Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

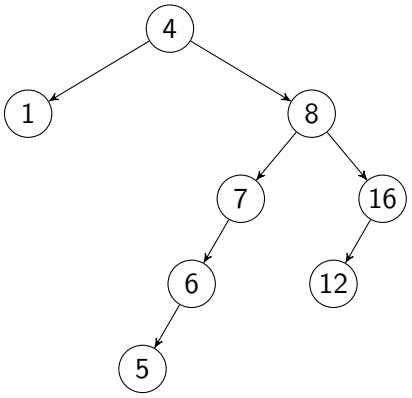
Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

23. März 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 4 FS 16

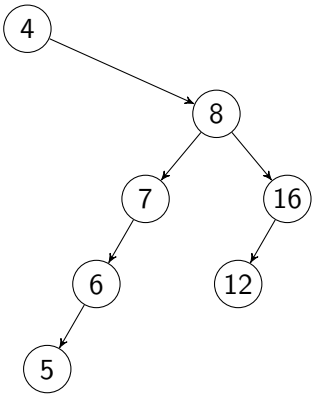
Lösung 4.1 *Suchbäume.*

a) Es ergibt sich der folgende Baum:

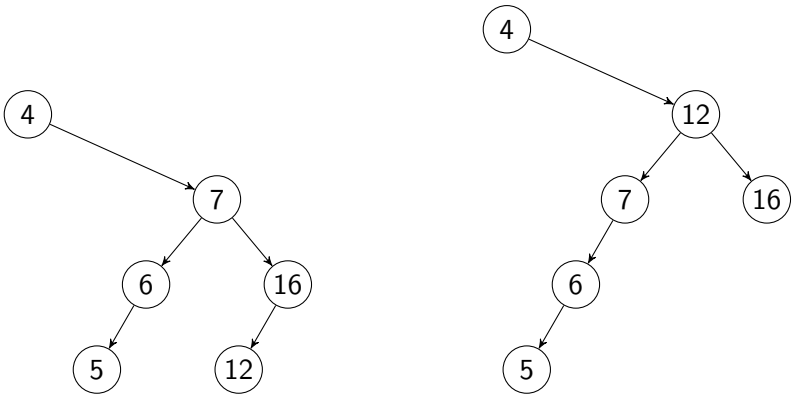


b)
- Preorder: $4, 1, 8, 7, 6, 5, 16, 12$
- Postorder: $1, 5, 6, 7, 12, 16, 8, 4$
- Inorder: $1, 4, 5, 6, 7, 8, 12, 16$

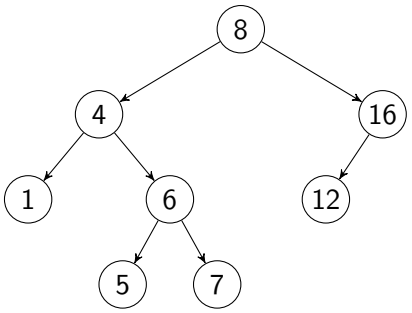
c) Schlüssel 1 ist in einem Blatt gespeichert und kann daher direkt gelöscht werden. Wir erhalten damit den folgenden Baum.



Um den Schlüssel 8 zu löschen, ersetzen wir zunächst 8 durch den Schlüssel des *symmetrischen Vorgängers* (d.h., durch den grössten Schlüssel kleiner als 8) oder durch den Schlüssel des *symmetrischen Nachfolgers* (d.h., durch den kleinsten Schlüssel grösser als 8) und löschen den entsprechenden Vorgänger- oder Nachfolgerknoten. Im ersten Fall entsteht der linke Baum, im zweiten der rechte Baum.



d) Es entsteht der folgende AVL-Baum:



Lösung 4.2 *Sortieralgorithmen.*

a) Eine Folge $f_1, f_2, \ldots, f_n$ bildet einen Min-Heap, falls $f_i \leq f_{2i}$ für alle i mit $2i \leq n$ und $f_i \leq f_{2i+1}$ für alle i mit $2i + 1 \leq n$. Für die sortierte Folge ist sogar $f_i \leq f_j$ für alle $j \geq i$, daher ist die sortierte Folge ein Min-Heap.

b) Wegen der Bedingung $f_i \leq f_{2i}$ für alle i mit $2i \leq n$, kann das grösste Element nur an Positionen i mit $2i > n$ stehen, also genau in der hinteren Hälfte des Arrays.

c) Die Verfahren *Sortieren durch Einfügen* und *Bubblesort* sind bereits in naiven Implementationen stabil. *Mergesort* ist stabil, wenn man beim Verschmelzen darauf achtet, bei zwei gleichen Elementen immer das weiter vorne stehende zu bevorzugen. Bei *Sortieren durch Auswahl*, *Quicksort* und *Heapsort* gibt es keinen einfachen Weg, Stabilität sicherzustellen.

Sortieren durch Auswahl, *Sortieren durch Einfügen*, *Bubblesort* und *Heapsort* arbeiten direkt auf dem zu sortierenden Array und sind daher in-situ. *Quicksort* benötigt zwischen $\Omega(\log n)$ und $\mathcal{O}(n)$ viel extra Speicher um die rekursiven Funktionsaufrufe im Speicher zu halten. Dieser Speicher wird nicht für die Folgenelemente verwendet, daher ist *Quicksort* ebenfalls in-situ. *Mergesort* muss wiederholt Teile des Arrays verschmelzen. Es gibt (komplizierte) Verfahren, das Verschmelzen in-situ auszuführen, aber keine, die durch simple Modifikation des Standardalgorithmus erreichbar sind.

Lösung 4.3 *Zweidimensionales Maximum Subarray Problem (**Programmieraufgabe**).*

Die einfachste Lösung für dieses Problem überprüft jede mögliche Teilmatrix, indem jeweils ein Eintrag für die obere linke und ein weiterer Eintrag für die untere rechte Ecke der Teilmatrix fixiert wird. Es gibt $\Theta(n^4)$ Teilmatrizen für eine $(n \times n)$-Matrix. Dann wird die Summe der Einträge berechnet, indem die einzelnen Einträge aufsummiert werden, und so das Maximum dieser Summen gefunden. Die Summe aller Einträge einer $(a \times b)$-Matrix kann in $\Theta(ab)$ Laufzeit

berechnet werden. Daher hat dieser Algorithmus eine Laufzeit in $\Theta(n^6)$ und kann die Testfälle der einfachsten Kategorie lösen.

Die Summe einer gegebenen Teilmatrix kann in konstanter Zeit berechnet werden, wenn wir folgende Information bereits vorausberechnet haben: Für eine Matrix d ist $ps[i][j]$ ist die Summe der Einträge der Teilmatrix mit der oberen linken Ecke $(1, 1)$ und der unteren rechten Ecke (i, j). Es gilt $ps[i][j] = ps[i-1][j]+ps[i][j-1]-ps[i-1][j-1]+d[i][j]$ (wir subtrahieren $ps[i-1][j-1]$, weil dieser Term doppelt gezählt wird). Wir können diese Informationen in $\Theta(n^2)$ Laufzeit einfach vorausberechnen. Nun kann die Summe einer Teilmatrix mit Koordinaten $(i1, j1)$ oben links und $(i2, j2)$ unten rechts wie folgt berechnet werden: $ps[i2][j2] - ps[i1 - 1][j2] - ps[i2][j1 - 1] + ps[i1 - 1][j1 - 1]$. Die Laufzeit dieses Algorithmus ist daher in $\Theta(n^4)$, womit die Testfälle der ersten beiden Kategorien gelöst werden können.

Um alle Testfälle zu lösen, muss das Problem auf das Maximum Subarray Sum-Problem aus der Vorlesung reduziert werden. Seien die erste und letzte Spalte einer gegebenen Teilmatrix $j1 \leq j2$. Wir können eine Teilmatrix, die durch diese beiden Spalten begrenzt ist, finden indem wir das Maximum Subarray Sum-Problem für das Array *sub* lösen, welches wie folgt definiert ist: $sub[i] = \Sigma_{j=j1}^{j=j2} d[i][j]$. Dieses Array kann wiederum mit *ps* vorberechnet werden in Laufzeit $\Theta(n^2)$. Nach der Vorberechnung benötigt der Algorithmus $\Theta(n^3)$ Laufzeit, wobei n^2 mögliche Spaltenpaare fixiert werden müssen und jeweils $\Theta(n)$ für das Lösen des entsprechenden Maximum Subarray Sum-Problem benötigt wird.

Implementierung

Im Folgenden finden Sie Implementierungen für der Algorithmen mit Laufzeit $\Theta(n^4)$ und $\Theta(n^3)$. Implementierung des $\Theta(n^4)$ Algorithmus:

```
int [][] partial_sums = new int [n+1][n+1];
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        partial_sums[i][j] = partial_sums[i][j-1]+partial_sums[i-1][j]-partial_sums[i-1][
            j-1]+d[i-1][j-1];
for (int i1=1;i1<=n;i1++){
    for (int j1=1;j1<=n;j1++){
        for (int i2=i1;i2<=n;i2++){
            for (int j2=j1;j2<=n;j2++){
                int cur = partial_sums[i2][j2]-partial_sums[i2][j1-1]-partial_sums[i1-1][
                    j2]+partial_sums[i1-1][j1-1];
                if (global_max<cur)
                    global_max = cur;
            }
        }
    }
}
```

Implementierung des $\Theta(n^3)$ Algorithmus:

```
import java.util.Scanner;

class Main{
    public static void main(String [] args){
        Scanner in = new Scanner(System.in);
        int test = in.nextInt();
        int i,j;
        for (int t=0;t<test;t++){
            int n = in.nextInt();
            int[][] d = new int[n][n];
            for (i=0;i<n;i++)
                for (j=0;j<n;j++){
                    d[i][j] = in.nextInt();
                }

            int global_max=0;
            int[][] partial_sums = new int [n][n];
            for (i=0;i<n;i++)
                partial_sums[i][0] = d[i][0];
            for (j=1;j<n;j++){
                for (i=0;i<n;i++){
                    partial_sums[i][j] = partial_sums[i][j-1]+d[i][j];
                }
            }
            for (int j1=0;j1<n;j1++){
                for (int j2=j1;j2<n;j2++){
                    int local_max=0;
                    int [] sub = new int [n];
                    for (i=0;i<n;i++){
                        if (j1>0) sub[i] = partial_sums[i][j2]- partial_sums[i][j1-1];
                            else sub[i] = partial_sums[i][j2];
                    }
                    int [] dp = new int [n];
                    dp[0] = sub[0];
                    if (local_max<dp[0]) local_max = dp[0];
                    for (i=1;i<n;i++){
                        if (dp[i-1]<=0)
                            dp[i] = sub[i]; else
                            dp[i] = dp[i-1]+sub[i];
                        if (local_max<dp[i]) local_max = dp[i];
                    }
                    if (global_max<local_max) global_max = local_max;
                }
            }
            System.out.println(global_max);
        }
    }
}
```