Lösningsförslag för tentamen i Datastrukturer (DAT036) från 2012-12-18

Nils Anders Danielsson

1. När koden körs kommer talsekvensen 0, 1, 2, ..., $\mathtt{n}-1$ att sättas in i trädet. Eftersom sekvensen är sorterad kommer trädet att vara maximalt obalanserat, och insättningarna linjära i trädets storlek. Tidskomplexiteten blir

$$\Theta\left(\sum_{i=0}^{\mathtt{n}-1} i\right) = \Theta(\mathtt{n}^2).$$

2. (a) En möjlig lösning (skriven i ett språk som liknar Java):

```
public void reverse() {
    // En ny array med grannlistor.
    List<Integer>[] rev =
      new LinkedList<Integer>[nodes];

    // Till att börja med är grannlistorna tomma.
    for (int i = 0; i < nodes; i++) {
        rev[i] = new LinkedList<Integer>();
    }

    for (int i = 0; i < nodes; i++) {
        for (Integer j : adjacent[i]) {
            // Det finns en kant från i till j i
            // grafen, så låt oss lägga till en kant
            // från j till i i den reverserade grafen.
            rev[j].add(i);
        }
    }

    adjacent = rev;
}
```

   Algoritmen utför $O(1)$ arbete för varje nod och kant, så den är linjär i grafens storlek.

   (b) Ett förslag: Använd två arrayer med grannlistor, en för direkta efterföljare och en för direkta föregångare. Då kan man reversera grafen

genom att byta ut den ena arrayen mot den andra, och vice versa, på konstant tid (genom att ändra två pekare).

3. Jag gissar att många kommer att använda en hashtabell, men jag ger några andra lösningar.

   Låt M vara det maximala antalet hårstrån på ett huvud.

   En lösning: Sortera listan. Gå sedan igenom resultatet och avgör om två intilliggande värden är lika. Om vi använder radixsortering är tidskomplexiteten $O(d(N+n))$, där n är listans längd, N antalet olika siffror som används i talrepresentationen, och d antalet siffror i det största talet i listan. I vårt fall kan vi t ex välja $N = 2$,[1] vilket ger $d = O(\log M) = O(1)$ och den totala tidskomplexiteten $O(n) + O(n) = O(n)$.

   En annan lösning: Om listans längd är minst $M + 2$ så säger duvslagsprincipen att det finns två personer i listan med samma antal hårstrån på huvudet, så vi kan ge ”ja” som svar utan att inspektera talen. För kortare listor kan vi använda algoritmen ovan. Totala tidskomplexiteten blir $O(1)$ (med en stor ”konstant”).

4. *För trea:* Använd AVL-träd med par av nycklar och värden i noderna, och implementera `new`, `insert` och `member` som vanligt (de här operationerna har rätt tidskomplexitet). Låt `nth-smallest(`i`)` gå igenom trädet i inordning, och ge värdet i nod nummer i som svar; värstafallstidskomplexiteten för den här operationen är $O(n)$.

   För fyra: Använd AVL-träd med ett extra storleksfält i noderna, så att varje nod innehåller information om hur många noder som finns i delträdet för vilket noden är rot:

```
public class Node {
    public  int key, value;
    private int height, size;
    private Node left, right;

    // Konstruerare och/eller metoder (alla O(1)) som
    // säkerställer att height och size uppdateras på ett
    // korrekt sätt (så länge man inte introducerar cykler
    // i trädet).

    ...

    // Vänster barn.
    public Node left() {
        return left;
    }
```

[1] Kanske inte det bästa valet, men det duger här.

```
    // Höger barn.
    public Node right() {
        return right;
    }

    // Storleken av trädet rotat i n (som får vara null).
    public static int size(Node n) {
        return n == null ? 0 : n.size;
    }
}
```

Implementera `new`, `member` och `insert` på ungefär samma sätt som de vanliga AVL-trädsoperationerna (som har rätt tidskomplexiteter); vid insättning behöver man ibland ändra på storleksfälten, men om man använder ”rätt” implementationsteknik så kan det hanteras av de utelämnade `Node`-konstruerarna och -metoderna ovan. Implementera till sist `nth-smallest` på följande sätt:

```
public int nth-smallest(int i) {
    return nth-smallest(i, root);
}

// Hittar i-te noden (räknat från 1) i trädet rotat i n.
// Precondition: 1 <= i <= trädets storlek.
public int nth-smallest(int i, Node n) {
    assert 1 <= i && i <= Node.size(n);

    // Det vänstra delträdets storlek.
    int leftSize = Node.size(n.left());

    if (leftSize >= i) {
        return nth-smallest(i, n.left());
    } else if (leftSize + 1 == i) {
        return n.value;
    } else {
        return nth-smallest(i - (leftSize + 1), n.right());
    }
}
```

I värsta fallet är `nth-smallest` linjär i trädets höjd: $\Theta(\log n)$.

5. Se till exempel materialet från den föreläsning som gavs 2012-11-05.

6. Antag att en riktad, ändlig graf $G = (V, E)$ är given. Betrakta den underliggande acykliska grafen $G' = (V', E')$, där

$$V' = \{\, C \subseteq V \mid C \text{ består av noderna hos en SCC i } G \,\}$$

och

$$E' = \left\{ (c_1, c_2) \in V'^2 \;\middle|\; \exists v_1 \in c_1, v_2 \in c_2.\ (v_1, v_2) \in E \right\}.$$

Om G representeras på lämpligt sätt kan vi konstruera en grannlisterepresentation av G' (där varje nod representeras av ett tal, inte en delmängd av V) på linjär tid, $\Theta(|V| + |E|)$, med hjälp av en effektiv SCC-algoritm.

Betrakta nu följande fyra uttömmande fall:

- G' är tom: Då är G starkt sammanhängande.
- G' innehåller exakt en nod s utan ingående kanter och exakt en nod t utan utgående kanter: Då kan G göras starkt sammanhängande genom att lägga till som mest en kant från SCCn svarande mot t till SCCn svarande mot s.
- G' innehåller minst två noder utan ingående kanter: Då finns det två SCCer i G utan ingående kanter, så det krävs minst två kanter för att göra G starkt sammanhängande.
- G' innehåller minst två noder utan utgående kanter: Då krävs det också, på motsvarande sätt, minst två kanter för att göra G' starkt sammanhängande.

Vi kan alltså lösa problemet genom att beräkna antalet noder i G' utan ingående respektive utgående kanter, vilket om G' representeras med grannlistor kan göras på linjär tid ($O(|V'| + |E'|) = O(|V| + |E|)$).

Tidskomplexiteten blir $\Theta(|V| + |E|)$, vilket (åtminstone med en liten ”konstant”) nog får sägas vara effektivt.