Bäume und der Sequence–ADT

Motivation: Der Sequence–ADT

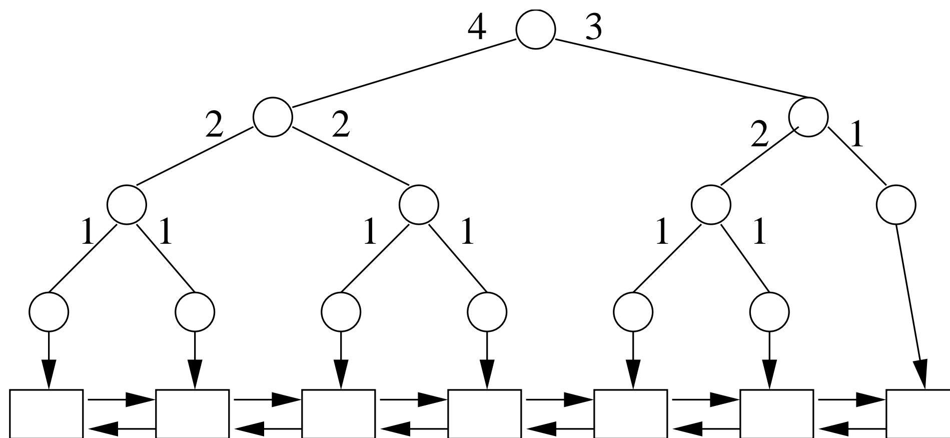
Bei der Vorstellung verschiedener Implementierungen für Stacks, Queues und Deques wurde vor allem auf die Unterschiede zwischen Arrays fester Größe, dynamiyschen Arrays (Vektoren) sowie einfach und doppelt verketteten Listen hingewiesen. Die Gemeinsamkeit all dieser Realisierungen lässt sich auf den Kern reduzieren, dass man mit Ihnen alle Elemente einer geordneten Kollektion in der gegebenen Reihenfolge durchlaufen kann. Diese Funktionalität wird durch das Java–Interface `Iterator<E>` beschrieben, das nur die Hauptmethoden `boolean hasNext()` und `E next()` hat. Ein Iterator `it` einer Kollektion liefert mit dem ersten Aufruf `it.next()` das erste Element der Kollektion und mit jedem weiteren Aufruf das jeweils nächste Element. Mit `it.hasNext()` kann vor jedem `it.next()` Aufruf geprüft werden, ob man schon beim letzten Element der Kollektion angekommen ist. Diese Methoden reichen bereits aus, um Pseudocode–Schleifenanweisungen der Form `for all x` $\in$ ... zu implementieren.
Allerdings kann man mit dem minimalen Konzept der Iteratoren noch keine Einfüge– oder Löschoperationen umsetzen. Solche Erweiterungen werden (in verschiedenen Ausprägungen) in den Java–Interfaces `Collection<E>, List<E>, ListIterator<E>` und `Set<E>` beschrieben.

Der Sequence–ADT, den wir im folgenden besprechen wollen, ist in dieser Form kein Java–Interface (es gibt eine Java–Klasse `Sequence`, aber darin geht es um musikalische Informationen im MIDI–System), am nächsten liegt das Java–Interface `List<E>`. In einer solchen Sequence (Folge) hier werden Daten eines Typs `E` in einer linearen Ordnung gehalten (also ein Iterator), aber der Zugriff auf die Daten kann an beliebÃger Stelle erfolgen. Dabei unterscheidet man zwischen der Position und dem Rang eines Elements. Unter der Position des Elements verstehen wir eine Referenz auf das Element. Dagegen ist der Rang die nummerierte Stelle des Elements in der Folge. Neben dem Zugriff auf ein Element über seine Position oder über seinen Rang sind Einfügeoperationen vor oder hinter einer gegebenen Position bzw. auf einem bestimmten Rang und Löschoperationen an einer Position oder auf einem bestimmten Rang gefordert.

Wie bereits bei Deques besprochen, kann dieser Datentyp durch dynamische Arrays oder durch doppelt verkettete Listen implementiert werden. Der einzige wesentliche Vorteil der Array-Implementierung liegt im Zugriff auf das Element mit Rang k, für den man bei der Listenimplementierung k next-Referenzen vom Anfangselement verfolgen muss. Beim Einfügen und Löschen mit Rang hat die Array-Implementierung zwar auch den Vorteil, das man unmittelbar auf die Stelle zugreifen kann, aber dafür mus der gesamte Array-Inhalt hinter dieser Stelle um eins nach rechts (beim Einfügen) oder um eins nach links (beim Löschen) verschoben werden. Die Vor- und Nachteile dieser Implementierungen werden in der folgenden Tabelle deutlich. Die Länge der Folge wird dabei mit n bezeichnet.

Operation	Laufzeit bei Array-Implementierung	Laufzeit bei Implementierung mit doppelt verk. Listen
Zugriff auf Position	$\Theta(1)$	$\Theta(1)$
Zugriff auf Rang	$\Theta(1)$	$\Theta(n)$
Einfügen mit Position	$\Theta(n)$	$\Theta(1)$
Einfügen mit Rang	$\Theta(n)$	$\Theta(n)$
Löschen mit Position	$\Theta(n)$	$\Theta(1)$
Löschen mit Rang	$\Theta(n)$	$\Theta(n)$

Für große Werte von n ist keine dieser Implementierungen wirklich befriedigend. Die Idee für eine effizientere Implementierung gründet sich zuerst auf die Beobachtung, dass sich alle Nachteile der Listenimplementierung auf einen Punkt fokussieren lassen: Das Finden der Position (Referenz) bei gegebenen Rang. Ist das geschehen, kann auch in konstanter Zeit eingefügt und gelöscht werden. Um den Zusammenhang zwischen Rang und Position algorithmisch schneller bearbeiten zu können, wird ein binärer Baum über die Liste gelegt, so dass jedes Blatt eine Referenz auf ein darunter liegendes Listenelement hält. Wenn man zusätzlich in jedem inneren Knoten vermerkt, wieviele Blätter der linke und der rechte Teilbaum enthalten, kann man leicht einen Algorithmus entwerfen, der bei gegebenem k von der Wurzel beginnend das k-te Blatt erreicht. Die Zeit ist proportional zur Tiefe des Baums, bei einem balancierten Baum $O(\log_2 n)$. Beim Einfügen und Löschen muss natürlich auch der Baum aktualisiert werden, aber das geht wieder in logarithmischer Zeit.



Das Hauptproblem besteht darin, dass bei einer ungünstigen Folge von Einfügeoperationen der Baum die Balanceeigenschaft verlieren und damit die Zeit auf $\Omega(n)$ anwachsen kann. Es gibt verschiedene Ansätze zur Lösung dieses Problems mit denen wir uns in den nächsten Vorlesungen beschäftigen werden.

Bäume mit Wurzeln (rooted Trees)

Definition: Ein **(gewurzelter) Baum** besteht aus einer Menge T von **Knoten**, die wie folgt strukturiert ist:

1. Es gibt genau einen hervorgehobenen Knoten $r \in T$, die *Wurzel* des Baums
2. Jeder Knoten außer r hat genau einen *Vaterknoten* (pc: *Elternknoten*)
3. Die Wurzel r ist *Vorfahr* jedes Knotens.

Dabei wird der Begriff $v \in T$ ist *Vorfahr* $u \in T$ rekursiv definiert, durch die Fälle

1. $u = v$ oder
2. v ist Vorfahr des Vaterknotens von u

Weitere Begriffe, die mit dieser Definition in Zusammenhang stehen, sind:

- u ist *Kind* von v genau dann, wenn v Vaterknoten von u ist.

- u ist *Nachkomme* von v genau dann, wenn v Vorfahr von u ist.
- u und v sind *Geschwister* genau dann, wenn sie den selben Vaterknoten haben.
- Knoten ohne Kinder heißen *Blätter* oder *äußere Knoten*.
- Knoten mit (mindestens) einem Kind heißen *innere Knoten*.

Wenn wir in diesem Teil der Vorlesung von einem Baum sprechen, meinen wir immer einen gewurzelten Baum. Die in der Graphentheorie definierten Bäume sind ungewurzelt, aber wie man bei BFS und DFS gesehen hat, kann man durch Festlegung eines speziellen Knotens ungewurzelte Bäume leicht in gewurzelte Bäume verwandeln.

Definition: Ein Baum ist ein *geordneter Baum*, wenn für jeden Knoten die Menge seiner Kinder geordenet ist (erstes Kind, zweites Kind,...).

Ein ADT für geordnete, gewurzelte Bäume

Die Beschreibung eines ADTs für gewurzelte Bäume weist große Ähnlichkeit zum Listen-ADT auf. Auch hier konzentriert sich die Beschreibung auf die Knoten. Zur Reprs̈entation des Baums reicht dann ein einzelner Knoten, nämlich die Wurzel aus. Ein Baumknoten muss die Referenzen auf den Elternknoten und die Kinder sowie auf ein Objekt vom Typ `E` (falls der Knoten Daten speichern soll) halten. Der Typ `TreeNode` wird durch die folgenden Methoden beschrieben:

```
TreeNode  parent();    //der Elternkonten
E  element();          //die Daten
TreeNode[] children(); //Liste der Kinder
boolean isRoot();
boolean isLeaf();
void  setElem(E e);
void  setParent(TreeNode v);
void  addChild(TreeNode v);   //dazu muss v setParent(this) ausfuehren
void  addSubtree(TreeNode v); //macht das Gleiche wie  addChild(TreeNode v)
```

Definition: Ist T ein Baum und $T' \subseteq T$ eine Untermenge der Knotenmenge, so nennen wir T' einen *Unterbaum* von T, wenn T' selbst ein Baum ist und für jeden Knoten $v \in T'$ auch alle Kinder von v zu T' gehören.

Wie man leicht sieht gibt es eine Bijektion (d.h. eine umkehrbare Abbildung) zwischen der Menge der Knoten und der Menge der Unterbäume von T. Einerseits kann man jedem Knoten v, den Unterbaum aller Nachfolger von v (inklusive v als Wurzel) zuordnen. Für die Umkehrabbildung ordnen wir jedem Unterbaum seine Wurzel zu.

Definition: Sei T ein Baum und $v \in T$ ein Knoten. Der Abstand von v zur Wurzel nennen wir die *Tiefe* von v und den Abstand von v zu seinem weitesten Nachfahren die *Höhe* von v. Die Höhe der Wurzel definiert die Höhe und gleichzeitig die Tiefe des Baums.
Beide Definitionen kann man rekursiv formulieren und entsprechend implementieren:

$$\text{Tiefe}(v) \quad = \quad \begin{cases} 0 & \text{falls } v \text{ die Wurzel ist} \\ 1 + \text{Tiefe}(\text{Vater von } v) & \text{sonst} \end{cases}$$

$$\text{Höhe}(v) \quad = \quad \begin{cases} 0 & \text{falls } v \text{ ein Blatt ist} \\ 1 + max\{\text{Höhe}(u) | u \text{ ist Kind von } v\} & \text{sonst} \end{cases}$$

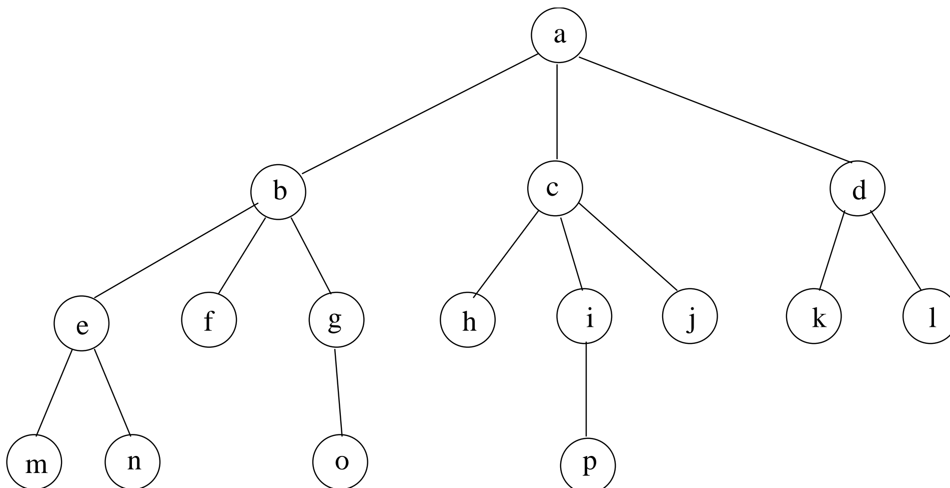
```
int depth(){   // als Methode von TreeNode
          int d=0;
          if(! isRoot()) d = 1 + parent().depth();
          return d;
}

int height(){   // als Methode von TreeNode
          int h=0;
          if(! isLeaf()){
                 for(int i=0; i< children().length; i++)
                          if(h < (children()[i]).height()){
                              h= (children()[i]).height();
                          }
                 h++;
          }
          return h;
}
```

Die Laufzeit der Tiefenberechnung ist proportional zur Tiefe, die Laufzeit der Höhenberechnung ist proportional zur Größe des Unterbaums.

Ähnlich wie bei verketteten Listen wird ein Baum nur durch seine Wurzel repräsentiert. Die vollständige Struktur ergibt sich erst durch die Verkettung von Referenzen. Da diese Struktur komplexer als bei verketteten Listen ist, spielen Methoden zum Durchlaufen von Bäumen (tree traversals) eine wichtige Rolle. Zwei dieser Durchlaufmethoden kann man für beliebige Bäume anwenden: *Preorder-* und *Postorder-Traversierungen*. Sie laufen nach den folgenden Regeln ab:

- **Preorder:** Für jeden zu durchlaufenden Teilbaum besuche zuerst die Wurzel und durchlaufe dann nacheinander die Teilbäume aller Kinder.
- **Postorder:** Für jeden zu durchlaufenden Teilbaum durchlaufe nacheinander die Teilbäume aller Kinder und zum Schluss die Wurzel.

Beispiel:



Für den abgebildeten Baum ergeben sich die Knotenfolgen

- $a,b,e,m,n,f,g,o,c,h,i,p,j,d,k,l$ beim Preorder-Durchlauf,
- $m,n,e,f,o,g,b,h,p,i,j,c,k,l,d,a$ beim Postorder-Durchlauf.