Datenstrukturen – Bäume

by

Dr. Günter Kolousek

Allgemeines

- verallgemeinerte Listenstruktur…
- Grundlage in Graphentheorie (Knoten, Kanten)
- Charakterisierung
  - genau ein Anfangsknoten (Wurzel)
  - Jeder Knoten (außer die Wurzel): genau einen Vorgänger (Vater, Elternknoten)
  - Jeder Knoten (außer Endknoten): mindestens ein Nachfolger (Kindknoten)
  - Bäume ‘wachsen’ in der Informatik von oben nach unten!
- Anwendungen
  - Darstellung logischer Beziehungen (Dateihierarchie, Syntaxbaum)
  - Suchen und Sortieren (z.B. Suchbaum)
  - Ableitungsbäume, Syntaxbäume, Codebäume
  - Entscheidungsbäume

Definitionen

- *Wurzel*: kein Vorgänger.
- *Innerer Knoten*: mindestens ein Nachfolger.
- *Blatt* oder *Endknoten*: kein Nachfolger.
- Anzahl der Kinder eines Knotens p: Rang von p.
- Höhe eines Knotens: längster Pfad vom Knoten zu erreichbaren Blatt (Anzahl der Kanten).
  - Höhe des Baumes t, der nur aus der Wurzel besteht ist 0.
  - Höhe eines beliebigen Baumes mit d Teilbäumen: $h(t) = max(h(t_1), h(t_2), ..., h(t_d)) + 1$
- Tiefe eines Knoten: Pfad vom Knoten zur Wurzel.
- Ordnung eines Baumes d: max. Anzahl von Kindern eines Knoten (von allen Knoten dieses Baumes).

Eigenschaften und Gliederungen

- Eigenschaften
  - Ein Baum mit n Knoten besitzt genau $n - 1$ Kanten.
  - Höhe des Baumes h = Höhe der Wurzel = Tiefe des Baumes = Tiefe des äußersten Blattes
- Arten
  - Ungeordnete Bäume: Nachfolgeknoten unterliegen **keiner** Reihenfolge (z.B. Filesystem)
  - Geordnete Bäume: Reihenfolge ist **relevant** (z.B. Syntaxbaum)
- Unterscheidung bzgl. Anzahl der Nachfolger
  - **Binäre Bäume**: Ordnung $d = 2$ (z.B. binärer Suchbaum, AVL-Baum).
  - **Mehrwegbäume**: Ordnung $d > 2$ (z.B. B-Baum).

Binärer Suchbaum

- Ordnung $d = 2$
- Knoten
  - Dateninhalte
  - Schlüssel: `key`
    - Kriterium vergleichbar (z.B. nummerisch, alphabetisch)
    - kann nur einmal im Baum vorkommen
  - linker und rechter Nachfolger: `left`, `right`
    - linker TB: Schlüssel sind kleiner als aktueller Schlüssel
    - rechter TB: Schlüssel sind größer als aktueller Schlüssel

Bedingungen im BSB

- Max. Anzahl von Blättern: 2^h
- Max. Gesamtanzahl von Knoten: $2^{h+1} - 1$
  - Summe aller Knoten aller Ebenen = $\sum 2^i$ für $i = 0 \ldots h$
    Erklärung: entspricht der größten darstellbaren Zahl bei $h + 1$ Bits d.h. $2^{h+1} - 1 = \sum 2^i$ für $i = 0 \ldots h$
- Hat ein Baum n innere Knoten, dann:
  - hat dieser maximal $n + 1$ Blätter
  - kann h maximal n sein
  - ist h minimal $\log_2(n + 1)$

Traversieren

- Durchlaufen aller Knoten eines Baumes in einer bestimmten Reihenfolge.
- Anwendungen
  - die Ausgabe aller Knotenwerte
  - das Durchführen von Operationen auf allen Knotenwerten
- Arten
  - preorder (Prefix): Wurzel, linke Seite, rechte Seite
  - inorder (Infix): linke Seite, Wurzel, rechte Seite
    Anwendung: sortierte Ausgabe
  - postorder (Postfix): linke Seite, rechte Seite, Wurzel
    Anwendung: Speicherfreigabe beim Löschen

Traversieren mit Rekursion

```python
def inorder(p):
    if p != None:
        inorder(p.left)
        write(p.key)        # Operation auf Knoten
        inorder(p.right)
```

Traversieren ohne Rekursion

- Symmetrischer Nachfolger (symmetrical successor, NF): Knoten mit kleinstem Schlüssel des rechten Teilbaumes.
  - Fädelungszeiger: zeigt auf symm. NF (gefädelter Baum)
  - Nachfolgezeiger als Fädelungszeiger, aber: Markierung!

```python
def symm_succ(p):
    if p.right != None:
        if p.right_is_threaded:
            return p.right
        else:
            q = p.right;
            while q.left != p:
                q = q.left
            return q
    else:
        return None  # p hat keinen symm. NF
```

- Algorithmus
  1. Weitest links stehenden Knoten suchen
  2. wiederholter Aufruf von `symm_succ()`

Suchen mit Rekursion

- geg.: Wurzelknoten (Anker) und zu suchender Knoten (Schlüssel)

```
def search(p, key):  # Suchen in Baum p nach key
    if (p == None):
        return None  # Baum leer: nicht gefunden
    else:
        if key == p.key:
            return p  # gefunden
        else:
            if key < p.key:
                # im linken TB weitersuchen
                return search(p.left, key)
            else:
                # im rechten TB weitersuchen
                return search(p.right, key)
```

- Nachteile
  - Bei jedem Knoten Überprüfung, ob Blatt $\rightsquigarrow$ Stoppknoten
  - Rekursiv: mehr Ressourcen. $\rightsquigarrow$ iterative Suche

Stoppknoten

- Hinzufügen eines zusätzlichen Knotens
- Alle NF von eigentlichen Blättern $\leadsto$ Stoppknoten
- Beim Suchen
  - Key von Stoppknoten setzen
  - Abfrage auf `== None` kann entfallen
  - am Schluss: auf Stoppknoten abfragen
- Beim Einfügen
  - Referenz auf Stoppknoten hinzufügen
- Beim Löschen
  - Referenz auf Stoppknoten umhängen

Suchen ohne Rekursion

```python
def search(p, key):
    while p != None:
        if key == p.key:
            return p
        elif key < p.key:
            p = p.left
        else:
            p = p.right
    return None
```

Einfügen mit Rekursion

```python
def insert(p, key):
    if p == None:  # Baum leer?
        return Node(key)  # neuen Knoten anlegen
    else:
        if key < p.key:
            p.left = insert(p.left, key)  # in den
        elif key > p.key:
            # in den rechten TB
            p.right = insert(p.right, key)
        return p  # bestehender Knoten zurueck

root = None
root = insert(root, 10)
root = insert(root, 5)
root = insert(root, 15)
```

Einfügen ohne Rekursion

```
def insert(p, key):  # p != None
    while True:
        if key == p.key:
            return False  # schon vorhanden
        elif key < p.key:  # im li TB weitersuchen
            if p.left == None:  # linker TB leer!
                p.left = Node(key)  # anlegen
                return True
            else:  # li TB nicht leer
                p = p.left  # weiter
        else:
            if p.right == None:
                p.right = Node(key)
                return True
            else p = p.right
```

Löschen

1. Zu löschenden Knoten suchen
2. Löschknoten = Blatt: löschen
3. Löschknoten = Knoten mit **einem** Teilbaum: kurzschließen
4. sonst: Löschknoten ersetzen durch (2 Möglichkeiten)
   4.1 den Knoten mit dem größten Wert aus dem linken Teilbaum (Knoten, der am weitesten rechts steht)
   4.2 den Knoten mit dem kleinsten Wert aus dem rechten Teilbaum (Knoten, der am weitesten links steht)

Löschen – 2

```python
def remove(p, key):  # mit call-per-reference!!!
    if p == None: pass  # Key nicht im Baum
    else:
        if key < p.key: remove(p.left, key)
        elif key > p.key: remove(p.right, key)
        else:  # p.key == key
            if p.left == None: p = p.right  # kurzschliessen
            elif p.right == None: p = p.left  # kurzschl.
            else:  # p.left != None und p.right != None
                q = parentSymmSucc(p)
                if p == q:  # re Kind von q ist symm. NF
                    p.key = q.right.key
                    q.right = q.right.right
                else:  # li Kind von q ist symm. NF
                    p.key = q.left.key
                    q.left = q.left.right
```

Löschen – 3

- Vater des symm NF

```
def parentSymmSucc(p):
    if p.right.left != None:
        p = p.right
        while p.left.left != None:
            p = p.left
    return p
```

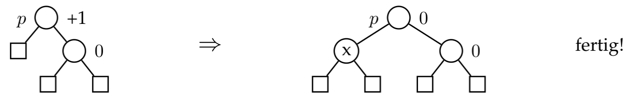
- `remove` funktioniert nicht in Programmiersprachen, die ausschließlich ”per-value” übergeben, daher:
  - `remove(p.left, key)` $\rightsquigarrow$ `p.left = remove(p.left, key)`
  - detto mit rechts
  - `return p` am Ende von `else` hinzufügen
- Speicher von Knoten wird nicht explizit freigegeben

AVL Baum

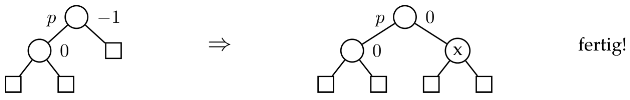
- BSB kann degenerieren
  - beim Einfügen Umordnungen vornehmen $\leadsto$ ausgeglichene Bäume
- Mathematiker Adelson-Velskii und Landis (1962)
- spezieller BSB
  - bei jedem Knoten unterscheidet sich die Tiefe des li TB von der des re TB um maximal 1.
  - Balance eines Knoten p
    - bal(p) = h(p.right) $-$ h(p.left)
  - d.h. drei zulässige Balancen: $-1, 0, +1$
- Vorteil: Geringerer Suchaufwand, da nicht degeneriert (Suchschritte: $O(\log_2(n))$ mit n = max. Anzahl von Knoten)
- Nachteil: Höherer Aufwand bei Modifikationen

Einfügen in AVL

1. Leerer Baum: fertig
2. p ist Vater des Blattes, an dem die Suche endet:
   - $\mathrm{bal}(p) = +1$

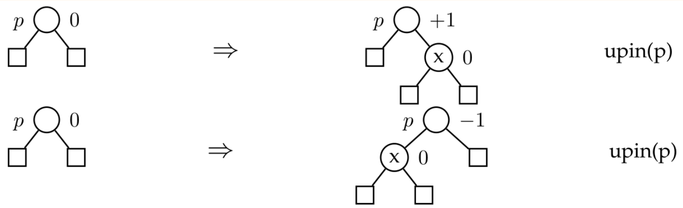


   - $\mathrm{bal}(p) = -1$



Einfügen in AVL – 2

- $\text{bal}(p) = 0$

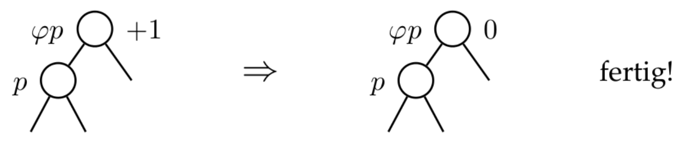


upin(p) wird aufgerufen, wenn: $\text{bal}(p) \in \{+1, -1\}$

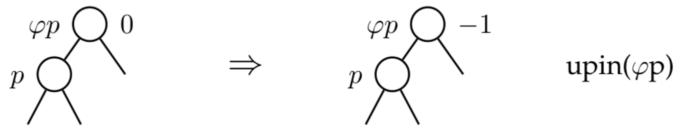
Funktion `upin`

Fall 1 [p ist **linkes** Kind seines Vaters φp]

- Fall 1.1 [bal$(\varphi p) = +1$]



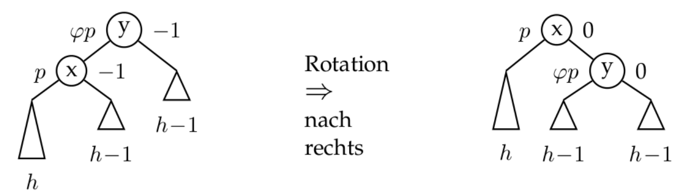
- Fall 1.2 [bal$(\varphi p) = 0$]



Funktion `upin` – 2

- Fall 1.3 [bal(φp) $= -1$]

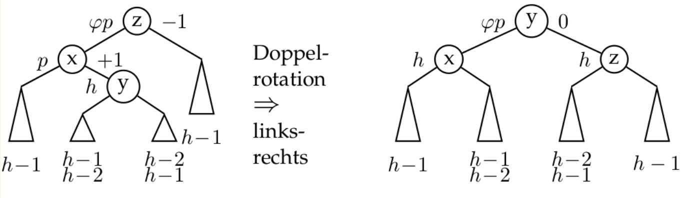
- Fall 1.3.1 [bal(p) $= -1$]



fertig!

Funktion `upin` – 3

- Fall 1.3.2 [bal$(p) = +1$]



fertig!

Mehrwegbäume

- Ordnung $d > 2$
- Implementierungsmöglichkeiten
  - Liste aller Kindknoten
  - Zeiger auf erstes Kind und Zeiger auf nächsten Bruderknoten
  - $\rightsquigarrow$ B-Baum

B-Baum

- ausgeglichener (balancierter), geordneter Mehrwegbaum
- Motivation
  - Speicherbedarf des Baumes > verfügbarer Hauptspeicher
  - Baum soll modifiziert werden können (löschen, einfügen).
  - Baum z.B. auf Festplatte speichern
  - #Plattenzugriffe soll minimiert werden (z.B. für DBMS)
- Prinzip
  - Knoten soll in einer 'Seite' (engl. page) Platz haben
    - füllt diese jedoch im allgemeinen nicht vollständig,
    - also noch Platz weitere Datensätze einzutragen.
  - Knoten
    - abwechselnd Seitenadressen (SA) und Datensätze (DS)
    - beginnend und endend mit SA (Ausnahme: Blätter).

B-Baum – 2

- Kriterium für kontrolliertes Wachstum gesucht
  - wie bei AVL
- B-Baum der Ordnung k hat folgende Eigenschaften:
  - Alle Blätter haben die gleiche Tiefe.
  - Jeder Knoten hat höchstens k Kindknoten.
  - Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens ceil$(k/2)$ Kindknoten.
  - Die Wurzel hat wenigstens 2 Kindknoten (im Trivialfall, dass die ganzen Daten in einen Knoten passen, ist sie ein Blatt).
  - Jeder Knoten mit i Kindknoten hat $i - 1$ DS.

B-Baum – 3

- k ist so zu wählen, dass ein Knoten gerade noch auf einer Seite Platz hat.
- Wenn die DS sehr lange Informationsteile haben, kann man anstatt des DS nur den Schlüssel und eine Adresse speichern. Dadurch läßt sich k größer wählen und der B-Baum hat eine geringere Höhe.
- Die Ordnung eines üblichen B-Baumes liegt etwa bei 100 bis 200.
- Ist $k = 199$, so haben B-Bäume mit bis zu 1999999 Schlüssel höchstens die Höhe 4.

B*-Baum

- Datensätze werden nur in den Blättern gespeichert.
- Zwischenknoten enthalten nur Schlüssel, die zur Steuerung des Suchvorganges dienen.
- Blätter enthalten nur Datensätze und sonst nichts.
- Vorteil: Innere Knoten können mehr Schlüssel enthalten. Der Baum wird breiter, hat aber weniger Ebenen (d.h. geringere Höhe).