapped version are shown. Implement a method swap on a binary search tree stored in a vector as explained above. Use SetLeftChild and SetRightChild (you can assume this method exists).
c. What is the time complexity of your swap method. Motivate.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Assume a circular structure called ***Circular double linked list***. This structure is similar to Double linked list, but the elements are chained in a circle, meaning the predecessor of the first element is the last element and the successor of the last element is the first element. Write the methods addFirst(Object val), addLast(Object val); removeFirst(), removeLast() and find(Object val).

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: (5 points)

Consider a directed acyclic graph. A “transpose” of a graphs is the same graph, with all edges transposed, i.e. an edge from node ***i*** to node ***j*** is transposed into an edge from node ***j*** to node ***i***. Write a transpose method on a graph in edge list representation. Motivate the time complexity of the method.

Provided Code

Vector

```
public class Vector {
	protected Object data[];
	protected int count;

	public Vector(int capacity) {
		data = new Object[capacity];
		count = 0;
	}

	public int size() {
		return count;
	}

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

	public Object get(int index) {
		return data[index];
	}

	public void set(int index, Object obj) {
		data[index] = obj;
	}

	public boolean contains(Object obj) {
		for(int i=0;i<count;i++) {
			if(data[i] == obj) return true;
		}
		return false;
	}

	public void addFirst(Object item) { … }

	public void addLast(Object o) {
		data[count] = o;
		count++;
	}

	public Object getFirst() { … }

	public Object getLast() { … }

	public void removeLast() { .. }

	public void removeFirst() { … }

}
```

DoubleLinkedList

```
public class DoubleLinkedList {
	private class DoubleLinkedListElement {
		private Object data;
		private DoubleLinkedListElement nextElement;
		private DoubleLinkedListElement previousElement;
		public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous) {
			data = v;
			nextElement = next;
			previousElement = previous;
			if(nextElement != null) nextElement.previousElement = this;
			if(previousElement != null) previousElement.nextElement = this;
		}

		public DoubleLinkedListElement(Object v) { this(v,null,null); }

		public DoubleLinkedListElement previous() { return previousElement; }

		public Object value() { return data; }

		public DoubleLinkedListElement next() { return nextElement; }

		public void setNext(DoubleLinkedListElement value) { nextElement = value; }

		public void setPrevious(DoubleLinkedListElement value) { previousElement = value; }
	}
	private int count;
	private DoubleLinkedListElement head;
	private DoubleLinkedListElement tail;

	public DoubleLinkedList()  {
		head = null;
		tail = null;
		count = 0;
	}

	public Object getFirst() { return head.value(); }

	public Object getLast() { return tail.value(); }

	public int size() {  return count; }

	public void addFirst(Object value) {
	}
	public void addLast(Object value) {
	}
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode; }

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n, TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Write a method **`void removeDuplicates()`** on the Vector class that traverses the vector and (in-place) removes all duplicates in it without creating or using an additional vector (or any other collections).
For example, after calling **`removeDuplicates()`** on the following vector:

[1, 2, 3, 3, 2, 4, 1]

will result in a vector containing the following elements:

[1, 2, 3, 4]

What is the time complexity of your algorithm?

Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

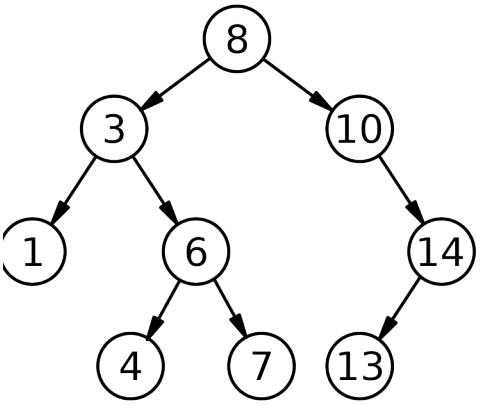
Exam Algorithms and Data Structures 30/01/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

a. Question 2/4: Trees (6 points)

A different way to store a binary search tree than the method we have seen in class, is to store all the data in a vector. Every element in the tree corresponds to a position in the array, according to a fixed rule. The root is stored in the array on position 1, and for every element stored at position i, the left child is stored at position 2*i and the right child is stored at position 2*i+1.

For instance, for the tree on the left, the corresponding vector is given on the right



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/	8	3	10	1	6		14			4	7			13

a. Implement a method SetLeftChild(int index, Comparable value) which takes the index in the vector of the current node and assigns the given value to the left child.
b. Two binary search trees are “element-equal” if they both store the same values, irrespective of the exact structure of the tree. Implement a method “element-equal” that checks whether both trees are storing exactly the same values. Implement this method on a binary search tree stored as a vector.
c. What is time complexity of your method? Motivate.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Assume a circular structure called ***Circular double linked list***. This structure is similar to Double linked list, but the elements are chained in a circle, meaning the predecessor of the first element is the last element and the successor of the last element is the first element. Write the methods addFirst(...), addLast(), removeFirst(), removeLast() and contains(Object val).

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: (5 points)

Consider a directed acyclic graph. A “transpose” of a graphs is the same graph, with all edges transposed, i.e. an edge from node ***i*** to node ***j*** is transposed into an edge from node ***j*** to node ***i***. Write a transpose method on a graph in adjacency matrix representation. Motivate the time complexity of the method.

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode; }

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n, TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 23/08/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (5 points)

Write a method **`void duplicate(int elem)`** on the Vector class that traverses the vector and (in-place) duplicates all values elem without creating or using an additional vector (or any other data structure).
For example, calling **`duplicate(3)`** on the following vector

[1, 2, 3, 2, 4, 3, 1]

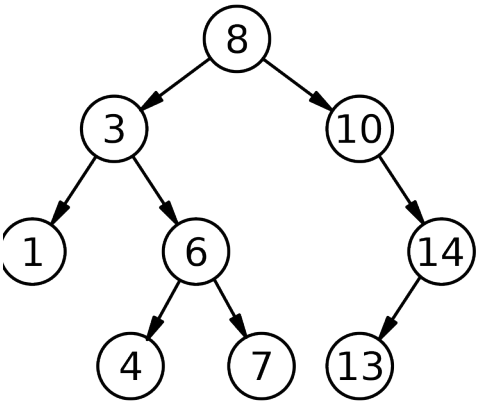
will result in a vector containing the following elements:

[1, 2, 3, 3, 2, 4, 3, 3, 1]

What is the time complexity of your algorithm? Motivate your answer.

Question 2/4: Trees (6 points)

Level order traversal is a traversal algorithm that visits tree nodes in order of depth, i.e. visiting first the root, then all the nodes from the second level (depth 1), etc. Write the level order traversal method that visits nodes based on the depth (top-down) and left to right. For instance, for this tree



the traversal order will be:

Level 1: 8

Level 2: 3, 10

Level 3: 1, 6, 14

Level 4: 4, 7, 13

What is the time complexity of your method? Motivate your answer.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4 (4 points)

Assume a circular structure called ***Circular double linked list***. This structure is similar to Double linked list, but the elements are chained in a circle, meaning the predecessor of the first element is the last element and the successor of the last element is the first element. Write the methods addFirst(...), addLast(), removeFirst(), removeLast() and contains(Object val).

Question 4/4: (5 points)

Consider a directed acyclic graph. A “transpose” of a graph is the same graph, with all edges transposed, i.e. an edge from node ***i*** to node ***j*** is transposed into an edge from node ***j*** to node ***i***. Write a transpose method on a graph in adjacency matrix representation. Give and motivate the time complexity of the method.

Exam Algorithms and Data Structures 23/08/2023, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Provided Code

Linked List

```
public class LinkedList implements Comparable{
  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }

  public String toString()
  {
    String result = "(";
    ListElement d = head;
    while(d != null)
    {
      result += d.first().toString();
      result += " ";
      d = d.rest();
    }
    result += ")";
    return result;
  }
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode; }

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n, TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }

   private Vector nodes;

   public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 28/08/2024, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: Vectors (4 points)

Assume a SortedCircularVector structure that works as a standard CircularVector, but with all elements sorted. Write a method **void remove(Comparable o)** that will remove all instances of the elements equal to **o** and will use binary search to identify elements that need to be removed. What is the time complexity of the method?

Exam Algorithms and Data Structures 28/08/2024, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Stack (5 points)

Implement the Rotate(int n, Rotate direction) method on Queue. The method takes the **n** topmost elements and moves them in the given direction (right or left).
E.g. if n = 3 and direction = right, the items on position 1, 2, 3 are moved to positions 2, 3, 1, i.e. the first item moves to the end.
If n = 3 and direction = left, the items 1, 2, 3 are moved to positions 3, 1, 2, i.e. the last item moves to the top.
The rest of the items in the stack remains at their positions. What is the time complexity of your Rotate method?

(1) apple
(2) banana ===right rotate==>
(3) cucumber

(2) banana
(3) cucumber
(1) apple

(1) cucumber
(2) banana ===left rotate==>
(3) apple

(3) apple
(1) cucumber
(2) banana

Exam Algorithms and Data Structures 28/08/2024, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4: Trees (6 points)

Let's define average depth of a binary search tree as the arithmetic average of depths of all leaves. Write a method that computes the exact value of the average depth in the BinarySearchTree class. What is the time complexity of your method?

Exam Algorithms and Data Structures 28/08/2024, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (5 points)

Under certain conditions the Bellman-Ford algorithm does not converge to the shortest path. Explain what these conditions are and explain what goes wrong in that case. Provide an example of a graph in which it fails and use that example to explain the problem.

Exam Algorithms and Data Structures 28/08/2024, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Provided Code

Linked List

```
public class Vector {
	protected Object data[];
	protected int count;

	public Vector(int capacity) { ...}
 public int size() { ... }
	public boolean isEmpty() { ... }
	public Object get(int index) { }
	public void set(int index, Object obj) { ... }
	public boolean contains(Object obj) {
		for(int i=0;i<count;i++) {
			if(data[i] == obj) return true;
		}
		return false;
	}
	public void addFirst(Object item) { ... }
	public void addLast(Object o) { ... }
	public Object getFirst() { ... }
	public Object getLast() { ... }
	public void removeLast() { .. }
public void removeFirst() { ... }
	public boolean binarySearch(int key){
		int start = 0;
		int end = count - 1;
		while(start <= end){
			int middle = (start + end + 1) / 2;
			if(key < data[middle]) end = middle -1;
			else if(key > data[middle]) start = middle + 1;
			else return true;
		}
		return false;
	}


}
public class CircularVector {
	private Object[] data;
	private int first;
	private int count;
	public CircularVector(int capacity) {
		count = 0;
		first = 0;
		data = new Object[capacity];
	}
	public int size() {...}
	public void addFirst(Object element) {...}
	public void addLast(Object element) {...}
	public Object getFirst() {...}
	public Object getLast() {...}
```

```
        public void removeFirst() {...}
        public void removeLast() {...}
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode, rightNode;
    public TreeNode(Comparable v) {
      value = v;
      leftNode = null;
      rightNode = null;
    }
    public TreeNode(Comparable v, TreeNode left, TreeNode right) {
      value = v;
      leftNode = left;
      rightNode = right;
    }
    public TreeNode getLeftTree() { return leftNode; }

    public TreeNode getRightTree() { return rightNode; }

    public Comparable getValue() { return value; }
  }
  protected TreeNode root;
  public Tree() {
    root = null;
  }
  public void traverseNode(TreeNode n, TreeAction action) {
     if(n != null) {
       if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
       action.run(n);
       if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
     }
   }
   public void traverseInOrder(TreeAction action) {
      traverseNode(root,action);
   }
 public void traverse(TreeAction action){
       Stack t = new Stack();
       t.push(root);
       while(!t.empty()) {
          TreeNode n = (TreeNode)t.pop();
          action.run(n);
          if(n.getLeftTree() != null) t.push(n.getLeftTree());
          if(n.getRightTree() != null) t.push(n.getRightTree());
       }
    }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable
   {
      private Comparable info;
      private Vector edges;

      public Node(Comparable label)
      {
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e)
      {
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel()
      {
         return info;
      }
   }

   private class Edge implements Comparable
   {
      private Node toNode;

      public Edge(Node to)
      {
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }
   private Vector nodes;
      public Graph()
   {
      nodes = new Vector();
   }

   public void addNode(Comparable label)
   {
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
   {
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 17/01/2025, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: LinkedList (4 points)

Write a method **`LinkedList extractBigger(Comparable o)`** on the LinkedList which will create a new linked list containing elements bigger than the given input parameter. The elements should be removed from the original (this) LinkedList. Make sure the method **`extractBigger(…)`** does not create any new ListElements, but only manipulates the elements from the original LinkedList. What is the time complexity of this method?

For instance, calling **`extractBigger(2)`** on the LinkedList with following elements:
LL1 : 1->2->3->1->2->4->1
will return a new LinkedList:
LL2 : 3->4
While changing the content of the original LL1 to:
LL1 : 1->2->1->2->1

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Queue (5 points)

Write a method **`void reverse()`** on the Queue, which will **recursively** reverse the queue's content.

For example, calling the **`reverse()`** method on a queue containing the following elements: [8, 7, 2, 5, 1], would result in a (reversed) queue: [1, 5, 2, 7, 8]

The **`reverse()`** method needs to be recursive, and you can only use the queue's push() and pop() methods.

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4: Trees (6 points)

Write a method **`void divisibleBy(int x)`** on the BinarySearchTree which will print all the numbers divisible by the given number **`x`**.

For example, **`divisibleBy(3)`** called on a BST containing elements: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, should print: “3, 6, 9”.
What is the time complexity of your method?

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (5 points)

Write a method `bool multiplePathsExist(Node start, Node end)` on an undirected graph that will return true if there is **more than one path** between the two nodes.

Provided Code

Linked List

```
public class LinkedList {
	private class ListElement {
		private Object el1;
		private ListElement el2;

		public ListElement(Object el, ListElement nextElement) {
			el1 = el;
			el2 = nextElement;
		}
		public ListElement(Object el) {
			this(el, null);
		}
		public Object first() {
			return el1;
		}
		public ListElement rest() {
			return el2;
		}
		public void setFirst(Object value) {
			el1 = value;
		}
		public void setRest(ListElement value) {
			el2 = value;
		}
	}

	private ListElement head;

	public LinkedList() {
		head = null;
	}
	public void addFirst(Object o) {
		head = new ListElement(o, head);
	}
	public Object getFirst() {
		return head.first();
	}
	public Object get(int n) {
		ListElement d = head;
		while (n > 0) {
			d = d.rest();
			n--;
		}
		return d.first();
	}
	public String toString() {
		String s = "";
		ListElement d = head;
		while (d != null) { ... }
		return s;
	}
}
```

Queue

```
	public class Queue<T extends Comparable<T>> {
	private Vector<T> data;

	public Queue() {
		data = new Vector<T>(10);
	}
	public void push(T o) {
		data.addLast(o);
	}
	public T pop() {
		T topEl = data.getFirst();
		data.removeFirst();
		return topEl;
	}
	public T top() {
		return data.getFirst();
	}
	public int size() {
		return data.size();
	}
	public boolean empty() {
		return data.size() == 0;
	}
	@Override
	public String toString() {
		return data.toString();
	}
}
```

Binary Tree

```
public class Tree {

   public class TreeNode implements Comparable {
      protected Comparable   value;
      protected TreeNode   leftNode, rightNode;
      public TreeNode(Comparable v) {
         value = v;
         leftNode = null;
         rightNode = null;
      }
      public TreeNode(Comparable v, TreeNode left, TreeNode right) {
         value = v;
         leftNode = left;
         rightNode = right;
      }
      public TreeNode getLeftTree() { return leftNode; }

      public TreeNode getRightTree() { return rightNode; }

      public Comparable getValue() { return value; }
   }
   protected TreeNode root;
   public Tree() {
      root = null;
   }
   public void traverseNode(TreeNode n, TreeAction action) {
       if(n != null) {
          if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
          action.run(n);
          if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
       }
    }
    public void traverseInOrder(TreeAction action) {
        traverseNode(root,action);
    }
  public void traverse(TreeAction action){
        Stack t = new Stack();
        t.push(root);
        while(!t.empty()) {
           TreeNode n = (TreeNode)t.pop();
           action.run(n);
           if(n.getLeftTree() != null) t.push(n.getLeftTree());
           if(n.getRightTree() != null) t.push(n.getRightTree());
        }
     }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable{
      private Comparable info;
      private Vector edges;

      public Node(Comparable label){
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e){
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel(){
         return info;
      }
   }

   private class Edge implements Comparable{
      private Node toNode;

      public Edge(Node to){
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }
   private Vector nodes;
   public Graph(){
      nodes = new Vector();
   }

   public void addNode(Comparable label){
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2){
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 17/01/2025, 09.00 - 12.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: LinkedList (4 points)

Write a method **`LinkedList duplicateBigger(Comparable o)`** on the LinkedList which will create a new linked list containing elements bigger than the given input parameter and all of them will be in the resulting list duplicated, i.e. will be there twice. The elements should be removed from the original (this) LinkedList. What is the time complexity of this method?

For instance, calling duplicate**`Bigger(2)`** on the LinkedList with following elements:
LL1 : 1->2->3->1->2->4->1
will return a new LinkedList:
LL2 : 3->3->4->4
While changing the content of the original LL1 to:
LL1 : 1->2->1->2->1

Question 2/4: Stack (5 points)

Write a method **`void reverse()`** on the Stack, which will **recursively** reverse the stack's content. You can adapt the parameters and/or return type of the recursive method.

For example, calling the **`reverse()`** method on a stack containing the following elements: [8, 7, 2, 5, 1], would result in a (reversed) stack: [1, 5, 2, 7, 8]

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4: Trees (6 points)

Write a method **`int deepestEven()`** on the BinarySearchTree which will return the depth of the deepest even number/element in a tree.

What is the time complexity of your method?

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (5 points)

Write a method `int numberOfPaths(Node start, Node end)` on a directed acyclic graph that will return the number of paths that exist between two nodes.

Provided Code

Linked List

```
public class LinkedList {
	private class ListElement {
		private Object el1;
		private ListElement el2;

		public ListElement(Object el, ListElement nextElement) {
			el1 = el;
			el2 = nextElement;
		}
		public ListElement(Object el) {
			this(el, null);
		}
		public Object first() {
			return el1;
		}
		public ListElement rest() {
			return el2;
		}
		public void setFirst(Object value) {
			el1 = value;
		}
		public void setRest(ListElement value) {
			el2 = value;
		}
	}

	private ListElement head;

	public LinkedList() {
		head = null;
	}
	public void addFirst(Object o) {
		head = new ListElement(o, head);
	}
	public Object getFirst() {
		return head.first();
	}
	public Object get(int n) {
		ListElement d = head;
		while (n > 0) {
			d = d.rest();
			n--;
		}
		return d.first();
	}
	public String toString() {
		String s = "";
		ListElement d = head;
		while (d != null) { ... }
		return s;
	}
}
```

Stack

```
public class Stack {

        private LinkedList data;

        public Stack()
        {
           data = new LinkedList();
        }

        public void push(Comparable o)
        {
           data.addFirst(o);
        }

        public Comparable pop()
        {
           if(data.empty()) return null;
           else return data.removeFirst();
        }

        public Comparable top()
        {
           return data.getFirst();
        }

        …
}
```

Binary Tree

```
public class Tree {

   public class TreeNode implements Comparable {
      protected Comparable   value;
      protected TreeNode   leftNode, rightNode;
      public TreeNode(Comparable v) {
         value = v;
         leftNode = null;
         rightNode = null;
      }
      public TreeNode(Comparable v, TreeNode left, TreeNode right) {
         value = v;
         leftNode = left;
         rightNode = right;
      }
      public TreeNode getLeftTree() { return leftNode; }

      public TreeNode getRightTree() { return rightNode; }

      public Comparable getValue() { return value; }
   }
   protected TreeNode root;
   public Tree() {
      root = null;
   }
   public void traverseNode(TreeNode n, TreeAction action) {
       if(n != null) {
          if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
          action.run(n);
          if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
       }
    }
    public void traverseInOrder(TreeAction action) {
        traverseNode(root,action);
    }
  public void traverse(TreeAction action){
        Stack t = new Stack();
        t.push(root);
        while(!t.empty()) {
           TreeNode n = (TreeNode)t.pop();
           action.run(n);
           if(n.getLeftTree() != null) t.push(n.getLeftTree());
           if(n.getRightTree() != null) t.push(n.getRightTree());
        }
     }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable{
      private Comparable info;
      private Vector edges;

      public Node(Comparable label){
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e){
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel(){
         return info;
      }
   }

   private class Edge implements Comparable{
      private Node toNode;

      public Edge(Node to){
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }
   private Vector nodes;
   public Graph(){
      nodes = new Vector();
   }

   public void addNode(Comparable label){
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2){
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Algorithms and Data Structures 27/08/2025, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 1/4: LinkedList (5 points)

Write a method **`LinkedList extractDuplicates()`** on the LinkedList which will create a new linked list containing all elements that the original list contains more than once. The redundant elements should be removed from the original (this) LinkedList while keeping the first instance in the original list. Make sure the method **`extractDuplicates()`** does not create any new ListElements, but only manipulates the elements from the original LinkedList. What is the time complexity of this method?

For instance, calling **`extractDuplicates()`** on the LinkedList with following elements:
LL1 : 1->2->3->1->2->4->1
will return a new LinkedList:
LL2 : 1->2->1
While changing the content of the original LL1 to:
LL1 : 1->2->3->4

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 2/4: Stack (5 points)

Write a method **`void insertAt(Object o, int n)`** on the Stack, which will insert object o at the position n from the bottom. Use a recursive approach where only stack's push(), top() and pop() methods can be used.

For example, calling the **`insertAt(42,3)`** method on a stack containing the following elements: [8, 7, 2, 5, 1], would result in a stack: [8, 7, 2, 42, 5, 1]

Exam Algorithms and Data Structures 27/08/2025, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 3/4: Trees (6 points)

We want to investigate whether a given tree contains another given tree as a subtree. We build the code for this in two steps.

A. Make a method boolean equalSubtree(TreeNode node1, TreeNode node2) which takes as arguments two given TreeNodes and verifies whether both are the same. Two TreeNodes are the same, when their data is the same and when both the left and right subtree are the same (recursive property).
B. In question A, assume the depth of node1 is X and the depth of node2 is Y. Give the timecomplexity of your equalSubtree method in terms of X and Y. Explain,
C. Now make the desired method in the Tree class: boolean containsSubtree(Tree anothertree) which verifies whether the given Tree has anotherTree as a subtree. Make use of the method from question A for this.

Exam Algorithms and Data Structures 27/08/2025, 13.00 - 16.00, Name:

You cannot use any material you bring yourself, but all relevant code is provided. Please do not forget to write your name on every sheet of paper.

Question 4/4: Graphs (4 points)

Let's assume that on a directed graph a node is called a ***source*** when the in-degree (number of incoming edges) is smaller than out-degree (number of outgoing edges). Write a method `int countSources()` on a directed graph that will return the number of ***sources*** in the graph. Assume the provided representation of the graph.

Provided Code

Linked List

```
public class LinkedList {
	private class ListElement {
		private Object el1;
		private ListElement el2;

		public ListElement(Object el, ListElement nextElement) {
			el1 = el;
			el2 = nextElement;
		}
		public ListElement(Object el) {
			this(el, null);
		}
		public Object first() {
			return el1;
		}
		public ListElement rest() {
			return el2;
		}
		public void setFirst(Object value) {
			el1 = value;
		}
		public void setRest(ListElement value) {
			el2 = value;
		}
	}

	private ListElement head;

	public LinkedList() {
		head = null;
	}
	public void addFirst(Object o) {
		head = new ListElement(o, head);
	}
	public Object getFirst() {
		return head.first();
	}
	public Object get(int n) {
		ListElement d = head;
		while (n > 0) {
			d = d.rest();
			n--;
		}
		return d.first();
	}
	public String toString() {
		String s = "";
		ListElement d = head;
		while (d != null) { ... }
		return s;
	}
}
```

Queue

```
public class Queue<T extends Comparable<T>> {
	private Vector<T> data;
	public Queue() {
		data = new Vector<T>(10);
	}
	public void push(T o) {
		data.addLast(o);
	}
	public T pop() {
		T topEl = data.getFirst();
		data.removeFirst();
		return topEl;
	}
	public T top() {
		return data.getFirst();
	}
	public int size() {
		return data.size();
	}
	public boolean empty() {
		return data.size() == 0;
	}
	@Override
	public String toString() {
		return data.toString();
	}
}
```

Stack

```
public class Stack {
	private LinkedList data;
	public Stack()
	{
	data = new LinkedList();
	}

	public void push(Comparable o)
	{
	data.addFirst(o);
	}

	public Comparable pop()
	{
	if(data.empty()) return null;
	else return data.removeFirst();
	}

	public Comparable top()
	{
	return data.getFirst();
	}
	…
}
```

Binary Tree

```
public class Tree {

   public class TreeNode implements Comparable {
      protected Comparable   value;
      protected TreeNode   leftNode, rightNode;
      public TreeNode(Comparable v) {
         value = v;
         leftNode = null;
         rightNode = null;
      }
      public TreeNode(Comparable v, TreeNode left, TreeNode right) {
         value = v;
         leftNode = left;
         rightNode = right;
      }
      public TreeNode getLeftTree() { return leftNode; }

      public TreeNode getRightTree() { return rightNode; }

      public Comparable getValue() { return value; }
   }
   protected TreeNode root;
   public Tree() {
      root = null;
   }
   public void traverseNode(TreeNode n, TreeAction action) {
       if(n != null) {
          if(n.getLeftTree() != null)  traverseNode(n.getLeftTree(),action);
          action.run(n);
          if(n.getRightTree() != null)                traverseNode(n.getRightTree(),action);
       }
    }
    public void traverseInOrder(TreeAction action) {
        traverseNode(root,action);
    }
  public void traverse(TreeAction action){
        Stack t = new Stack();
        t.push(root);
        while(!t.empty()) {
           TreeNode n = (TreeNode)t.pop();
           action.run(n);
           if(n.getLeftTree() != null) t.push(n.getLeftTree());
           if(n.getRightTree() != null) t.push(n.getRightTree());
        }
     }
}
```

Graph

```
public class Graph
{
   public class Node implements Comparable{
      private Comparable info;
      private Vector edges;

      public Node(Comparable label){
         info = label;
         edges = new Vector();
      }

      public void addEdge(Edge e){
         edges.addLast(e);
      }

      public int compareTo(Object o){ ... }

      public Comparable getLabel(){
         return info;
      }
   }

   private class Edge implements Comparable{
      private Node toNode;

      public Edge(Node to){
         toNode = to;
      }

      public int compareTo(Object o){ ... }
   }
   private Vector nodes;
   public Graph(){
      nodes = new Vector();
   }

   public void addNode(Comparable label){
      nodes.addLast(new Node(label));
   }

   public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2){
      Node n1 = findNode(nodeLabel1);
      Node n2 = findNode(nodeLabel2);
      n1.addEdge(new Edge(n2));
   }
}
```

Exam Programming Practicum 20/01/2012

- You cannot use any material you bring yourself, but all relevant code is provided after the questions.
- Please start every question on a new sheet.
- Good luck !!!

Question 1: Linked List (4 points)

a) Write a method that removes a given element from a LinkedList. You are NOT allowed to copy the elements into a new linked list.
b) Make a drawing that shows how pointers are updated.
c) What is the time complexity of this method? Explain why.
d) Is the remove method implemented on a double linked list more efficient than on a linked list? Explain why (not). You do not need to implement the remove method on double linked list.

ANSWER

```
public void remove(Comparable e)
{
        ListElement current = head;
        ListElement previous = null;
        for(int i=0; i<count; i++)
        {
                if (current.first().compareTo(e) == 0)
                {
                        if (i == 0)
                                head = current.rest();
                        else
                                previous.setRest(current.rest());
                        count--;
                        return;
                }
                previous = current;
                current = current.rest();
        }
}
```

This one does a single iteration over the list, so this one is O(n).
Time complexity for double linked list is identical.

Question 2: Vector (3 points)

Implement the addSorted method, which adds an element at the correct position in an already sorted vector.

ANSWER

```
private void addSorted(Comparable o)
{
        if(isEmpty())
                        data[0] = o;
        else
        {
                int i = count - 1;
                while((i>=0)&&(data[i].compareTo(o) > 0))
                {
                        data[i+1] = data[i];
```

```
                    i--;
                }
                data[i+1]=o;
            }
            count++;
        }
```

Question 3: Circular List (4 points)

a) Implement the reverse method on a circular list. HINT: make sure that your method does not loop forever.
b) Make a drawing that shows how pointers are updated.
c) What is the time complexity of this method? Explain why.

ANSWER

```
public void reverse()
{
        if(head == null) return;
        else if(head.rest() == null) return ;
        else
        {
                ListElement prev = null;
                ListElement current = head;
                ListElement next = head.rest();
                while(prev != tail)
                {
                        current.setRest(prev);
                        prev = current;
                        current = next;
                        next = current.rest();
                }
                tail = head;
                head = prev;
                tail.setRest(head);
        }
}
```

This one does a single iteration over the list, so this one is O(n).

Question 4: Tree Sort (6 points)

The tree sort algorithm is a simple method for sorting a linked list. In this question, we will develop a straightforward implementation for it. The tree sort algorithm consists of two steps:

1. You traverse all elements of your linked list and insert them into a binary search tree.
2. You create a new empty linked list and you traverse the binary search tree in the correct order to have the elements from small to big and you put each element you traverse in the new linked list.

a) What is the time complexity of this tree sort? Explain why.
b) Implement step 1 of the above tree sort algorithm.

c) Before you can implement the second step of the tree sort algorithm, you need to add a new traverse method to your binary search tree class, which can traverse a binary search tree, such that the elements are visited from the small to big. Call this method "traverseInOrder". This traverse method HAS to be a recursive one in this assignment. Do not use a stack or a queue for this !!!

d) Is it possible to implement this traverse with a stack or queue rather than using a recursive method? Explain why (not).

e) Use the traverseInOrder method of your tree to access every node from smallest to biggest and to add them into a new LinkedList.

ANSWER

a) The time complexity of tree sort is O(n log n). That is because of step1. In that step, you need to traverse the linked list O(n) and for each element, add it to a binary search tree O(log n) for each of the n elements, so step 1 takes O(n log n). Step 2 requires you to traverse the entire tree O(n) and add each element in the list O(1).

b)

```
public Tree list2Tree()
{
        Tree sortTree = new Tree();
        ListElement d = head;
        while(d!= null)
        {
                sortTree.insert(d.el1);
                d = d.cdr();
    }
    return sortTree;
}
```

c)

```
public void traverseNode(TreeNode n,TreeAction action)
{
        if(n != null)
        {
                if(n.getLeftTree() != null)
           traverseNode(n.getLeftTree(),action);
                action.run(n);
                if(n.getRightTree() != null)
           traverseNode(n.getRightTree(),action);
        }
}

public void traverseInOrder(TreeAction action)
{
        traverseNode(root,action);
}
```

d) No, this is actually not possible. The in order traversal requires that you process the left child, then do the action and then process the right child of any node. With a stack or a queue, it is possible to first process both left and right child and then perform the action, or to first perform the action and only then to process the children.

e)

```
public LinkedList treeSort()
{
        Tree sortTree = list2Tree();

        final LinkedList result = new LinkedList();
        sortTree.traverseInOrder(new TreeAction()
        {
                public void run(Tree.TreeNode n)
                {
                        result.addFirst(n.getValue());
                }
        });
        return result;
}
```

What we did not discuss in class is the fact that the result variable needs to be declared final in this case, because JAVA "Cannot refer to a non-final variable inside an inner class defined in another method". Do not mind about this, if you did not declare this variable as final, it was also ok.

Question 5: Graphs (3 points)

Consider the given adjacency list representation of a graph. You may assume the underlying linked list implements the remove method you wrote in question 1.

a) Write a method that removes a node from a graph.
b) What is the time complexity of this method? Explain why.

ANSWER

```
// Graph.Node
public void removeEdge(Edge e)
{
        edges.remove(e);
}

// Graph
public void removeNode(Comparable label)
{
        Node node = new Node(label);
        nodes.remove(node);
        for (int i=0; i<nodes.size(); i++)
        {
                Node aNode = (Node) nodes.get(i); // Not optimal
                aNode.removeEdge(new Edge(node));
        }
}
```

For every node we have to remove the given node from the edge list (O(n)), therefore, the time complexity for this method is O(n^2).

Provided Code

Linked List

```
public class LinkedList implements Comparable{

	class ListElement
	{
		private Comparable el1;
		private ListElement el2;

		public ListElement(Comparable el, ListElement nextElement)
		{
			el1 = el;
			el2 = nextElement;
		}

		public ListElement(Comparable el)
		{
			this(el,null);
		}

		public Comparable first()
		{
			return el1;
		}

		public ListElement rest()
		{
			return el2;
		}

		public void setFirst(Comparable value)
		{
			el1 = value;
		}

		public void setRest(ListElement value)
		{
			el2 = value;
		}
	}

	protected ListElement head;
	private int count = 0;

	public LinkedList()
	{
		head = null;
	}


	public void addSorted(Comparable o)
	{
		// an empty list, add element in front
		if(head == null) head = new ListElement(o,null);
		else if(head.car().compareTo(o) > 0)
		{
			// we have to add the element in front
			head = new ListElement(o,head);
		}
		else
		{
			// we have to find the first element which is bigger
			ListElement d = head;
			while((d.cdr() != null)&&(d.cdr().car().compareTo(o) < 0))
			{
				d = d.cdr();
			}
			ListElement next = d.cdr();
			d.setCdr(new ListElement(o,next));
		}
		count++;
	}

}
```

Vector

```
public class Vector implements Comparable
{
	protected Comparable data[];
	protected int count;

	public Vector(int capacity)
	{
		data = new Comparable[capacity];
		count = 0;
	}
```

```
}
```

Binary Search Tree

```
public class Tree {

	public class TreeNode implements Comparable {
		protected Comparablevalue;
		protected TreeNode	leftNode;
		protected TreeNode	rightNode;

		public TreeNode(Comparable v)
		{
			value = v;
			leftNode = null;
			rightNode = null;
		}

		public TreeNode(Comparable v, TreeNode left, TreeNode right)
		{
			value = v;
			leftNode = left;
			rightNode = right;
		}

		public TreeNode getLeftTree()
		{
			return leftNode;
		}

		public TreeNode getRightTree()
		{
			return rightNode;
		}

		public Comparable getValue()
		{
			return value;
		}

	}

	protected TreeNode root;

	public Tree()
	{
		root = null;
	}

	public void insert(Comparable element)
	{
		insertAtNode(element, root, null);
	}

	private void insertAtNode(Comparable element, TreeNode current, TreeNode parent)
	{
		// if the node we check is empty
		if (current == null)
		{
			TreeNode newNode = new TreeNode(element);
			// the current node is empty, but we have a parent
			if (parent != null)
			{
				// do we add it to the left?
				if (element.compareTo(parent.value) < 0)
				{
					parent.leftNode = newNode;
				}
				// or do we add it to the right?
				else
				{
					parent.rightNode = newNode;
				}
			}
			// the current node is empty and it has no parent, we actually have an empty tree
			else
				root = newNode;
		} else if (element.compareTo(current.value) == 0)
		{
			// if the element is already in the tree, what to do?
		}
		// if the element is smaller than current, go left
		else if (element.compareTo(current.value) < 0)
		{
			insertAtNode(element, current.getLeftTree(), current);
		}
		// if the element is bigger than current, go right
		else
			insertAtNode(element, current.getRightTree(), current);
	}
```

```
	public void traverse(TreeAction action)
	{
		QueueVector t = new QueueVector();
		//Stack t = new Stack();
		t.push(root);
		while(!t.empty())
		{
			TreeNode n = (TreeNode)t.pop();
			action.run(n);

			if(n.getLeftTree() != null) t.push(n.getLeftTree());
			if(n.getRightTree() != null) t.push(n.getRightTree());
		}
	}

}
```

Graph

```
public class Graph {

	public class Node implements Comparable
	{
		private Comparable info;

		protected LinkedList edges;

		public Node(Comparable l)
		{
			info = l;
			edges = new LinkedList();
		}

		public void addEdge(Edge e)
		{
			edges.addFirst(e);
		}

		public int compareTo(Object o)
		{
			Node n = (Node)o;
			return n.info.compareTo(info);
		}

		public Comparable getLabel()
		{
			return info;
		}

	}

	protected class Edge implements Comparable
	{
		public Node toNode;
		public Edge(Node to)
		{
			toNode = to;
		}

		public int compareTo(Object o)
		{
			Edge n = (Edge)o;
			return n.toNode.compareTo(toNode);
		}

		public Node getToNode()
		{
			return toNode;
		}

	}


	private LinkedList nodes;

	public Graph()
	{
		nodes = new LinkedList ();
	}

	public void addNode(Comparable label)
	{
		nodes.addFirst(new Node(label));
	}

	protected Node findNode(Comparable nodeLabel)
	{
		// ...
	}
```

```
	public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
	{
		Node n1 = findNode(nodeLabel1);
		Node n2 = findNode(nodeLabel2);
		n1.addEdge(new Edge(n2));
	}

}
```

Exam Programming Practicum 03/09/2012

- You cannot use any material you bring yourself, but all relevant code is provided after the questions.
- Please start every question on a new sheet.
- Good luck !!!

Question 1: Vectors, Linked Lists, Sorting (6 points)

Gnome Sort is an algorithm based on a technique used by gnomes to sort flowers by their height. Assume a row of *n* flowers where each flower has a position between *0* and *n-1*. A gnome at position *i* can see the flower at his current position, the previous flower (*i-1*) and the next flower (*i+1*).

Step by step, the gnome will move from position *0* to position *n-1*. At each step the gnome looks at the current flower and the next flower. If both flowers are in the right order, he moves on to the next flower. In case the flowers are not in the right order, he will switch both flowers and move back to the previous flower.

There are two exceptions: when the gnome cannot move to the previous flower, because he is in the beginning of the row, he will move on to the next position; if he cannot move to a next flower, because he is at the end of the row, his job is finalized.

a) Implement a method gnomeSort that sorts a Vector according to the given methodology.
b) Implement a method gnomeSort that sorts a Double Linked List according to the given methodology.
c) What is the time complexity of both methods? Motivate your answer.

Question 2: Stacks (4 points)

a) Provide an implementation of two stacks using a single vector. The stacks should only reach an overflow if the full capacity of the vector is used.
b) What is the time complexity of the push and pop operations? Is the time complexity the same for both stacks? Motivate your answers.

Question 2: Trees (4 points)

a) Add a method to the Tree class that checks whether it equals a second given tree.
b) What is the time complexity of this method? What are the best and worst-case scenarios? Motivate your answers.

Question 4: Graphs (6 points)

a) Add a method to the Graph class that returns a new graph where all edges are reversed. For example, an edge a->b in the input graph becomes b->a in the output graph.
b) Add another method that changes the directions of the edges of a graph in place, i.e. without creating a new graph.
c) What is the time complexity of both methods? Motivate your answers.

Provided Code

Vector

```
public class Vector implements Comparable
{
	protected Comparable data[];
	protected int count;

	public Vector(int capacity)
	{
		data = new Comparable[capacity];
		count = 0;
	}

	public Comparable get(int index)
	{
		return data[index];
	}

	public void set(int index, Comparable obj)
	{
		data[index] = obj;
	}

}
```

Double Linked List

```
public class DoubleLinkedList
{
	private class DoubleLinkedListElement
	{
		private Object data;
		private DoubleLinkedListElement nextElement;
		private DoubleLinkedListElement previousElement;

		public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous)
		{
			data = v;
			nextElement = next;
			previousElement = previous;

			if(nextElement != null) nextElement.previousElement = this;
			if(previousElement != null) previousElement.nextElement = this;
		}

		public DoubleLinkedListElement(Object v)
		{
			this(v,null,null);
		}

		public DoubleLinkedListElement previous()
		{
			return previousElement;
		}

		public Object value()
		{
			return data;
			}

		public DoubleLinkedListElement next()
		{
			return nextElement;
		}

		public void setNext(DoubleLinkedListElement value)
		{
			nextElement = value;
		}

		public void setPrevious(DoubleLinkedListElement value)
		{
			previousElement = value;
		}
	}

	private int count;
	private DoubleLinkedListElement head;
	private DoubleLinkedListElement tail;

	public DoubleLinkedList()
	{
		head = null;
		tail = null;
		count = 0;
	}

}
```

Tree

```
public class Tree {
	public class TreeNode implements Comparable
	{
		protected Comparable value;
		protected TreeNode leftNode;
		protected TreeNode rightNode;
		public TreeNode(Comparable v)
		{
			value = v;
			leftNode = null;
			rightNode = null;
		}

		public TreeNode(Comparable v, TreeNode left, TreeNode right)
		{
			value = v;
			leftNode = left;
			rightNode = right;
		}

		public TreeNode getLeftTree()
		{
			return leftNode;
		}

		public TreeNode getRightTree()
		{
			return rightNode;
		}

		public Comparable getValue()
		{
			return value;
		}

	}

	protected TreeNode root;

	public Tree()
	{
		root = null;
	}


}
```

Graph

```
public class Graph {

	public class Node implements Comparable
	{
		private Comparable info;

		protected LinkedList edges;

		public Node(Comparable l)
		{
			info = l;
			edges = new LinkedList();
		}

		public void addEdge(Edge e)
		{
			edges.addFirst(e);
		}

		public int compareTo(Object o)
		{
			Node n = (Node)o;
			return n.info.compareTo(info);
		}

		public Comparable getLabel()
		{
			return info;
		}
	}

	protected class Edge implements Comparable
	{
		public Node toNode;

		public Edge(Node to)
		{
			toNode = to;
		}

		public Node getToNode()
		{
			return toNode;
		}
	}



	private LinkedList nodes;

	public Graph()
	{
		nodes = new LinkedList ();
	}

	public void addNode(Comparable label)
	{
		nodes.addFirst(new Node(label));
	}

	public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
	{
		Node n1 = findNode(nodeLabel1);
		Node n2 = findNode(nodeLabel2);
		n1.addEdge(new Edge(n2));
	}
}
```

Exam Programming Practicum 28/01/2013

- You cannot use any material you bring yourself, but all relevant code is provided after the questions.
- Please start every question on a new sheet. Write your name on top of every sheet.
- Good luck!

Question 1: Vectors (3 points)

Develop and efficient in-place method void partitionEvenOdd() on the Vector class that partitions the vector in even and odd numbers. The algorithm must terminate with the original vector containing all its even elements proceeding by all its odd elements. For the vector [7 1 7 74 21 7 9 26 10] the result might for instance be [74 10 26 17 7 21 9 7] (other solutions possible). The method must be an in-place algorithm, which means that you may only use a constant memory space in addition to the vector. In practice, this means that you may not use another temporary vector.

Question 2: Stacks and Queues (3 points)

Provide an implementation of a Queue using two stacks. You cannot use any other data structures than the stacks. Assuming that the push and pop operations of the stacks have a time complexity of O(1), what is the time complexity of your queue's push and pop implementations?

Question 3: Binary Search Trees (4 points)

Implement a method that returns the successor of a TreeNode in a binary search tree (BST). A successor of a node *n* is defined as the smallest element *x* in the tree such that *x* is bigger than the value of *n* (or null if that does not exist). You may assume that the BST does not contain duplicate keys.

Question 4: Double Linked List (4 points)

The “fropple” method on double linked lists swaps every two consecutive elements in a double linked list. For example, the list [1 2 3 4 5 6] is froppled into [2 1 4 3 6 5]. This fropple operation can be performed in O(n) with a single iteration over the double linked list. Make a detailed drawing of how the fropple operator changes the pointers in a double linked list. As the order of changing pointers is crucial, please number them. Implement the fropple method on the double linked list class. Your code should not create a new double linked list; the current list should be modified! Traverse the list only once!

Question 5: Graphs (6 points)

A loop is a cycle of length 1, i.e. an edge from a node to itself.

a. Write a method that removes all loops from a directed graph in the adjacency matrix representation.
b. What is the time complexity of this operation?
c. Write a method that removes all loops from a directed graph in the adjacency list representation. Don’t forget to provide implementations of missing methods of underlying data structures.
d. What is the time complexity of this operation?

Provided Code

Linked List

```
public class LinkedList implements Comparable{

  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }


  public void addSorted(Comparable o)
  {
    // an empty list, add element in front
    if(head == null) head = new ListElement(o,null);
    else if(head.car().compareTo(o) > 0)
    {
      // we have to add the element in front
      head = new ListElement(o,head);
    }
    else
    {
      // we have to find the first element which is bigger
      ListElement d = head;
      while((d.cdr() != null)&&(d.cdr().car().compareTo(o) < 0))
      {
        d = d.cdr();
      }
      ListElement next = d.cdr();
      d.setCdr(new ListElement(o,next));
    }
    count++;
  }

}
```

Vector

```
public class Vector implements Comparable
{
  protected Comparable data[];
  protected int count;

  public Vector(int capacity)
  {
    data = new Comparable[capacity];
    count = 0;
  }
```

```
}
```

Binary Search Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }

    public TreeNode getRightTree()
    {
      return rightNode;
    }

    public Comparable getValue()
    {
      return value;
    }

  }

  protected TreeNode root;

  public Tree()
  {
    root = null;
  }

  public void insert(Comparable element)
  {
    insertAtNode(element, root, null);
  }

  private void insertAtNode(Comparable element, TreeNode current, TreeNode parent)
  {
    // if the node we check is empty
    if (current == null)
    {
      TreeNode newNode = new TreeNode(element);
      // the current node is empty, but we have a parent
      if (parent != null)
      {
        // do we add it to the left?
        if (element.compareTo(parent.value) < 0)
        {
          parent.leftNode = newNode;
        }
        // or do we add it to the right?
        else
        {
          parent.rightNode = newNode;
        }
      }
      // the current node is empty and it has no parent, we actually have an empty tree
      else
        root = newNode;
    } else if (element.compareTo(current.value) == 0)
    {
      // if the element is already in the tree, what to do?
    }
    // if the element is smaller than current, go left
    else if (element.compareTo(current.value) < 0)
    {
      insertAtNode(element, current.getLeftTree(), current);
    }
    // if the element is bigger than current, go right
    else
      insertAtNode(element, current.getRightTree(), current);
  }
```

```
  public void traverse(TreeAction action)
  {
    QueueVector t = new QueueVector();
    //Stack t = new Stack();
    t.push(root);
    while(!t.empty())
    {
      TreeNode n = (TreeNode)t.pop();
      action.run(n);

      if(n.getLeftTree() != null) t.push(n.getLeftTree());
      if(n.getRightTree() != null) t.push(n.getRightTree());
    }
  }

}
```

Double Linked List

```
public class DoubleLinkedList
{
  private class DoubleLinkedListElement
  {
    private Object data;
    private DoubleLinkedListElement nextElement;
    private DoubleLinkedListElement previousElement;

    public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous)
    {
      data = v;
      nextElement = next;
      previousElement = previous;

      if(nextElement != null) nextElement.previousElement = this;
      if(previousElement != null) previousElement.nextElement = this;
    }

    public DoubleLinkedListElement(Object v)
    {
      this(v,null,null);
    }

    public DoubleLinkedListElement previous()
    {
      return previousElement;
    }

    public Object value()
    {
      return data;
    }

    public DoubleLinkedListElement next()
    {
      return nextElement;
    }

    public void setNext(DoubleLinkedListElement value)
    {
      nextElement = value;
    }

    public void setPrevious(DoubleLinkedListElement value)
    {
      previousElement = value;
    }
  }

  private int count;
  private DoubleLinkedListElement head;
  private DoubleLinkedListElement tail;

  public DoubleLinkedList()
  {
    head = null;
    tail = null;
    count = 0;
  }

  public void addFirst(Object value)
  {
    head = new DoubleLinkedListElement(value,head,null);
    if(tail == null) tail = head;
    count++;
  }


  public void addLast(Object value)
  {
    tail = new DoubleLinkedListElement(value,null,tail);
```

```
      if(head ==null) head = tail;
      count++;
    }
}
```

Graph Matrix

```
public class MatrixGraph
{
  private Matrix data;
  private int nrNodes;

  public MatrixGraph(int nrNodes)
  {
    data = new Matrix(nrNodes);
    self.nrNodes = nrNodes;
  }

  public void addEdge(int from, int to, double w)
  {
    data.set(from, to, w);
  }

  public double getEdge(int from, int to)
  {
    return (Double)data.get(from, to);
  }

}
```

Graph Adjacency List

```
public class Graph {

  public class Node implements Comparable
  {
    private Comparable info;
    protected LinkedList edges;

    public Node(Comparable l)
    {
      info = l;
      edges = new LinkedList();
    }

    public void addEdge(Edge e)
    {
      edges.addFirst(e);
    }

    public int compareTo(Object o)
    {
      Node n = (Node)o;
      return n.info.compareTo(info);
    }

    public Comparable getLabel()
    {
      return info;
    }

  }

  protected class Edge implements Comparable
  {
    public Node toNode;
    public Edge(Node to)
    {
      toNode = to;
    }

    public int compareTo(Object o)
    {
      Edge n = (Edge)o;
      return n.toNode.compareTo(toNode);
    }

    public Node getToNode()
    {
      return toNode;
    }

  }

  private LinkedList nodes;

  public Graph()
  {
    nodes = new LinkedList ();
  }

  public void addNode(Comparable label)
  {
    nodes.addFirst(new Node(label));
  }

  protected Node findNode(Comparable nodeLabel)
  {
    // ...
  }

  public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
  {
    Node n1 = findNode(nodeLabel1);
    Node n2 = findNode(nodeLabel2);
    n1.addEdge(new Edge(n2));
  }

}
```

Exam Programming Practicum 02/09/2013

- You cannot use any material you bring yourself, but all relevant code is provided after the questions.
- Please start every question on a new sheet. Write your name on top of every sheet.
- Good luck!

Question 1: A mathematical puzzle

A well-known mathematical puzzle is the following one: suppose you have a row of 1000 coins. All coins are the same, they have two sides: up and down. Let us number the coins from 1 to 1000 to identify them. All coins are facing up. Now, you first flip every second coin, so this is coin 2,4,6, …, 1000. In a second step, you flip every third coin, so this is coin 3, 6, 9, … This way, you continue flipping coins, until in the very last step, only the last coin is flipped. The challenge in this puzzle is then to tell what coins are facing up after this procedure.

In this question, you will make a computer program that executes all steps form the puzzle. We will store coins in a Vector. Each coin is simply represented by a Boolean.

1. Make a method "initialize" that takes no arguments and returns a vector of 1000 coins, all facing up.
2. Make a method "solve" that takes the initialized vector as an argument, and solves the puzzle by simulating all steps of the puzzle.
3. (extra credit) Which coins are facing up at the end?

Question 2: Trees

1. Extend the representation of the Tree class, such that each node in the tree stores a pointer to its parent.
2. Make a method to convert a tree without such parent pointers (assume each parent field is null) into a tree with the appropriate parent pointers, without copying/rebuilding the tree.
3. Make a method that stores all ancestors of a given node in a linked list.

Question 3: Double linked list

1. Consider a double linked list that contains only numbers. Make a method that splits such a double linked list into two double linked lists: one double linked list that contains all odd elements, the other one containing only the even elements. Make sure your method does not copy the elements into new lists, but rather rearranges pointers.
2. What is the time complexity of this method? Motivate your answer.

Question 4: Mixed Questions

Answer each question with true or false and motivate your answer in a few sentences.

1. Using a double linked list, both a stack and a queue can be implemented with a time complexity of $O(1)$ for both the push and the pop method.
2. Appending two linked lists into a single linked list can be done in $O(1)$.
3. Summing all elements in a binary search tree can be done in $O(\log n)$.
4. Checking whether an edge between two given nodes exists in a directed acyclic graph cannot be done more efficient than $O(n^2)$.

Provided Code

Linked List

```
public class LinkedList implements Comparable{

  class ListElement
  {
    private Comparable el1;
    private ListElement el2;

    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }

    public ListElement(Comparable el)
    {
      this(el,null);
    }

    public Comparable first()
    {
      return el1;
    }

    public ListElement rest()
    {
      return el2;
    }

    public void setFirst(Comparable value)
    {
      el1 = value;
    }

    public void setRest(ListElement value)
    {
      el2 = value;
    }
  }

  protected ListElement head;
  private int count = 0;

  public LinkedList()
  {
    head = null;
  }


  public void addSorted(Comparable o)
  {
    // an empty list, add element in front
    if(head == null) head = new ListElement(o,null);
    else if(head.car().compareTo(o) > 0)
    {
      // we have to add the element in front
      head = new ListElement(o,head);
    }
    else
    {
      // we have to find the first element which is bigger
      ListElement d = head;
      while((d.cdr() != null)&&(d.cdr().car().compareTo(o) < 0))
      {
        d = d.cdr();
      }
      ListElement next = d.cdr();
      d.setCdr(new ListElement(o,next));
    }
    count++;
  }

}
```

Vector

```
public class Vector implements Comparable
{
  protected Comparable data[];
  protected int count;

  public Vector(int capacity)
  {
    data = new Comparable[capacity];
    count = 0;
  }
```

```
}
```

Binary Search Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
    }

    public TreeNode getRightTree()
    {
      return rightNode;
    }

    public Comparable getValue()
    {
      return value;
    }

  }

  protected TreeNode root;

  public Tree()
  {
    root = null;
  }

  public void insert(Comparable element)
  {
    insertAtNode(element, root, null);
  }

  private void insertAtNode(Comparable element, TreeNode current, TreeNode parent)
  {
    // if the node we check is empty
    if (current == null)
    {
      TreeNode newNode = new TreeNode(element);
      // the current node is empty, but we have a parent
      if (parent != null)
      {
        // do we add it to the left?
        if (element.compareTo(parent.value) < 0)
        {
          parent.leftNode = newNode;
        }
        // or do we add it to the right?
        else
        {
          parent.rightNode = newNode;
        }
      }
      // the current node is empty and it has no parent, we actually have an empty tree
      else
        root = newNode;
    } else if (element.compareTo(current.value) == 0)
    {
      // if the element is already in the tree, what to do?
    }
    // if the element is smaller than current, go left
    else if (element.compareTo(current.value) < 0)
    {
      insertAtNode(element, current.getLeftTree(), current);
    }
    // if the element is bigger than current, go right
    else
      insertAtNode(element, current.getRightTree(), current);
  }
```

```
  public void traverse(TreeAction action)
  {
    QueueVector t = new QueueVector();
    //Stack t = new Stack();
    t.push(root);
    while(!t.empty())
    {
      TreeNode n = (TreeNode)t.pop();
      action.run(n);

      if(n.getLeftTree() != null) t.push(n.getLeftTree());
      if(n.getRightTree() != null) t.push(n.getRightTree());
    }
  }

}
```

Double Linked List

```
public class DoubleLinkedList
{
  private class DoubleLinkedListElement
  {
    private Object data;
    private DoubleLinkedListElement nextElement;
    private DoubleLinkedListElement previousElement;

    public DoubleLinkedListElement(Object v, DoubleLinkedListElement next, DoubleLinkedListElement previous)
    {
      data = v;
      nextElement = next;
      previousElement = previous;

      if(nextElement != null) nextElement.previousElement = this;
      if(previousElement != null) previousElement.nextElement = this;
    }

    public DoubleLinkedListElement(Object v)
    {
      this(v,null,null);
    }

    public DoubleLinkedListElement previous()
    {
      return previousElement;
    }

    public Object value()
    {
      return data;
    }

    public DoubleLinkedListElement next()
    {
      return nextElement;
    }

    public void setNext(DoubleLinkedListElement value)
    {
      nextElement = value;
    }

    public void setPrevious(DoubleLinkedListElement value)
    {
      previousElement = value;
    }
  }

  private int count;
  private DoubleLinkedListElement head;
  private DoubleLinkedListElement tail;

  public DoubleLinkedList()
  {
    head = null;
    tail = null;
    count = 0;
  }

  public void AddFirst(Object value)
  {
    head = new DoubleLinkedListElement(value,head,null);
    if(tail == null) tail = head;
    count++;
  }


  public void AddLast(Object value)
  {
    tail = new DoubleLinkedListElement(value,null,tail);
```

```
    if(head ==null) head = tail;
    count++;

  }
}
```

Graph

```
public class Graph {

  public class Node implements Comparable
  {
    private Comparable info;
    protected LinkedList edges;

    public Node(Comparable l)
    {
      info = l;
      edges = new LinkedList();
    }

    public void addEdge(Edge e)
    {
      edges.addFirst(e);
    }

    public int compareTo(Object o)
    {
      Node n = (Node)o;
      return n.info.compareTo(info);
    }

    public Comparable getLabel()
    {
      return info;
    }

  }

  protected class Edge implements Comparable
  {
    public Node toNode;
    public Edge(Node to)
    {
      toNode = to;
    }

    public int compareTo(Object o)
    {
      Edge n = (Edge)o;
      return n.toNode.compareTo(toNode);
    }

    public Node getToNode()
    {
      return toNode;
    }

  }


  private LinkedList nodes;

  public Graph()
  {
    nodes = new LinkedList ();
  }

  public void addNode(Comparable label)
  {
    nodes.addFirst(new Node(label));
  }

  protected Node findNode(Comparable nodeLabel)
  {
    // ...
  }

  public void addEdge(Comparable nodeLabel1, Comparable nodeLabel2)
  {
    Node n1 = findNode(nodeLabel1);
    Node n2 = findNode(nodeLabel2);
    n1.addEdge(new Edge(n2));
  }

}
```

Exam Algorithms and Datastructures

Bart Jansen

2013-2014 9.00 - 12.00

1 Graphs - 5 points

The intersection between two directed acyclic graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be defined very intuitively as $G = G_1 \cap G_2 = (VE) = (V_1 \cap V_2, E_1 \cap E_2)$.

1. Make a method that takes two graphs as arguments and returns a third graph which is the intersection of both graphs. Assume that we are working with the adjacency list representation.
2. In the implementation you have given, what is the time complexity of finding $V = V1 \cap V2$?
3. Can the graph $G = G_1 \cap G_2$ as defined above (where both G_1 and G_2 are acyclic) have cycles? Explain why (not) in max 5 lines.

2 LinkedList - 5 points

Some textbooks advocate that the implementation of the linked list data structure requires a different approach than the one we have taken in this course: They argue that the first ListElement in the linked list should ALWAYS be a "dummy" element which is NEVER changed or removed. This way, the head in the linked list class never points to null; in case the list is empty it still points to the "dummy". The advocates of this approach claim the consequence of storing this dummy is much more elegant code. In this question we will develop such a linked list class.

1. Create a constructor which creates an empty linked list, already having the "dummy" first element.
2. Create a method "removeFirst" that removes the first element form a linked list, taking care that the dummy element is not removed or changed.

3 Sorting - 4 points

Consider a vector where all elements can only take any of two values (say 0 and 1). This vector can be sorted very efficiently. Implement a sorting algorithm that will sort the vector in linear time and in place under these conditions.

4 Binary Search Trees - 4 points

Implement a method that returns the largest and the second largest element from a binary search tree. What is the time complexity of this method?

5 Binary Search Trees - 2 points for a good idea, 2 extra if fully correct

The performance of the binary search tree in retrieving elements becomes suboptimal when the tree becomes unbalanced. An approach to overcome this problem could be that the entire binary search tree is copied to a new tree, once it becomes unbalanced. Describe the required steps to transfer the unbalanced tree into a balanced one. (hint: an intermediate data structure might be very useful). Make a method that takes a binary search tree and inserts each element in the new tree, which is then guaranteed to be balanced. What is the time complexity of this approach?

```
public class LinkedList
{
   private class ListElement
    {
         private Object el1;
         private ListElement el2;
         public ListElement(Object el, ListElement nextElement)
         {
              el1 = el;
              el2 = nextElement;
         }
         public ListElement(Object el)
         {
              this(el,null);
         }
         public Object first()
         {
              return el1;
         }
         public ListElement rest()
         {
              return el2;
         }
         public void setFirst(Object value)
         {
              el1 = value;
         }
         public void setRest(ListElement value)
         {
              el2 = value;
         }
    }

   private ListElement head;

    public LinkedList()
   {
      head = null;
   }
   public void addFirst(Object o){ ... }
   public Object getFirst(){ ... }
   public Object get(int n){ ... }
    ...

}
```

```
import java.util.Comparator;

public class Tree
{
   public class TreeNode implements Comparable
   {
         protected Comparable value;
         protected TreeNode leftNode;
         protected TreeNode rightNode;
         protected TreeNode parentNode;

         public TreeNode(Comparable v, TreeNode left, TreeNode right, TreeNode parent)
         {
              value = v;
              leftNode = left;
              rightNode = right;
              parentNode = parent;
         }
         public TreeNode(Comparable v)
         {
              this(v,null,null,null);
         }

       public TreeNode getLeftTree(){return leftNode;}
       public TreeNode getRightTree(){return rightNode;}
       public TreeNode getParent(){return parentNode;}
       public Comparable getValue(){return value;}

       public int compareTo(Object arg0)
       {
          TreeNode node2 = (TreeNode)arg0;
          return value.compareTo(node2.value);
       }
   }

    // the root of our tree
    private TreeNode root;

    public Tree()
    {
        root = null;
    }

    public void insert(Comparable element)
    {
        // here goes the code to insert an element
        // in the binary search tree
    }

    public boolean find(Comparable element)
    {
        // here goes the code the verify whether
        // an element is present in the tree
    }
 }

public abstract class TreeAction
{
    public abstract void run(TreeNode n);
}
```

```
public class Graph
{
    public class Node implements Comparable
    {
        private Comparable info;
        private Vector edges;

        public Node(Comparable label)
        {
            info = label;
            edges = new Vector();
        }

        public void addEdge(Edge e)
        {
            edges.addLast(e);
        }

        public int compareTo(Object o){ ... }

        public Comparable getLabel()
        {
            return info;
        }
    }

    private class Edge implements Comparable
    {
        private Node toNode;
        public Edge(Node to)
        {
            toNode = to;
        }

        public int compareTo(Object o){ ... }
    }

    private Vector nodes;

    public Graph()
    {
        nodes = new Vector();
    }

    public void addNode(Comparable label)
    {
        nodes.addLast(new Node(label));
    }

    protected Node findNode(Comparable nodeLabel)
    {
        return (Node) nodes.contains(new Node(nodeLabel));
    }

    public void addEdge(Comparable nodeLabel1,
                        Comparable nodeLabel2)
    {
        Node n1 = findNode(nodeLabel1);
        Node n2 = findNode(nodeLabel2);
        n1.addEdge(new Edge(n2));
    }
}
```

```
public class DoubleLinkedList
{
    private class DoubleLinkedListElement
    {
        private Object data;
        private DoubleLinkedListElement nextElement;
        private DoubleLinkedListElement previousElement;

        public DoubleLinkedListElement(Object v,
                                       DoubleLinkedListElement next,
                                       DoubleLinkedListElement previous)
        {
            data = v;
            nextElement = next;
            previousElement = previous;

            if (nextElement != null)
                nextElement.previousElement = this;
            if (previousElement != null)
                previousElement.nextElement = this;
        }

        public DoubleLinkedListElement(Object v)
        {
            this(v,null,null);
        }

        public DoubleLinkedListElement previous()
        {
            return previousElement;
        }

        public Object value()
        {
            return data;
        }

        public DoubleLinkedListElement next()
        {
            return nextElement;
        }

        public void setNext(DoubleLinkedListElement value)
        {
            nextElement = value;
        }

        public void setPrevious(DoubleLinkedListElement value)
        {
            previousElement = value;
        }
    }
```

```
    private int count;
    private DoubleLinkedListElement head;
    private DoubleLinkedListElement tail;

    public DoubleLinkedList()
    {
        head = null;
        tail = null;
        count = 0;
    }

    public Object getFirst()
    {
        return head.value();
    }

    public Object getLast()
    {
    return tail.value();
    }

    public int size()
    {
        return count;
    }

    public void addFirst(Object value)
    {
        head = new DoubleLinkedListElement(value, head, null);
        if (tail == null) tail = head;
        count++;
    }

    public void addLast(Object value)
    {
        tail = new DoubleLinkedListElement(value, null, tail);
        if (head == null) head = tail;
        count++;
    }

    public void print()
    {
        DoubleLinkedListElement d = head;
        System.out.print("(");
        while(d != null)
        {
            System.out.print(d.value() + " ");
            d = d.next();
        }
        System.out.println(")");
    }
}
```

Exam Programming Practicum 01/09/2014, 09.00 - 12.00

- You cannot use any material you bring yourself, but all relevant code is provided after the three questions.
- Please start every question on a new sheet.
- Good luck!!!

Question 1: True or False. (5 points)

For each of the statements below, indicate whether they are correct or not. No motivation required.

a. In the worst case, the time complexity of finding an element in a binary search tree containing N elements is O(N).
b. Any unbalanced tree of size n can be balanced in O(log n) rotations.
c. In a graph G=(V,E) if a vertex v in V is visited during level k of a breadth first search from source vertex s in V, then every path from s to v has a length of at most k.
d. DFS will take $O(V^2)$ time on a graph G=(V,E) represented as an adjacency matrix.
e. Suppose T is a Red-Black-Tree containing natural numbers. If S is exactly the same tree as T, except that each element has a value that is twice the value of the corresponding element in T, then S is also a Red-Black-Tree.

Question 2: Recursion (4 points)

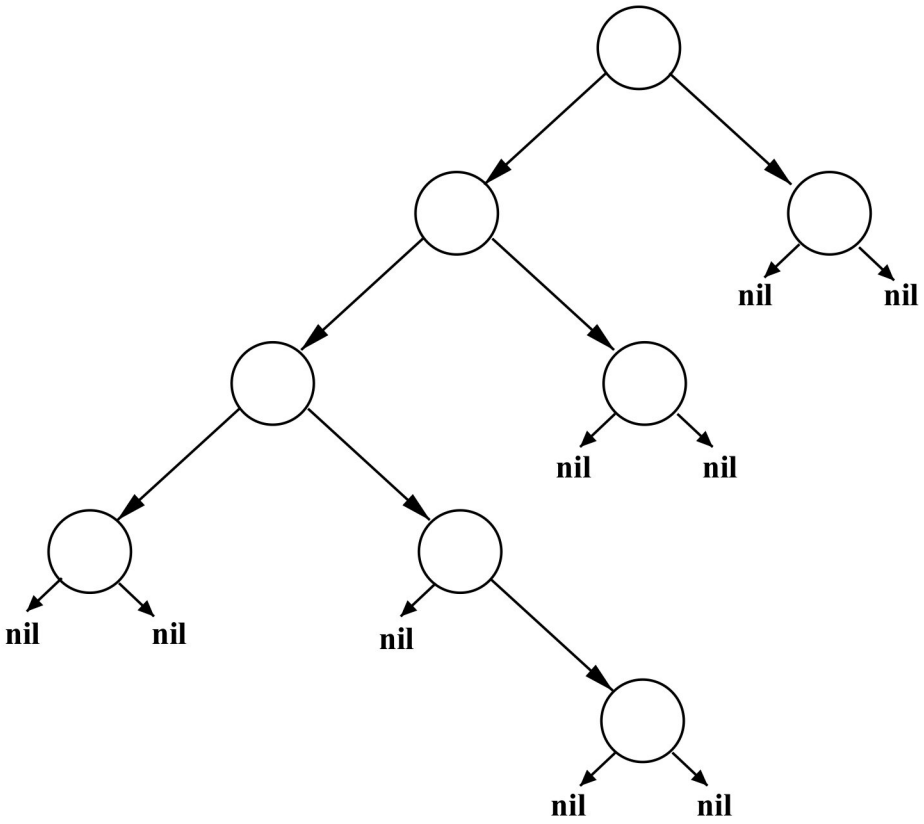
A number is an integer multiple of 9 if the sum of its digits is a multiple of 9. Hence, this is a recursive definition. Implement a method boolean isMultipleOf9(int number) to test for this.

You can use the following method that converts an integer number into a LinkedList of its digits: LinkedList integerToDigits(int number).

For example, integerToDigits(128) returns a linked list [1, 2, 8].

Your method can only return true or false if it gets a single digit number (or single element list) as input, otherwise it **should make a recursive call**. You can write multiple helper methods.

Question 3: Trees (7 points)



Consider the binary search tree given in the figure below.

a. Give the binary search tree with the above structure that contains the elements 2,3,5,7,11,13,17 and 19.
b. The nodes in this tree cannot be colored into a valid Red Black Tree. Give a proof for this statement.
c. By applying a single rotation to the tree, it can be converted into a valid Red Black Tree. Which one?
d. Implement a method that sums all the elements in a Red Black Tree (assume that the tree only contains integers).
e. What is the time complexity of such a sum method?

Question 4: Graphs (3 points)

If A = (V1,E1) and B = (V2,E2) are two unweighted DAG's and they have both the same set of vertices: V1=V2, implement A union B in the matrix representation.

Question 5: Linked Lists (6 points)

Since the elements of a linked list can be of any type, the elements can be Linked Lists as well. Consider for instance L1 = ((1,2),(3,4),(5,6)) to be a Linked List of 3 elements where the first element is a Linked List containing two elements: 1 and 2.

a. Write the code to create the Linked List L1.
b. Make a drawing showing the internal structure of the list.
c. Consider L1 as defined above and let L2 =((a,b),(c,d),(e,f)). The result of "merging" L1 with L2, is that L1 is now the list ((1,2,a,b),(3,4,c,d),(5,6,e,f)). Give the implementation of this "merging" function, it cannot create a new Linked List, but only change pointers in the existing lists. You can assume that both L1 and L2 have the same length. BUT: the elements in L1 and L2 can be lists when an arbitrary length.
d. What is the time complexity of the above method?

Provided Code

Linked List

```
public class LinkedList implements Comparable{


  class ListElement
  {
    private Comparable el1;
    private ListElement el2;


    public ListElement(Comparable el, ListElement nextElement)
    {
      el1 = el;
      el2 = nextElement;
    }


    public ListElement(Comparable el)
    {
      this(el,null);
    }


    public Comparable first()
    {
      return el1;
    }


    public ListElement rest()
    {
      return el2;
    }


    public void setFirst(Comparable value)
    {
```

```
    el1 = value;
  }

  public void setRest(ListElement value)
  {
    el2 = value;
  }
}

protected ListElement head;
private int count = 0;

public LinkedList()
{
  head = null;
}



public void addSorted(Comparable o)
{
  // an empty list, add element in front
  if(head == null) head = new ListElement(o,null);
  else if(head.car().compareTo(o) > 0)
  {
    // we have to add the element in front
    head = new ListElement(o,head);
  }
  else
  {
    // we have to find the first element which is bigger
    ListElement d = head;
    while((d.cdr() != null)&&(d.cdr().car().compareTo(o) < 0))
    {
      d = d.cdr();
```

```
        }
        ListElement next = d.cdr();
        d.setCdr(new ListElement(o,next));
      }
      count++;
    }

}
```

Binary Search Tree

```
public class Tree {

  public class TreeNode implements Comparable {
    protected Comparable   value;
    protected TreeNode   leftNode;
    protected TreeNode   rightNode;

    public TreeNode(Comparable v)
    {
      value = v;
      leftNode = null;
      rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left, TreeNode right)
    {
      value = v;
      leftNode = left;
      rightNode = right;
    }

    public TreeNode getLeftTree()
    {
      return leftNode;
```

```
  }

  public TreeNode getRightTree()
  {
    return rightNode;
  }

  public Comparable getValue()
  {
    return value;
  }

}

protected TreeNode root;

public Tree()
{
  root = null;
}

public void insert(Comparable element)
{
  insertAtNode(element, root, null);
}

private void insertAtNode(Comparable element, TreeNode current, TreeNode parent)
{
  // if the node we check is empty
  if (current == null)
  {
    TreeNode newNode = new TreeNode(element);
    // the current node is empty, but we have a parent
    if (parent != null)
```

```
      {
        // do we add it to the left?
        if (element.compareTo(parent.value) < 0)
        {
          parent.leftNode = newNode;
        }
        // or do we add it to the right?
        else
        {
          parent.rightNode = newNode;
        }
      }
      // the current node is empty and it has no parent, we actually have an empty tree
      else
        root = newNode;
    } else if (element.compareTo(current.value) == 0)
    {
      // if the element is already in the tree, what to do?
    }
    // if the element is smaller than current, go left
    else if (element.compareTo(current.value) < 0)
    {
      insertAtNode(element, current.getLeftTree(), current);
    }
    // if the element is bigger than current, go right
    else
      insertAtNode(element, current.getRightTree(), current);
  }
  public void traverse(TreeAction action)
  {
    QueueVector t = new QueueVector();
    //Stack t = new Stack();
    t.push(root);
    while(!t.empty())
```

```
    {
      TreeNode n = (TreeNode)t.pop();
      action.run(n);

      if(n.getLeftTree() != null) t.push(n.getLeftTree());
      if(n.getRightTree() != null) t.push(n.getRightTree());
    }
  }

}
```

Graph Matrix

```
public class MatrixGraph
{
  private Matrix data;
  private int nrNodes;

  public MatrixGraph(int nrNodes)
  {
    data = new Matrix(nrNodes);
    self.nrNodes = nrNodes;
  }

  public void addEdge(int from, int to, double w)
  {
    data.set(from, to, w);
  }

  public double getEdge(int from, int to)
  {
    return (Double)data.get(from, to);
  }

}
```