Algorithmendesign
Lösungen zu Übungsblatt 7
WS 13/14

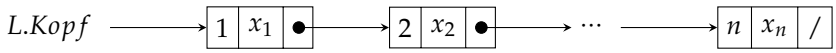
Gruppe 2
Max Bannach

Markus Richter (614027)

Aufgabe 7.1 Indizierte Listen

Teilaufgabe 1)

Die Implementierung einer *indizierten Liste* mithilfe verketteter Listen ergibt eine sortierte verkettete Liste. Dabei wird die lineare Reihenfolge der Liste durch die lineare Reihenfolge der Schlüssel definiert:



Das erste Feld enthält den Schlüssel, das zweite den Wert und das dritte den Zeiger *nachf* auf den Nachfolger. Element beschreibt im folgenden das gesamte Listenelement mit allen drei Feldern. Das Element aus der Aufgabenstellung entspricht also hier dem Wert im zweiten Feld. Gibt es keinen Nachfolger für das Element l, folgt daraus $l.nachf$ = NIL oder /. Das bedeutet l ist das Ende der Liste. Ein Attribut $L.kopf$ zeigt auf das erste Element der Liste. Wenn $L.kopf$ = NIL, so ist die Liste leer.

Sei n die Größe der Liste, also der letzte Index. Will man nun in der Liste suchen, so beginnt man bei $L.kopf$ und iteriert bis zum gesuchten Element l_k mit dem Schlüssel k und gibt den Wert des zweiten Feldes, also x_k zurück. Gibt es das Element nicht, so stößt die Suche nach $n < k$ Iterationen auf $l_i.nachf$ = NIL. Im schlimmsten Fall erfolgt also eine Iterationen über n Elemente, demnach beträgt die Laufzeit für **FIND**(k) $\mathcal{O}(n)$.

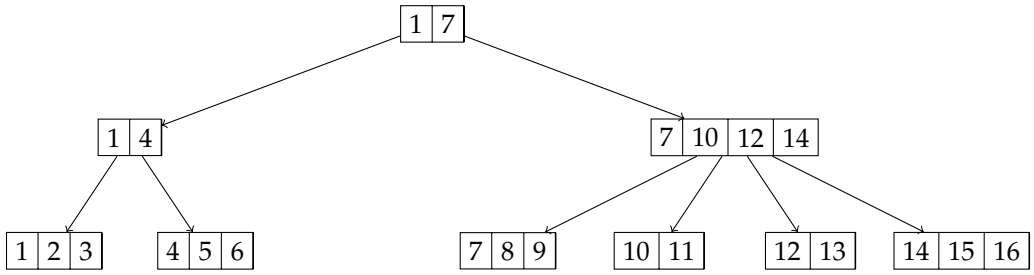
SEARCH(x) funktioniert analog, d. h. es erfolgt eine Iteration über die Elemente l_i der Liste, angefangen bei $L.kopf$. Der Unterschied besteht hier darin, dass l_i mit x nicht bzgl. des Schlüssels verglichen wird, sondern bzgl. des zweiten Feldes x_i. Wird ein Element gefunden kann ein *true* o. ä. zurückgegeben werden. Die Laufzeit beträgt $\mathcal{O}(n)$.

INSERT(k, x) iteriert bis zur Stelle $k-1$, ändert den Zeiger $l_{k-1}.nachf$ auf das neue Element l'_k, läuft nun von l_k bis l_n durch alle Elemente und inkrementiert deren Schlüssel um 1. Abschließend wird $l'_k.nachf$ auf das Element l_{k+1} gesetzt. Die Laufzeit beträgt damit auch $\mathcal{O}(n)$.

DELETE(k) iteriert bis zur Stelle k, setzt $l_{k-1}.nachf = l_{k+1}$. Iteriert nun von l_{k+1} bis l_n und dekrementiert die Schlüssel um 1.

Teilaufgabe 2)

Da es sich um eine indizierte Liste handelt bietet sich ein B^+*-Baum* an, der die inneren Knoten nach dem Index sortiert und nur auf den Blättern Werte enthält, d. h. alle Blätter b_i enthalten einen Zeiger auf den Wert x_i. Außerdem garantiert der B^+-Baum, dass alle Blätter auf einer Ebene liegen, der Baum also balanciert ist.



Sei $t \geq 2$ der minimale Grad des B$^+$-Baums. Dann hat jeder Knoten außer der Wurzel mindestens $t-1$ und maximal $2t-1$ Schlüssel und damit mindestens t bzw. $2t$ Kinder. Da auf jeder Ebene die Anzahl zu betrachtender Werte durch t geteilt wird, aber im schlimmsten Fall immer erst nach dem größten Schlüssel verzweigt wird entspricht die Laufzeit für **FIND**(k) $\mathcal{O}(t \log n)$.

SEARCH(x) macht im Grunde dasselbe wie oben, daher auch eine Laufzeit von $\mathcal{O}(t \log n)$.

Das Einfügen selbst würde ebenfalls logarithmisch ablaufen, da nur gesucht und eingefügt werden muss. Notfalls müssen Knoten verschmolzen oder aufgetrennt werden, was jedoch nicht allzu stark ins Gewicht fällt. Jedoch müssen hierbei alle nachfolgenden Indizes inkrementiert werden, was im schlimmsten Fall $n-1$ sein können, was für **INSERT**(k, x) zu einer Laufzeit von $\mathcal{O}(n)$ führt.

DELETE(k) ist dem Einfügen ähnlich. Auch hier würde das bloße Löschen nur logarithmische Laufzeit benötigen, durch das Ändern der Indizes ergibt sich aber eine Laufzeit von $\mathcal{O}(n)$.

Aufgabe 7.2 Quadtrees

Teilaufgabe 2)

siehe Anhang.

Teilaufgabe 2)

Man hat einen *uniformen Quadtree* $T = (V, E)$. Für die Kinder des Knotens gilt bzgl. der Größe der Fläche jeweils $n/2$. Die Struktur setzt sich rekursiv fort, daraus folgt, dass die Fläche rekursiv halbiert wird. Die Kernidee beim Arbeiten mit (x, y)-Koordinaten besteht nun darin, vorerst zu ermitteln im welchen Quadranten sich diese Koordinaten befinden. Möglich sind hier *North-West*, *North-East*, *South-East* und *South-West*.
Auf jeder Rekursionsstufe r ist es nötig zu wissen, was auf dieser Stufe die Hälfte in x- und y-Richtung ist, was hier, wegen der quadratischen Dimension, das Gleiche ist. Sei also die aktuelle Rekursionsstufe r und die Hälfte von x bzw. y nun k. Es gibt dann für x folgende Möglichkeiten:

1. $x > k$**:** East
2. $x \leq k$**:** West

Für y analog:

1. $x > k$**:** South
2. $x \leq k$**:** North

Aus der Kombination der beiden ergibt sich so eindeutig ein Quadrant und damit der gesuchte Kind-Knoten auf der Stufe r. Bevor man rekursiv fortfährt müssen nun die x, y-Koordinaten ggf. um k dekrementiert werden. Der rekursive Aufruf erfolgt solange bis gilt: $k = 1$. Dann ist der Wert nämlich der gesuchte Kind-Knoten selbst auf dieser Stufe. Es wird also immer ein Wert gefunden, der auch korrekt ist.

2.1)

Algorithm 1 2.1

Input: Quadtree $T = (V, E)$, Grösse n, Punkt $p(x, y)$, Wurzel $v \in V$
Output: Wert a

```
k ← n/2
while k ≠ 1 do
  if x > k then
    if y > k then
      hole Kante e ∈ E, e = (v, w), die South-East repräsentiert
      hole w und rufe dich rekursiv auf mit Input: T, k, p'(x − k, y − k), w
    else
      hole Kante e ∈ E, e = (v, w), die North-East repräsentiert
      hole w und rufe dich rekursiv auf mit Input: T, k, p'(x − k, y), w
    end if
  else
    if y > k then
      hole Kante e ∈ E, e = (v, w), die South-West repräsentiert
      hole w und rufe dich rekursiv auf mit Input: T, k, p'(x, y − k), w
    else
      hole Kante e ∈ E, e = (v, w), die North-West repräsentiert
      hole w und rufe dich rekursiv auf mit Input: T, k, p'(x, y), w
    end if
  end if
end while
Hole analog zu oben eine Kante e, nur das in w diesmal ein Wert ist.
a ← w
```

Korrektheit folgt aus der Beschreibung oben. **Laufzeit**: Die Anzahl der zu betrachtenden Werte wird pro Rekursion geviertelt, daraus folgt eine Laufzeit von $\mathcal{O}(\log n)$.

2.2)

Algorithm 2 2.2

Input: Quadtree $T = (V, E)$, Grösse n, Punkt $p(x, y)$, Wurzel $v \in V$, Wert α
  $k \leftarrow n/2$
  while $k \neq 1$ **do**
    if $x > k$ **then**
      if $y > k$ **then**
        hole Kante $e \in E, e = (v, w)$, die South-East repräsentiert
        hole w und rufe dich rekursiv auf mit Input: $T, k, p'(x - k, y - k), w$
      else
        hole Kante $e \in E, e = (v, w)$, die North-East repräsentiert
        hole w und rufe dich rekursiv auf mit Input: $T, k, p'(x - k, y), w$
      end if
    else
      if $y > k$ **then**
        hole Kante $e \in E, e = (v, w)$, die South-West repräsentiert
        hole w und rufe dich rekursiv auf mit Input: $T, k, p'(x, y - k), w$
      else
        hole Kante $e \in E, e = (v, w)$, die North-West repräsentiert
        hole w und rufe dich rekursiv auf mit Input: $T, k, p'(x, y), w$
      end if
    end if
  end while
  Hole analog zu oben eine Kante e, nur das in w diesmal ein Wert ist.
  $w \leftarrow \alpha$

Korrektheit folgt aus der Beschreibung oben. **Laufzeit**: Es geschieht im Grunde das Gleiche wie schon im Algorithmus davor, daher auch hier eine Laufzeit von $\mathcal{O}(\log n)$.

Teilaufgabe 3 Glück und Glas

1

Ist bursts$(\mathcal{A}) = 1$, so bleibt nichts anderes übrig als ganz unten anzufangen. Sei also $k = 1$ die gerade ausgewählte Sprosse der Leiter. Man wählt den Abstand $k \cdot 0,1$ Meter und lässt das Glas fallen. Falls es nicht zerbricht inkrementiert man k um 1 und versucht es nochmal. Das wiederholt man solange bis das Glas bricht. Für die Anzahl der maximalen Versuche gilt steps$(\mathcal{A}) = n$, da im schlimmsten Fall das Glas erst bei der höchsten Sprosse bricht, der Algorithmus aber von vorne alle Sprossen durchprobiert.

2

Bei einer binären Suchstrategie prüft der Algorithmus beginnend bei einer Sprosse rekursiv: Bricht das Glas für die aktuelle Sprosse? Wenn ja gehe zur kleineren Sprosse, wenn nicht, gehe zur größeren Sprosse
Es macht Sinn für die Wurzel $n/2$ zu wählen, um einen möglichst balancierten Baum zu erhalten und damit eine möglichst schnelle Laufzeit. Die maximale Anzahl der Schritte beträgt dann steps$(\mathcal{A}) = \log n$, allerdings gehen im schlimmsten Fall – bei einer sehr hohen Leiter – viele Gläser kaputt, nämlich nahezu bursts$(\mathcal{A}) = n/2$.

3

Eine mögliche Strategie ist mit $n/4$-Sprossen zu beginnen und jedes Mal, wenn das Glas nicht bricht um $n/4$ zu erhöhen. Wenn das Glas jedoch bricht, fängt man beim letzten Viertel, bei dem das Glas nicht zerbrochen ist, an und benutzt die Strategie aus Teilaufgabe 1, d. h. man geht von Sprosse zu Sprosse. Leider kann das Glas bereits beim ersten Viertel zerbrechen und damit hat man im schlimmsten Fall wieder $\text{steps}(\mathcal{A}) = n$:(