Vejledende løsninger

Opgave 1

Spørgsmål 1.1

Træet traverseres, og højre-venstre-hægterne byttes om undervejs.

```
void reflect(Node t) {
    if (t == null)
        return;
    reflect(t.left);
    reflect(t.right);
    Node temp = t.left; t.left = t.right; t.right = temp;
}
```

Alternativt kunne ombytningen være foretaget før de to rekursive kald.

Spørgsmål 1.2

```
Node reflection(Node t) {
    if (t == null)
        return null;
    return new Node(t.key, reflection(t.right), reflection(t.left));
}
```

Opgave 2

Spørgsmål 2.1

```
int minIndex(double[] a, int n) {
    int min = 1;
    for (int i = 2; i <= n; i++)
        if (a[i] < a[min])
            min = i;
    return min;
}
```

Der tages ikke højde for fejlsituationer: `(a == null || n < 1 || n > a.length)`.

Spørgsmål 2.2

```
int minIndex(double a[], int n) {
    int left = 1, right = n;
    while (left < right) {
        int middle = (left + right)/2;
        if (a[middle] < a[right])
            right = middle;
        else
            left = middle + 1;
     }
     return left;
}
```

Metoden kan alternativt programmeres ved brug af en rekursiv hjælpefunktion:

```
static int minIndex(double a[], int n) {
     return minIndexRec(a, 1, n);
}

int minIndexRec(double a[], int left, int right) {
    if (left == right)
        return left;
    int middle  = (left + right)/2;
    if (a[middle] < a[right])
        return minIndexRec(a, left, middle);
    return minIndexRec(a, middle + 1, right);
}
```

Opgave 3

Spørgsmål 3.1

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

Spørgsmål 3.2

Først argumenteres for, at hver gang løkkebetingelsen evalueres, gælder udsagnet

$$a[1] \leq a[2] \leq \ldots \leq a[i-1].$$

(1) Udsagnet er opfyldt første gang løkkebetingelsen evalueres.
Det ses let, idet $a[1] \leq a[1]$.

(2) Hvis udsagnet er opfyldt for et givet $i <= n$, vil det også være opfyldt næste gang løkkebetingelsen evalueres.

- Hvis $a[i] < a[i-1]$, foretages en ombytning af $a[i]$ og $a[i-1]$, hvorefter i mindskes med 1. Da ombytningen ikke har berørt nogen af elementerne i $a[1..i-1]$, gælder udsagnet.

- Hvis derimod $a[i] \geq a[i-1]$, så øges i med 1. Da $a[1] \leq a[2] \ldots \leq a[i-2]$ og $a[i-2] \leq a[i-1]$ medfører, at $a[1] \leq \ldots \leq a[i-1]$, gælder udsagnet også i dette tilfælde.

Hvis løkken terminerer, må i være lig med $n+1$ (da i maksimalt kan øges med 1 i hver iteration). I dette tilfælde gælder udsagnet $a[1] \leq a[2] \ldots \leq a[n]$. Elementerne $a[1]$, $a[2]$, ..., $a[n]$ vil altså være blevet sorteret i stigende orden.

Spørgsmål 3.3

I de tilfælde, hvor to elementer er placeret i faldende rækkefølge, mindskes i med 1. Men først efter, at de to elementer er ombyttet, så de står i stigende rækkefølge. Derfor kan i kun mindskes med 1 et endeligt antal gange. I alle andre tilfælde øges i med 1. Løkkebetingelsen, $i \leq n$, bliver derfor falsk efter et endeligt antal iterationer.

Spørgsmål 3.4

$O(n^2)$

Algoritmen svarer næsten til *sortering ved indsættelse*. Antallet af ombytninger er det samme, mens antallet af sammenligninger højst er det dobbelte.

Opgave 4

Spørgsmål 4.1

Nedenstående løsning benytter kun én hjælpemetode, nemlig `check(BinaryNode t)`, der – udover at checke om træet med rod t opfylder betingelse 1, 3 og 4 - returnerer antallet af sorte knuder på enhver vej fra `t` til blad.

Eventuelle fejludskrifter fås ved at kaste en `RuntimeException` med en beskrivende tekst som parameter.

```
public void check() {
    if (root != null) {
        if (root.color != BLACK)
            throw new RuntimeException("The root is not black");
        check(root);
    }
 }

int check(BinaryNode t) {
    if (t == null)
        return 0;
    if ((t.left  != null && t.left.element.compares(t.element)  >= 0) ||
        (t.right != null && t.right.element.compares(t.element) <= 0))
        throw new RuntimeException("The tree is not a search tree");
    if (t.color == RED &&
        ((t.left  != null && t.left.color  == RED) ||
         (t.right != null && t.right.color == RED)))
        throw new RuntimeException("Two consecutive red nodes on a path");
    int blacksOnLeftPath  = check(t.left);
    int blacksOnRightPath = check(t.right);
    if (blacksOnLeftPath != blacksOnRightPath)
        throw new RuntimeException
            ("Two paths with different number of black nodes");
    return t.color == RED ? blacksOnLeftPath : blacksOnLeftPath + 1;
}
```