Question 1: *For each if the questions below, check true or false and give a short explanation. Without explanation, a question is considered unmarked.*

True [x] Balancing is only relevant on sorted trees.
False [ ] .

Reason: *The intention of balancing is to keep the data retrieval complexity at log n*

True [x] Tail recursion can easily be transformed into iteration.
False [ ]

Reason: *It can be simply replced by a while loop with the negated base case as loop condition*

True [ ] Logarithmic complexity is often achieved by using iteration rather than recursion.
False [x] .

Reason: *Complexity is an algorithm property. It cannot be changed by the implementation technique. Logarithmic complexity results from successive problem division (divide and coquer strategy)*

True [x] Graphs are usually stored as lists or matrices.
False [ ] .

Reason: *Most graph algorithms work on linear structures and would be hard to implement on linked structures*

True [x] Sorting a linked list has the complexity $O(n^2)$.
False [ ] .

Reason: *The successive division in halves which results in O(n log n) cannot be efficiently applied to linked lists*

True [x] The search complexity of Hash tables is constant, O(k).
False [ ] .

Reason: *Because the storage location is immediately computed from the data (or key, resp., for hash maps)*

True [x] A recursive method should make visible progress towards the base condition..
False [ ] .

Reason: *Endless recursion is the worst case. So termination should be made obvious.*

Question 2: Sorted Tree

1. Explain how to print a sorted list from a sorted (binary) tree. Does it matter whether the tree is balanced or not?

2. in order to remove a node from a sorted tree, it has to be replaced by its immediate predecessor, i.e. the rightmost node in its left subtree. For the class BinTree below, **implement the helper method findPredecessor recursively.**

```
class BinTree {
     class Node {
         public Comparable info;
         public Node left;
         public Node right;
     }

    protected Node root;

     private Node findPredecessor (Node node)
     { /* implement this method recursively */}
}
```

```
 private Node findPredecessor(Node root) {
       if (root == null || root.left == null) return null;
       return rightLeaf(root.left);
 }
```

```
        /* pre node != null  */
        private Node rightLeaf(Node node) {
              if (node.right == null) return node;
              return rightLeaf(node.right);
    }
```

Question 3: Indexed Tables

The class below is an Indexed Table, i.e. it consists of an unsorted list of (bulky) entries, and an index table of the same size which stores an order for the bulky list entries. The idea is to do sorting by moving lists of indeces rather than lists of bulky entries. Write the add method which adds an entry, keeping the index table sorted. For simplicity, we use a list of limited size, not a flexible one. Assume the method getEntryPoint() as given.

```
    class IndexedTable {

        private String[] entries = new  String[size];
        private int [] index = new int[size];

          /* add entry if not contained, keeping the index table sorted by insertion
          */
          public void add(String entry) throws FullExecption {
              // implement this method
          }

         /* returns the first index element pointing to a greater or equal entry
          */
          private int getEntryPoint(String newEntry) {
              // assume this method as given
          }
    }
```

```
  public class IndexedTable {

    int limit = 100;
    private String[] entries = new String[limit];
    private int[] index = new int[limit];
    private int size = 0;  // number of entries, next free slot

    public void add(String entry) throws FullException {
          if (size == limit)throw new FullException();
          int loc = getEntryPoint(entry);
          if (entries[index[loc]]==entry) return;  // is contained
          entries[size] = entry;   // store new entry at end of list
          for (int i=size-1; i>=loc; i--)
                index[i+1]=index[i]; // move greater elements down
          index[loc] = size;       // add to index table
          size++;
    }


    private int getEntryPoint(String newEntry) {
          for (int i=0; i<size; i++) {
                if (entries[index[i]].compareTo(newEntry) >= 0)
                                    // insertion point or contained
                      return i;
          }
          return size;
    }
}
```

What is the complexity of finding an entry?
search complexity in a sorted list, i.e. O(log n) if binsearch is applied

Question 4: LinkedLists

1 . Using the class SimpleNode to construct a singly linked list**, implement a class Stack** according to the Lifo interface.

```
interface Lifo <E>{
   public void push(E element);
    public E pop();
}

class SimpleNode <E> {
       public E info;
       public SimpleNode <E> next;
}
```

```
public class SimpleStack <E> implements Lifo <E>{

    private SimpleNode <E> top;

    public E pop() {
          if (top==null) return null;
          E result = top.info;
          top = top.next;
          return result;

    }

    public void push(E element) {
          SimpleNode <E> newnode = new SimpleNode<E>();
          newnode.info = element;
          newnode.next = top;
          top = newnode;
    }
}
```

2. Modify the Stack class so that it uses an anchor node, and adapt push and pop, if necessary.

```
public class AnchorStack <E> implements Lifo <E> {

    // anchor allows identical access to first or middle element in a list
    private SimpleNode <E> anchor = new SimpleNode <E> ();

    public E pop() {
          if (anchor.next == null) return null;
          E result = anchor.next.info;
          anchor.next = anchor.next.next;
          return result;
    }

    public void push(E element) {
          SimpleNode <E> newnode = new SimpleNode<E>();
          newnode.info = element;
          newnode.next = anchor.next;
          anchor.next = newnode;
    }
}
```