2.5 Bäume

Bäume

- Nochmal: Suchen in Mengen ...
- Sortieren und Suchen
  - Nur sinnvoll für statische Mengen
  - Insert(), Delete() sind O(n)
  - Search() ist O(log n)
  - Speichermanagement

Bäume

- Nochmal: Suchen in Mengen …
- Hashing
  - Zahl der Objekte muss vorher bekannt sein → Tabellengröße
  - Insert(), Search() sind O(1)
  - Delete() nicht praktikabel bei geschlossenem Hashing

Bäume

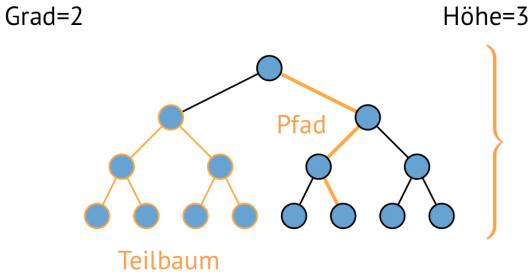
- Nochmal: Suchen in Mengen …
- Binärbäume
  - Beliebig dynamisch erweiterbar
  - Insert(), Delete(), Search() sind O(log n)
- Knoten enthalten mindestens
  - Zeiger P zum Vorgänger
  - Zeiger L, R zum linken, rechten Nachfolger
  - Suchschlüssel X

Optimalerweise: Balancierter Suchbaum (garantierte Komplexität von O(log n))

Bäume

- Effiziente und flexible Suchstruktur
  - Speichere Daten in jedem Knoten
  - Speichere Daten nur in den Blättern (z.B. Suchstruktur im Hauptspeicher, Datenblöcke auf der Festplattte)
  - Begriffe: Ordnung, Höhe, Pfad, …

Bäume



Wiederholung der grundlegenden Definitionen

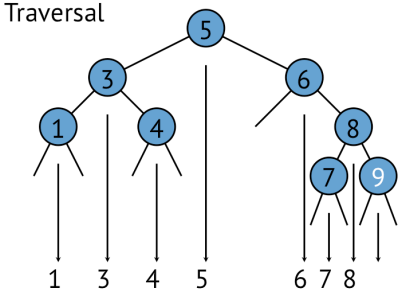
Bäume

- Suchbäume
  - Sei X_i ein Knoten und $X_1 \ldots X_{i-1}$ die Knoten im linken und $X_{i+1} \ldots X_n$ die Knoten im rechten Teilbaum, dann gilt $\max \{X_1 \ldots X_{i-1}\} < X_i < \min \{X_{i+1} \ldots X_n\}$
- Insert(), Delete(), Search(), ...
  - Aufwand proportional zur Pfadlänge

Annahme: Die Schlüssel sind eindeutig. (Daher < statt <=)

Bäume

Inorder Traversal



Führt in Suchbäuen zu sortierter Aufzählung der Elemente

Operationen

- Search(X,R)
  Suche Schlüssel X in Teilbaum R
- Min(N), Max(N)
  Finde minimales/maximales Element in Teilbaum N
- Successor(N), Predecessor(N)
  Finde Vorgänger/Nachfolger zum Knoten N
- Insert(N,R) ...
- Delete(N,R) ...

Search()

```
• Search(X,R)
    if R = NIL or X = R.X then
      return R
    else
      if X < R.X then
        return Search(X,R.L)
      else
        return Search(X,R.R)
```

Min(), Max()

- Min(N)

```
while N.L ≠ NIL do
    N = N.L
return N
```

- Max(N)

```
while N.R ≠ NIL do
    N = N.R
return N
```

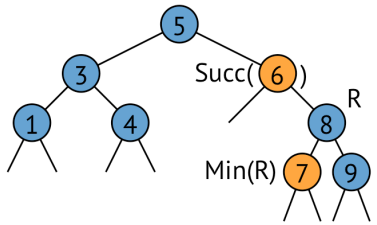

Successor()

- Successor(N)

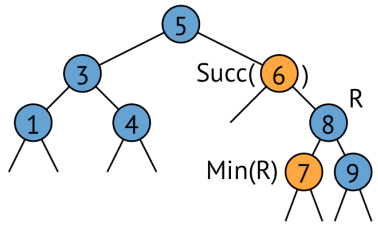
```
if N.R ≠ NIL then
    return Min( N.R )
else
    Q ← N.P
    while Q ≠ NIL and N = Q.R do
        N ← Q
        Q ← Q.P
    return Q
```


Predecessor analog dazu (mit vertauschten Richtungen)

Successor()

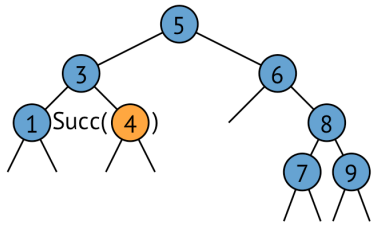


Successor()

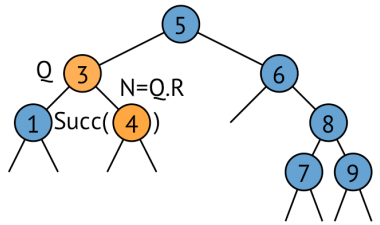


Succ(6) = 7

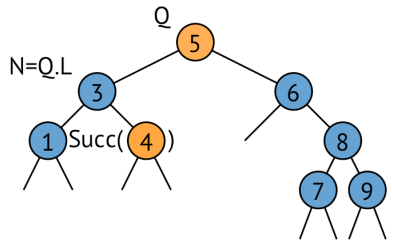
Successor()



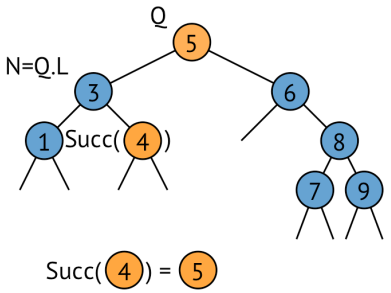
Successor()



Successor()



Successor()

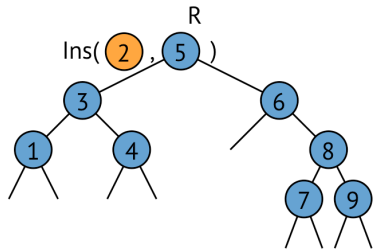


Insert()

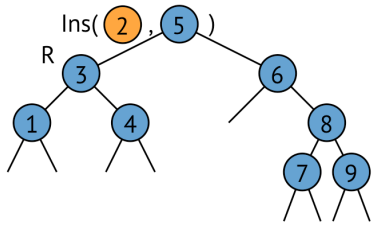
- Insert(N,R)

```
Q ← NIL
while R ≠ NIL do
    Q ← R
    if N.X < R.X then   R ← R.L
    else                R ← R.R
N.P ← Q; N.L ← NIL; N.R ← NIL;
if Q = NIL then
    root ← N
else
    if N.X < Q.X then   Q.L ← N
    else                Q.R ← N
```

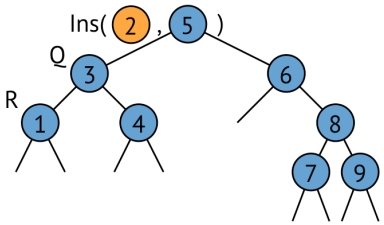

Insert()



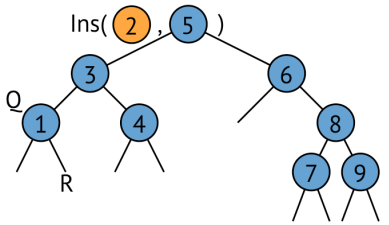
Insert()



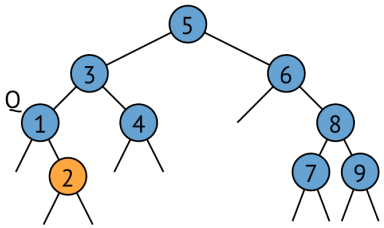
Insert()



Insert()



Insert()

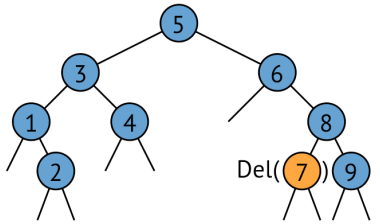


Struktur des Baums hängt von Reihenfolge des Einfügens ab!

Delete()

- Delete(N,R)
- Fallunterscheidung
  1. N hat keine Kinder (Blatt)
  2. N hat ein Kind
  3. N hat zwei Kinder

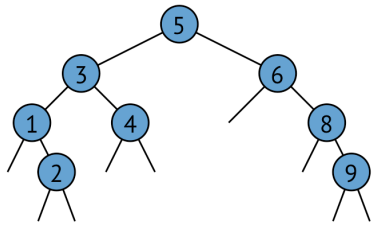
Delete()



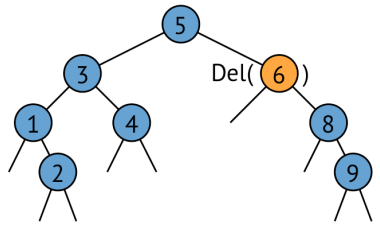
Fall 1: 7 hat keinen Sohn

Fall 1. und 2. sind unproblematisch

Delete()

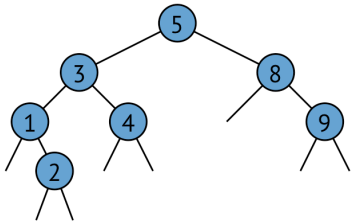


Delete()

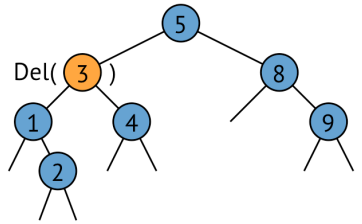


Fall 2: 6 hat genau einen Sohn

Delete()

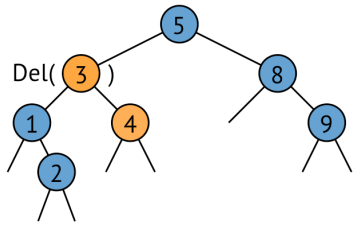


Delete()



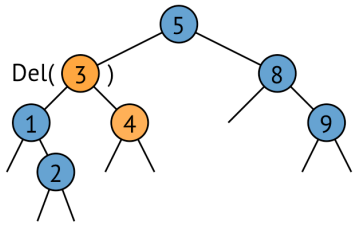
Fall 3: 3 hat zwei Söhne

Delete()



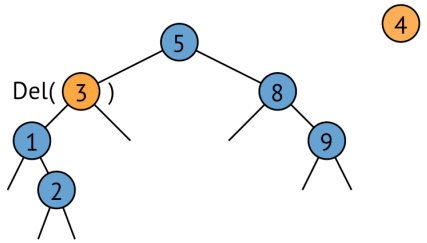
Depth(Successor(3)) > Depth(3)

Delete()



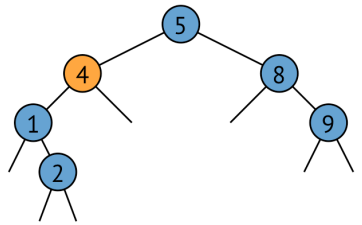
Successor(3) hat maximal ein Kind

Delete()



Kopiere Daten von 4 nach 3

Delete()



Kopiere Daten von 4 nach 3

Delete()

- Delete(N,R)

```
if N.L = NIL or N.R = NIL then   Q ← N
                         else    Q ← Succ(N)
if Q.L ≠ NIL then   S ← Q.L
              else  S ← Q.R
if S ≠ NIL then S.P ← Q.P
if Q.P = NIL then
   root ← S
else
   if Q = Q.P.L then    Q.P.L ← S
                else    Q.P.R ← S
if Q ≠ N then N.X ← Q.X // copy data
return Q
```


Aufwand

- Alle Operationen
  - Search(X,R)
  - Min(N), Max(N)
  - Successor(N), Predecessor(N)
  - Insert(N,R)
  - Delete(N,R)

  bearbeiten genau einen Pfad im Baum
- Aufwand = O(erwartete Pfadlänge)

Aufwand

- Erwartete Höhe eines Suchbaums mit n Knoten $N_1 \ldots N_n$
  - Alle Schlüssel verschieden
  - $N_i.X < N_{i+1}.X$
- Hängt von der Einfügereihenfolge ab
- Mittelwert über alle Permutationen

Aufwand

- Höhe eines binären Suchbaums mit n Knoten: H_n
- Exponentielle Höhe: $X_n = 2^{H_n}$
- Sei N_i der Wurzelknoten, dann bilden $N_1 \ldots N_{i-1}$ und $N_{i+1} \ldots N_n$ die Teilbäume
- Also

$$H_n = 1 + \max\{ H_{i-1}, H_{n-i} \}$$
$$X_n = 2 \times \max\{ X_{i-1}, X_{n-i} \}$$

Aufwand

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$\begin{aligned} E(X_n) &= \frac{2}{n}\sum_{i=1}^{n} E(\max\{X_{i-1}, X_{n-i}\}) \\ &\leq \frac{2}{n}\sum_{i=1}^{n}(E(X_{i-1}) + E(X_{n-i})) \\ &= \frac{4}{n}\sum_{i=0}^{n-1} E(X_i) \end{aligned}$$

Induktion, s. Cormen :

$$= \frac{1}{4}\binom{n+3}{3} \quad = O(n^3)$$

Aufwand

- Erwartungswert ... Mittelwert über alle möglichen Indizes i des Wurzelknotens

$$2^{E(H_n)} \leq E(2^{H_n}) = E(X_n) \leq cn^3$$

$$E(H_n) = \log(cn^3) = 3\log n + \log c = O(\log n)$$

Aufwand

- $E(H_n) = O(\log n)$ bedeutet, dass die Pfadlängen in einem Suchbaum mit n Knoten im erwarteten Fall $O(\log n)$ sind.
- Alle Operationen Insert(), Delete(), … haben einen erwarteten Aufwand von $O(\log n)$
- Aber Worst-Case Aufwand $O(n)$!!!