Scripting und Algorithmen

Bäume

Harald Lampesberger

FH OÖ, Department für Sichere Informationssysteme

SAL2VO, SS23, Version: 24. Mai 2023

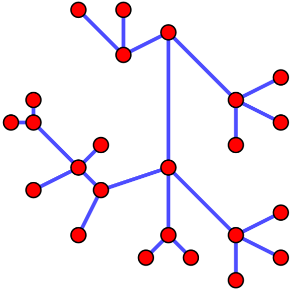
FH OBERÖSTERREICH
UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

Bäume

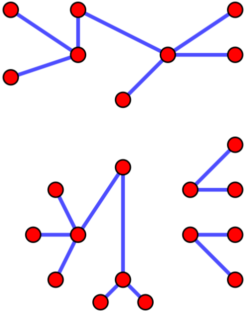
Baum

Ein **Wald** (forest) ist ein azyklischer, ungerichteter Graph

- Die Zusammenhangskomponenten werden jeweils als **Baum** (tree) bezeichnet
- Knoten mit Grad = 1 sind Blätter



Tree



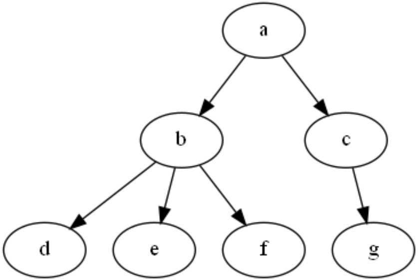
Forest

Bildquelle https://www.mathreference.org/index/page/id/393/lg/en

Gewurzelter Baum

Nicht-Blatt-Knoten wird Wurzel (root node)

- Das Ergebnis ist ein gewurzelter Baum
- In der Praxis wird mit Baumstruktur eigentlich immer ein gewurzelter Baum gemeint
- Wurzel ist *oben*, Kinder *unten* dargestellt
- Jeder Knoten spannt einen Teilbaum auf (auch Blätter)



Tiefe und Höhe

- Tiefe eines Knotens ist die Anzahl der Kanten von Wurzel zum Knoten
- Höhe des Baums ist die maximale Tiefe

Beispiele für Baumstrukturen

Abbildung von hierarchischen Strukturen

- Organisationsstruktur, Phylogenischer Baum, Stammbaum, ...

Betriebssystem

- Praktisch jedes Filesystem
- Prozesshierarchien

Semistrukturierte Datenformate

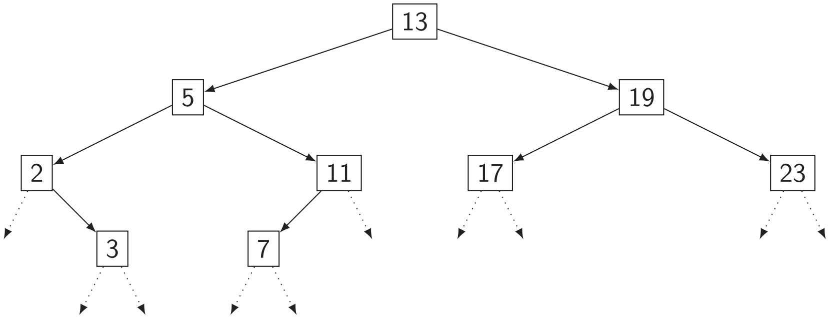
- HTML (Web!), XML, JSON

Spezialfall Binärbaum

Jeder Knoten hat maximal zwei Kindknoten (links, rechts)

Anwendung: Binärsuchbaum

- Jeder Knoten hat einen Wert; Wert des linken Kindes ist immer kleiner und Wert des rechten Kindes ist immer größer als der eigene Wert; mehr dazu später



Bäume in Python

Baumstruktur in Python

Wie ein Graph

- Adjazenzliste, usw.

Objektorientiert

```
class BinaryTree(object):
    def __init__(self, key, value):
        self.key = key          # Wert
        self.value = value      # Datenablage
        self.left = None        # for BinaryTree
        self.right = None       # for BinaryTree
```

Suche in Bäumen

Allgemeine Suche in Bäumen

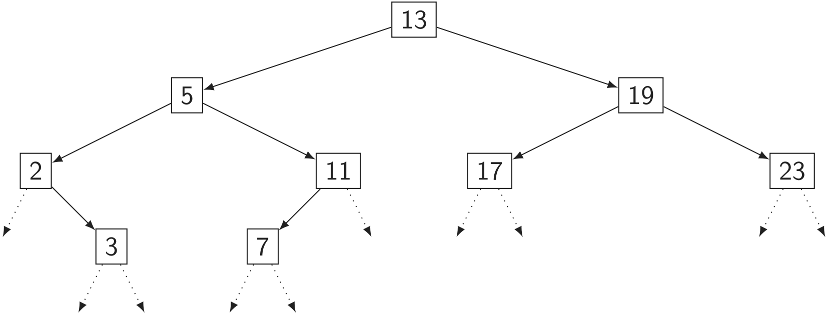
Bei gewurzelten gerichteten Bäumen ist der Start = Wurzel

Wie bei Graphen: Tiefen- und Breitensuche

Variantén der Tiefensuche (zB für Binärsuchbäume)

- Pre-order traversal (= Tiefensuche)
- In-order traversal
- Post-order traversal

Beispiel



Breitensuche („level-weise“) :	13, 5, 19, 2, 11, 17, 23, 3, 7
Pre-order (Graph-Tiefensuche) :	13, 5, 2, 3, 11, 7, 19, 17, 23
In-order :	2, 3, 5, 7, 11, 13, 17, 19, 23
Post-order :	3, 2, 7, 11, 5, 17, 23, 19, 13

Pre-Order Traversal

Zuerst selbst, linker Teilbaum, rechter Teilbaum

- Werte im Binärsuchbaum werden dadurch in aufsteigender Reihenfolge ausgegeben

```
class BinaryTree(object):
    # ...
    def preorder(self):
        print(self.key)
        if self.left:
            self.left.preorder()
        if self.right:
            self.right.preorder()
```

In-Order Traversal

Zuerst linker Teilbaum, dann selbst, rechter Teilbaum

```
class BinaryTree(object):
    # ...
    def inorder(self):
        if self.left:
            self.left.inorder()
        print(self.key)
        if self.right:
            self.right.inorder()
```

Post-Order Traversal

Zuerst linker Teilbaum, rechter Teilbaum, erst danach selbst

```
class BinaryTree(object):
    # ...
    def postorder(self):
        if self.left:
            self.left.postorder()
        if self.right:
            self.right.postorder()
        print(self.key)
```

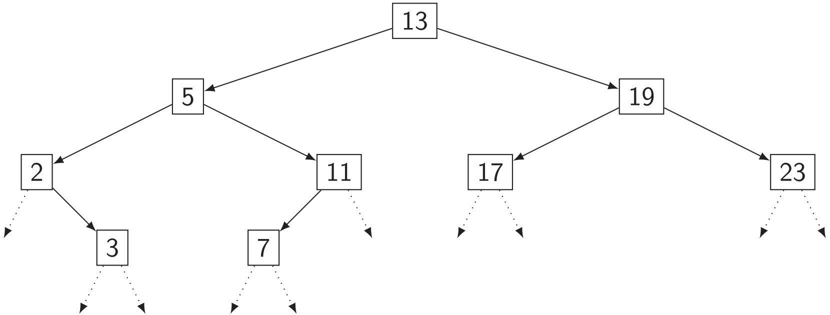
Binärsuchbaum

Binärsuchbaum

Dynamische Datenstruktur für schnelle Suche

Jeder Knoten hat einen Wert (Key) und folgende Eigenschaften

- Werte im linken Teilbaum sind immer kleiner als der eigene Wert
- Werte im rechten Teilbaum sind immer größer als der eigene Wert



Element finden I/II

Durch die Eigenschaften wird die Suche nach einem Element eine DFS-Variante

```
class BinarySearchTree(BinaryTree):
    def search(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            if self.left:
                return self.left.search(key)
        else:
            # key > slf.key
            if self.right:
                return self.right.search(key)
        return None
```

Element finden II/II

Unser Beispiel-Binärsuchbaum ist **balanciert**

- Rekursive Definition
- Die Höhen der zwei Teilbäume links und rechts unterscheiden sich maximal um 1
- UND die beiden Teilbäume links und rechts sind auch balanciert

Höhe eines balancierten Binärsuchbaums mit n Knoten ist $\log n$

Speicherkomplexität ist in allen Fällen $\mathcal{O}(n)$

Zeitkomplexität

- Worst case (nicht balanciert): $\mathcal{O}(n)$
- Avg case (balanciert): $\mathcal{O}(\log n)$

Element einfügen I/II

Erstes Element wird zur Wurzel; Folgeelemente werden in den linken oder rechten Teilbaum eingeordnet

```
class BinarySearchTree(BinaryTree):
  # ...
    def insert(self, key, value):
        if key == self.key:
            self.value == value      # key exists, update value
        elif key < self.key:
            if self.left: self.left.insert(key, value)
            else:         self.left = BinarySearchTree(key, value)
        else:
            # key > self.key
            if self.right: self.right.insert(key, value)
            else:          self.right = BinarySearchTree(key, value)
```

Element einfügen II/II

Zeitkomplexität hängt direkt davon ab, ob der Baum balanciert ist

- Worst case (nicht balanciert): $\mathcal{O}(n)$
- Avg case (balanciert): $\mathcal{O}(\log n)$

Ein balancierter Baum kann nach dem Einfügen die Balance verlieren

- Aus diesem Grund gibt es selbst-balancierende Binärsuchbäume

Element löschen I/III

Es gibt drei verschiedene Fälle beim Löschen

1. Der zu löschende Knoten ist ein Blatt (keine Kinder)
   - Kann einfach gelöscht werden (Referenz im Elternknoten entfernen)

2. Der zu löschende Knoten hat genau ein Kind
   - Kind ersetzt den zu löschenden Knoten
   - zB Werte vom Kind übernehmen und danach Kindknoten löschen

3. Der zu löschende Knoten hat zwei Kinder (schwieriger Fall)
   - Finde den inorder-Nachfolger
   - Ersetze den zu löschenden Knoten durch den inorder-Nachfolger
   - Lösche den inorder-Nachfolger

Element löschen II/III

```python
class BinarySearchTree(BinaryTree):
  # ...
    def delete(self, key):
        if key < self.key:
            self.left = self.left.delete(key)
        elif key > self.key:
            self.right = self.right.delete(key)
        else:                                   # delete slf by return new subtree
            if not self.left:                   # zero or one child
                return self.right
            elif not self.right:
                return self.left
            else:                               # two children
                successor = self.successor()
                self.key = successor.key
                self.value = successor.value
                self.right = self.right.delete(self.key)   # del former successor
        return self
```

Element löschen III/III

```
class BinarySearchTree(BinaryTree):
  # ...
    def min(self):                    # descent left until no further
        if self.left: return self.left.min()
        else:         return self

    def successor(self):              # successor is min() of the right subtree
        if self.right: return self.right.min()
        else:          return None
```

Zeitkomplexität des Löschens

- Worst case (nicht balanciert): $\mathcal{O}(n)$
- Avg case (balanciert): $\mathcal{O}(\log n)$

Durch Löschen kann der Binärsuchbaum die Balance verlieren

Zusammenfassung

Baumstrukturen sind in der Informatik überall zu finden

- Netzwerktechnik / verteilte Systeme / Semistrukturierte Datenformate

Was ist der Vorteil eines Binärsuchbaums gegenüber einer Hash-Datenstruktur?

- Im best case sind beide schnell, im worst case sind beide gleich schlecht
- Binärsuchbaum kann effizient inorder-traversiert werden

Suche in einem Baum geht schnell, wenn er balanciert ist

- Deswegen gibt es selbstbalancierende Binärbäume, zB AVL-Baum, Rot-Schwarz-Baum
- Oder andere Baumkonzepte, zB B-Tree, R-Tree, Quadtree

Grundlage für schnelles Suchen

- Rot-Schwarz-Binärbaum für den CFS Scheduler und mmap/munmap im Linux Kernel
- B-Tree für Indizes in Datenbanken und Dateisystemen