Dr. I. Rogina (rogina@ira.uka.de)
Dr. C. Mayerl (mayerl@ira.uka.de)

Übungsblatt 9

Matrixmultiplikation

Abgabe bis: 21.06.2002, 12:00

Bonussystem: (*) Theorieaufgabe zum Korrigieren, (**) Programmieraufgabe zum Korrigieren

***Aufgabe 1:** *Matrixmultiplikation nach Winograd (2 Punkte)*
Multiplizieren Sie die beiden gegebenen Matrizen mit dem Algorithmus nach Winograd. Achten Sie darauf, dass der Rechenweg ersichtlich ist.

$$\begin{pmatrix} 3 & 4 & 2 & 6 \\ 3 & 5 & 7 & 5 \end{pmatrix} \begin{pmatrix} 1 & 8 \\ 3 & 4 \\ 6 & 4 \\ 3 & 2 \end{pmatrix}$$

***Aufgabe 2:** *Matrixmultiplikation nach Strassen (4 Punkte)*
Multiplizieren Sie die beiden angegebenen Matrizen mit dem Algorithmus nach Strassen. Achten Sie darauf, dass der Rechenweg ersichtlich ist.

$$\begin{pmatrix} 2 & 5 \\ 6 & 9 \end{pmatrix} \begin{pmatrix} 3 & 7 \\ 4 & 8 \end{pmatrix}$$

***Aufgabe 3:** *Boolsche Matrizen (3 Punkte)*
Wenden Sie den Algorithmus der vier Russen auf die beiden gegebenen Boolschen Matrizen an. Nehmen Sie dabei als Wortlänge k=2 an und geben Sie alle Zwischenschritte an.

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

****Aufgabe 4:** *Implementierung der Matrixmultiplikation (6 Punkte)*
Implementieren Sie die Matrixmultiplikation in JAVA jeweils einmal

- mit dem herkömmlichen Verfahren,
- nach Winograd.

Schreiben Sie auch eine Methode zur Ausgabe einer Matrix und testen Sie ihr Programm mit den Matrizen aus Aufgabe 1.

Übungsblatt 5 mit Musterlösungen

Lösung 1:

Teilaufgabe a)

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
----------------------------------------------------------------------
       2                    101   11    59              41      136
                            124   57   105              18       21
                                        82             110
```

Teilaufgabe b)

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13  14 15 16 17 18 19  20  21 22
---------------------------------------------------------------------------
       2                   101 124 11 57 59 105 82       41 18 110 136 21
                                1     1      1  2           1   2       1
```

Teilaufgabe c)

```
 0  1  2  3  4  5  6  7  8  9  10  11 12 13  14 15 16 17 18 19 20  21 22
---------------------------------------------------------------------------
       2   110             101 124  11 57 59 105       82 41 18     136 21
             3                  1      1      1         2     1          1
```

Teilaufgabe d)

```
 0   1  2  3  4  5  6 7   8   9  10  11 12 13 14 15 16 17 18 19 20  21 22
---------------------------------------------------------------------------
21 124 57        2   82 105 101      11    59 18          41 110   136
 2   1  1        1    1   1                    1              1
```

Teilaufgabe e)

Es ist problematisch, Einträge einfach so zu löschen, da unter Umständen Einträge, die bei der Kollisionsauflösung eingetragen wurden, nicht mehr auffindbar sind. Die Lösung des Problems besteht darin, dass man Einträge markiert (FREI, BELEGT, GELÖSCHT). Beim Suchen übergeht man einen als GELÖSCHT markierten Eintrag, beim Neueintrag kann an dieser Stelle eingetragen werden.

Lösung 2:

a)

70	2	23	57	9	56	95	72	59	89	
70	2	23	57	89	56	95	72	59	9	
70	2	23	72	89	56	95	57	59	9	
70	2	95	72	89	56	23	57	59	9	
70	89	95	72	9	56	23	57	59	2	
95	**89**	**70**	**72**	**9**	**56**	**23**	**57**	**59**	**2**	**Heap**

b)

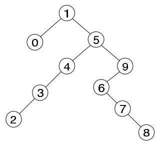
89	72	70	59	9	56	23	57	2	95	
72	59	70	57	9	56	23	2	95	89	
70	59	56	57	9	2	23	95	89	72	
59	57	56	23	9	2	95	89	72	70	
57	23	56	2	9	95	89	72	70	59	
56	23	9	2	95	89	72	70	59	57	
23	2	9	95	89	72	70	59	57	56	
9	2	95	89	72	70	59	57	56	23	
2	95	89	72	70	59	57	56	23	9	
95	**89**	**72**	**70**	**59**	**57**	**56**	**23**	**9**	**2**	**Sortiert**

Lösung 3:

a)

```
class Suchbaum {

  int elem;
  Suchbaum left, right;

  public Suchbaum(int k) {

    /**
     * create: int -> Suchbaum
     * @pre  true
     * @post (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     *        AND this.elem == k
     *        AND this.left == null AND this.right == null
     */

    elem  = k;
    left  = null;
    right = null;
  }

  public Suchbaum add(int k) {

    /**
     * add:  Suchbaum x int -> Suchbaum
     * @pre  (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     * @post (EXIST b: b.elem = k) AND
     *       (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     */

    Suchbaum s;

    if (k == elem) return this;

    if (k < elem) {

      if (left != null) left = left.add(k);
      else { s = new Suchbaum(k); left = s; }

    } else {

       if (right != null) right = right.add(k);
       else { s = new Suchbaum(k); right = s; }
    }

    return this;
  }

  boolean lookup(int k) {

    /**
     * lookup: Suchbaum x int -> Bool
     * @pre  (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     * @post (result = (EXIST b: b.elem = k)) AND
     *       (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     */

    if (k == elem) return true;

    if (k < elem && left != null) return left.lookup(k);
    else if (k > elem && right != null) return right.lookup(k);
    else return false;
  }

  public String postorder() {

    /**
     * postorder: Suchbaum -> String
     * @pre  (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     * @post (FORALL b, b': EXIST i, j: res_i = b.elem AND res_j = b'.elem
     *        AND i < j IMPLIES NOT b.lookup(b'.elem)) AND
     *       (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     */

    String res = "";

    if (left  != null) res += left.postorder();
    if (right != null) res += right.postorder();

    return res + elem + " ";
  }

  public String preorder() {

    /**
     * preorder: Suchbaum -> String
     * @pre  (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     * @post (FORALL b, b': EXIST i, j: res_i = b.elem AND res_j = b'.elem
     *        AND i < j IMPLIES (EXIST k, b'': k <= i < j AND
     *        b''.elem = res_k AND b''.lookup(b'.elem) AND
     *       (FORALL b: b.left != null IMPLIES b.elem > b.left.elem
     *        AND b.right != null IMPLIES b.elem < b.right.elem)
     */

    String res = elem + " ";

    if (left  != null) res += left.preorder();
    if (right != null) res += right.preorder();

    return res;
  }

  public static void main(String[] args) {

    Suchbaum st = new Suchbaum(1);

    st = st.add(5).add(9).add(6).add(7).add(8).add(4).add(0).add(3).add(2);
    System.out.println("Ist 6 im Suchbaum? " + st.lookup(6));
    System.out.println("Ist 10 im Suchbaum? " + st.lookup(10));
    System.out.println("Postorder: " + st.postorder());
    System.out.println("Preorder : " + st.preorder());
  }
}
```

b) Ergebnis: aufgebauter Suchbaum:



c) Ist 6 im Suchbaum? true
Ist 10 im Suchbaum? false
Postorder: 0 2 3 4 8 7 6 9 5 1
Preorder : 1 0 5 4 3 2 9 6 7 8

Lösung 4:

a) Am Anfang betrachten wir A(n) und B(m), also die kompletten Zeichenketten. Um unser Problem zu reduzieren, haben wir drei Möglichkeiten:

1. Wir löschen a_n aus a und betrachten danach nur noch A(n-1) und B(m)
2. Wir fügen b_m in A ein und betrachten danach nur noch A(n) und B(m-1) (b_m wird ganz ans Ende von A angehängt. Wir betrachten danach nur A(n), weil wir wissen, dass alle Zeichen rechts von a_n mit dem Ende von B übereinstimmen)
3. a) Falls $a_n \neq b_m$ ändern wir a_n in b_m
   b) Falls $a_n = b_m$ machen wir nichts
   In beiden Fällen betrachten wir danach nur noch A(n-1) und B(m-1)

Daraus ergibt sich direkt die rekursive Funktion C(i,j), die alle drei Möglichkeiten ausprobiert und die minimalen Kosten zurückgibt:

```
C(i,j) {
   if (i==0) return j;
   if (j==0) return i;
   c1 = C(i-1, j) + 1;                    // Löschen von a_i
   c2 = C(i, j-1) + 1;                    // Anhängen von b_j
   if (a_i == b_j) then c3 = C(i-1, j-1);
   else c3 = C(i-1, j-1) + 1;             // Ändern von a_i in b_j
   return min(c1, c2, c3);
}
```

Der Aufruf C(n,m) liefert dann die gesuchten minimalen Kosten. Der Algorithmus hat ***exponentiellen*** Aufwand.

b) Beim dynamischen Programmieren wird eine Tabelle angelegt, in die dann – ausgehend von bekannten Basiswerten – nach und nach alle Werte aus schon bekannten Werten berechnet werden. Einer der Tabellenwerte (im Normalfall der zuletzt berechnete) ist der gesuchte Zielwert.

c)

```
int[][] C = new int[n+1][m+1]
for (int i=0; i<=n; i++) C[i][0]=i;
for (int j=0; j<=m; j++) C[0][j]=j;

for (int i=1; i<=n; i++) {
   for (int j=1; j<=m; j++) {
      c1 = C[i-1][j] + 1;
      c2 = C[i][j-1] + 1;
      if (ai == bj) c3 = C[i-1][j-1];
      else c3 = C[i-1][j-1] + 1;
      C[i][j] = min(c1, c2, c3);
   }
}
return C[n][m];
```

Der Algorithmus hat ***O(n*m)*** Aufwand.

d) Die Tabelle sieht wie folgt aus:
Die gelben Markierungen zeigen einen möglichen Weg auf.

	B(0): „“	**B(1): „C“**	**B(2): „CH“**	**B(3): „CHA“**	**B(4): „CHAO“**	**B(5): „CHAOS“**
A(0): „“	0	1	2	3	4	5
A(1): „H“	1	1	1	2	3	4
A(2): „HA“	2	2	2	1	2	3
A(3): „HAS“	3	3	3	2	2	2
A(4): „HASE“	4	4	4	3	3	3