Skriftlig eksamen
Datastrukturer og algoritmer

Sommer 2001

Opgavesættet består af 3 opgaver, der ved bedømmelsen tillægges følgende vægte:

Opgave 1	50%
Opgave 2	30%
Opgave 3	20%

Alle sædvanlige hjælpemidler er tilladt. I tilfælde af unøjagtigheder i opgaveteksterne forventes det, at deltagerne selv præciserer besvarelsernes forudsætninger.

Opgavesættet består af 5 paginerede sider (incl. forsiden). Kontroller at din kopi er fuldstændig.

Opgave 1: Lister (50%)

Nedenfor ses et udkast til en Java-klasse, `List`, der implementer en liste af heltal.

```
public class List {
    public List() {
        head = tail = finger = new Node(0, null);
    }

    private class Node {
        Node(int v, Node n) { value = v; next = n; }

        int value;
        Node next;
    }

    private Node head, tail, finger;

    public void append(int v) {
        tail = tail.next = new Node(v, null);
        if (head.next == null)
            head.next = tail;
    }

    public boolean isEmpty() { /* spørgsmål 1.1 */ }
    public void reset() { finger = head; }
    public boolean more() { return finger.next != null; }
    public void advance() { finger = finger.next; }

    public int getValue() { return finger.next.value; }
    public void setValue(int v) { finger.next.value = v; }
    public void insert(int value) { /* spørgsmål 1.2 */ }
    public int remove() { /* spørgsmål 1.3 */ }
    public void sort() { /* spørgsmål 1.4 */ }
}
```

Listen er internt repræsenteret som en envejsliste af `Node`-objekter. Feltet `next` udpeger det efterfølgende `Node`-objekt i listen (`null`, hvis objektet er det sidste i listen).

Et særligt `Node`-objekt, `head`, udgør listens hoved, mens referencen `tail` peger det sidste `Node`-objekt i listen.

Metoden `append` tilføjer et nyt element bagerst i listen.

Spørgsmål 1.1 Programmér metoden `isEmpty`. Metoden skal returnere `true`, hvis listen er tom; ellers `false`.

Metoderne `reset`, `more` og `advance` benyttes til at gennemløbe en liste. Et kald af `reset` bevirker, at den aktuelle position sættes til den første position i listen. Et kald af `advance` flytter den aktuelle position frem til den efterfølgende. Et kald af `more` returnerer `true`, hvis og kun hvis den aktuelle position indeholder et listeelement.

Her er en kodestump, der bestemmer summen af en listes elementer:

```
int sum = 0;
for (list.reset(); list.more(); list.advance())
    sum += list.getValue();
```

For hvert element hentes dets værdi ved kald af metoden `getValue`.

Internt benyttes referencen `finger` til at pege på det `Node`-objekt, der er placeret på positionen umiddelbart før den aktuelle position. Derved forenkles programmeringen af metoderne `insert` og `remove`.

Spørgsmål 1.2 Programmér metoden `insert`, således at et kald, `insert(v)`, vil bevirke, at `v` indsættes i listen på den aktuelle position. Den aktuelle position flyttes ikke.

Spørgsmål 1.3 Programmér metoden `remove`, således at et kald vil bevirke, at elementet på den aktuelle position tages ud af listen, og dets værdi (`value`) returneres. Den aktuelle position flyttes ikke. Det er ikke nødvendigt at tage højde for, at metoden fejlagtigt kaldes, selvom den aktuelle position ikke indeholder et listeelement (`finger.next == null`).

Vi ønsker nu at tilføje en metode, `sort()`, til klassen `List`, der kan sortere listens elementer i stigende orden. Til det formål vil vi benytte følgende hjælpemetode:

```
private int removeMin() {
    Node prev = head, min = head.next;
    for (Node n = head.next; n.next != null; n = n.next)
        if (n.next.value < min.value)
          { prev = n; min = n.next; }
    prev.next = min.next;
    if (tail == min)
        tail = prev;
    return min.value;
}
```

Metoden fjerner det mindste element fra listen og returnerer dets værdi.

Spørgsmål 1.4 Programmer metoden `sort`.

Opgave 2: Binære søgetræer (30%)

Lad et binært søgetræ være repræsenteret ved hjælp af nedenstående to klasser:

```
class SearchTree {
    Node root;
}

class Node {
    Node left, right;
    int key;
}
```

Variablen `root` i klassen `SearchTree` udpeger træets rod. Ethvert tomt træ `Node`-refereres med referencekonstanten `null`.

Spørgsmål 2.1 Programmér en metode `printLeavesAscending` i klassen `SearchTree`, som udskriver nøglerne i træets blade i stigende orden.

Spørgsmål 2.2 Programmér en metode `printLeavesDescending` i klassen `SearchTree`, som udskriver nøglerne i træets blade i faldende orden.

Som bekendt kan rotationer benyttes til at balancere et søgetræ. Nedenfor ses en metode, der roterer en knude, `k2`, med sit venstre barn:

```
Node rotateWithLeftChild(Node k2) {
    Node k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

Spørgsmål 2.3 Antag at der i klassen `Node` indføres et ekstra felt, `Node dad`, som for enhver knude i træet skal referere til knudens far (`null`, hvis knuden er rod). Angiv de fornødne ændringer i metoden `rotateWithLeftChild`..

Opgave 3: Rekursion (20%)

Lad der være givet følgende klasse.

```
class Hocus {
    Hocus(int n) {
        a = new int[n];
        for(int i = 0; i < n; i++)
            a[i] = i + 1;
        pocus(0);
    }

    void pocus(int i) {
        if (i == a.length)
            write();
        else
            for (int j = i; j < a.length; j++) {
                swap(i, j);
                pocus(i + 1);
                swap(i, j);
            }
    }

    void write() {
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }

    void swap(int i, int j)
      { int t = a[i]; a[i] = a[j]; a[j] = t; }

    int a[];
}
```

Spørgsmål 3.1 Hvad udskrives ved udførelse af sætningen `new Hocus(3)`?