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(")");
    }
}
```