Algorithmen und Datenstrukturen

Kapitel 5: Datenstruktur Baum

Prof. Ingrid Scholl

FH Aachen - FB 5
scholl@fh-aachen.de

23.04.2018

Kapitel 5 - Überblick

Einleitung

- Datenstrukturen waren bisher linear bzw. 1-dimensional. Eine Vorgänger-Nachfolger-Relation.
- Die Datenstruktur Baum ist n-dimensional: Jedes Element bzw. Knoten kann bis zu n Nachfolger haben (1:n).
- Besonderer Baum: Binärbaum ist 2-dimensional. Jeder Knoten hat bis zu 2 Nachfolger.

Definition (Baum)

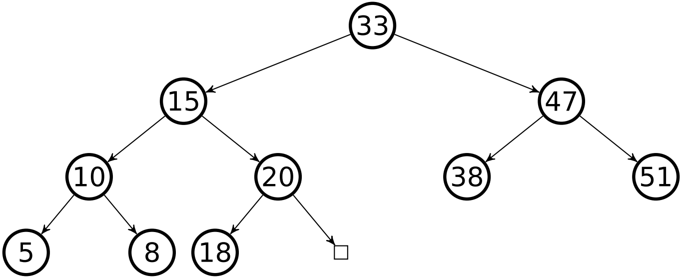
Ein Baum ist eine Datenstruktur, die aus einer Menge von Knoten besteht. Jeder Knoten kann auf andere Knoten über Kanten (Referenzen) verweisen.

Wichtigste Anwendung von geordneten Bäumen: SUCHEN

Bäume - Begriffe

- Der erste Knoten im Baum wird als Wurzel-Knoten bezeichnet.
- Bei einem Binärbaum kann jeder Knoten bis zu zwei Nachfolger-Knoten haben, einen linken und rechten Nachfolgerknoten.
- Hat ein Knoten keinen Nachfolger, verweist dieser auf die nullptr (Null-Referenz). Dieser Knoten wird als Blatt-Knoten bezeichnet.
- Alle Knoten, die keine Blatt-Knoten sind, werden als Innere Knoten bezeichnet.

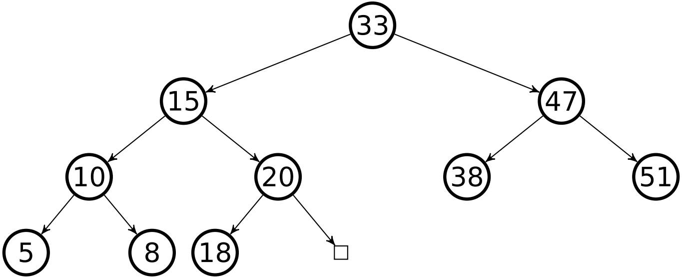
Baumeigenschaft: Niveau



Definition (Niveau)

Das **Niveau eines Knotens** ist die Länge des Pfades von der Wurzel bis zu diesem Knoten.

Baumeigenschaft: Höhe der Knoten



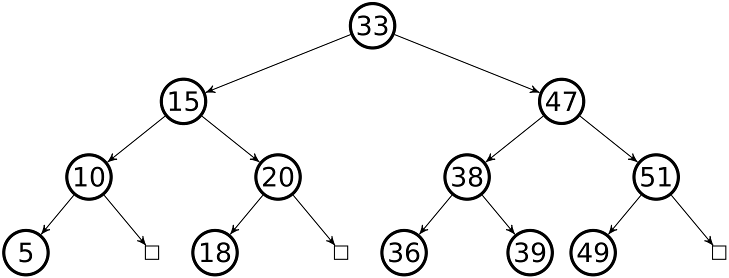
Definition (Höhe)

Die **Höhe eines Knotens** ist die Anzahl der Kanten bis zum tiefsten Blatt und berechnet sich rekursiv durch:

$$height(node) = \max\{height(node \rightarrow left), height(node \rightarrow right)\} + 1$$

Die Höhe eines Blattknotens = 0 und einer NullPtr-Referenz = -1.

Beispiel: Höhe und Niveau



Node	Height	Niv.	Node	Height	Niv.	Node	Height	Niv.
5			20			39		
10			33			47		
15			36			49		
18			38			51		

Weitere Baumeigenschaften

Definition (Pfad)

Ein Pfad in einem Baum ist eine Folge von verschiedenen Knoten, in der die aufeinander folgenden Knoten durch Kanten miteinander verbunden sind.

Definition (Grundlegende Baum-Eigenschaften:)

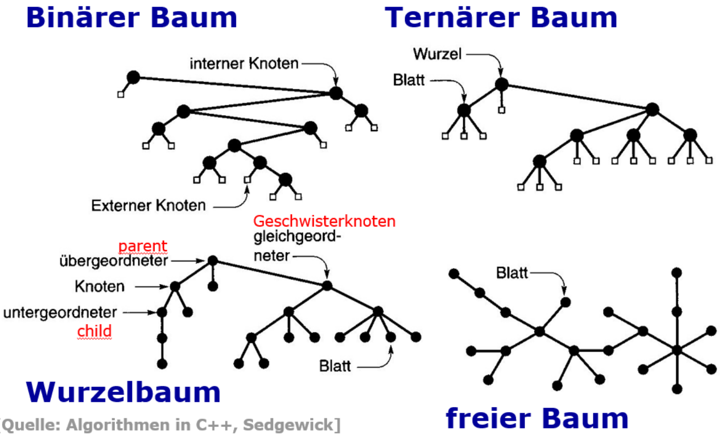
Zwischen jedem Knoten und der Wurzel gibt es genau einen Pfad. Dies bedeutet, dass:

- der Baum **zusammenhängend** ist und
- es **keine Zyklen** gibt.

Baumtypen unterscheiden sich durch:

- die maximale Anzahl *n* der Kindknoten (n-ärer Baum),
- durch die Anordnung der Kindknoten (geordneter Baum).

Beispiel: Baumtypen



[Quelle: Algorithmen in C++, Sedgewick]

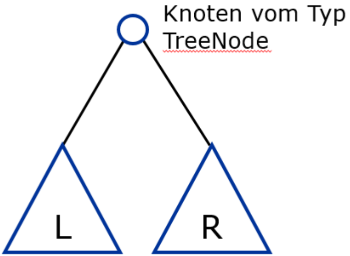
Binärbaum: Rekursiver Aufbau

Linker/Rechter Teilbaum

Ein Binärbaum besteht aus einem Wurzel-Knoten (DS TreeNode). Dieser ist entweder:

- eine Null-Referenz (bei einem leeren Binärbaum) oder
- verbunden mit Knoten, die über die linke Nachfolge-Referenz erreichbar sind (Linker Teilbaum) oder
- verbunden mit Knoten, die über die rechte Nachfolge-Referenz erreichbar sind (Rechter Teilbaum).

Dies gilt für jeden Knoten.



L: linker Teilbaum
R: rechter Teilbaum

Definition: Binärer Suchbaum

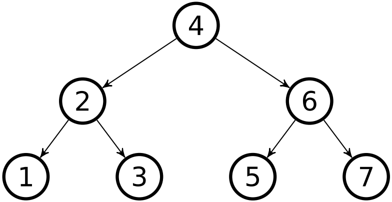
Definition (Binärer Suchbaum (*binary search tree - BST*))

Ein *binärer Suchbaum* ist ein Binärbaum und zeichnet sich zusätzlich dadurch aus, dass jeder Knoten einen Vergleichs-Schlüssel *key* hat. Sei x ein Knoten im BST.

- Wenn y ein Knoten im linken Teilbaum von x ist, dann gilt $key[y] < key[x]$.
- Ist y ein Knoten im rechten Teilbaum von x, dann gilt $key[y] \geq key[x]$.

Eigenschaften BST:

- Jeder Knoten hat bis zu 2 Nachfolger.
- Geordneter Baum durch Vergleichsschlüssel.
- Zyklenfrei.
- Zusammenhängend.



Beispiel: Aufbau Binärer Suchbaum

Erzeuge einen binären Suchbaum durch sukzessives Einfügen der Zahlen
5, 9, 10, 2, 7, 4, 1, 6, 3, 8

Implementierung BST

Ein BST kann mittels einem Array oder i.d.R. mit einer dynamischen verketteten Datenstruktur implementiert werden.

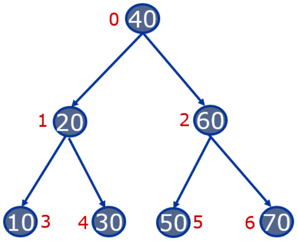
Beispiel: Array-Implementierung

```
int* tree = new int[10];
```

Indizes vom:

- Wurzel-Knoten:
- Vorgänger:
- Linken Nachfolger:
- Rechten Nachfolger:

40, 60, 20, 30, 10, 50, 70



i	0	1	2	3	4	5	6
tree[i]	40	20	60	10	30	50	70

Beispiel: Array-BST

Erzeuge sukzessive einen BST in ein Array durch Einfügen der folgenden Zahlen:

8, 4, 10, 6, 5, 10, 15, 30

Welche Indizes sind durch welche Knoten belegt?

Bsp: Interface Array-BST

```
class Entry {
public:
   int key;
   Value val;
};
```

```
class BinaryTree {
private:
   _______BST; // BST Array
public:
   BinaryTree(int n) {
      ____________________
      ____________________
      ____________________
   }
   ~BinaryTree;
   void put(int key, Value val);
   Value get(int key);
};
```

Dynamische Implementierung BST

Bei der dynam. Implementierung des BST besteht dieser aus referenzierten Knoten.

Datenstruktur für den Knoten:

```
class TreeNode{
public:
   int key;
   Value val;
   TreeNode *left;
   TreeNode *right;
};
```

DS geeignet für Vorwärts-Bewegungen durch den Baum, z.Bsp. entlang eines Pfades von der Wurzel bis zum Blatt - Top-Down

Dynamische Implementierung BST

Datenstruktur für den Knoten:

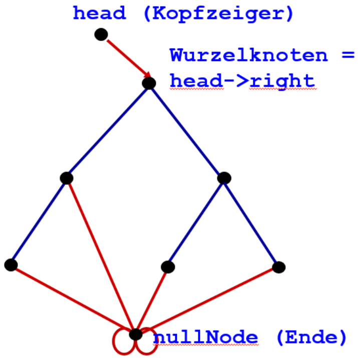
```
class TreeNode{
public:
   int key;
   Value val;
   TreeNode *left;
   TreeNode *right;
   // Zusätzlich:
   TreeNode *parent;
};
```

DS geeignet für Rückwärts-Bewegungen, z.Bsp. entlang eines Pfades vom Blatt bis zur Wurzel - Bottom-Up

Dynamische Implementierung BST

Dynam. BST-Datenstruktur:

```
class BinaryTree{
private:
   TreeNode *head;
   TreeNode *nullNode;
public:
   BinaryTree();
   ~BinaryTree;
   void put(int key, Value val);
   Value get(int key);
   bool remove(int key);
};
```



Dynamische Implementierung BST

Dynam. BST-Datenstruktur:

```
class BinaryTree{
private:
   TreeNode *head;
   TreeNode *nullNode;
public:
   BinaryTree();
   ~BinaryTree;
   void put(int key, Value val);
   Value get(int key);
   bool remove(int key);
};
```

- nutzt DS für Knoten
- Einstieg in den BST mit Ankerknoten head.
  root = head->right;
- optional: nullNode
- Client-Methoden: Konstruktor, Destruktor, Einfügen, Löschen, Suchen.
- Weitere Methoden: Element updaten, Geschwisterknoten, Elternknoten, Algorithmen zur Traversierung durch alle Knoten des Baumes.

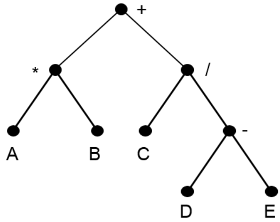
Beispiel: Aufbau eines BST aus der Postfix-Notati

Postfix: **A B * C D E - / +**

```
char c; Stack stack(50);
TreeNode *nullnode = new TreeNode();
TreeNode *x = new TreeNode();
nullnode->left = nullnode;
nullnode->right = nullnode;
while(cin.get(c))
{   while(c == '␣') cin.get(c);
    x = new TreeNode();
    x->info = c;
    x->left = nullnode;
    x->right = nullnode;
    if (c == '+' || c == '*'||...)
    {   x->right = stack.pop();
        x->left = stack.pop();
    }
    stack.push(x);
}
```

```
class TreeNode {
public:
  char info;
  TreeNode *left;
  TreeNode *right;
}
```

Syntax-Baum:



Algorithmen zur Traversierung eines Binärbaumes

Ziel: Besuche alle Knoten im Binärbaum durch einen Traversierungs-Algorithmus.

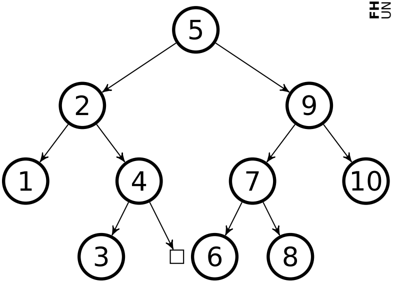
1. **Inorder (LWR):**
   traversiert zuerst rekursiv den linken Teilbaum (L), dann den Knoten (W) selbst und anschließend den rechten Teilbaum (R)
2. **Preorder (WLR):**
   traversiert zuerst den Knoten selbst, dann Traversieren des linken und rechten Teilbaums
3. **Postorder (LRW):**
   traversiert zuerst beide Teilbäume, dann den Knoten selbst
4. **Levelorder:**
   traversiert niveau- bzw. levelweise die Knoten

Inorder (LWR)

Ausgabe des Binärbaumes in Inorder-Reihenfolge:

Algorithm 1: Inorder

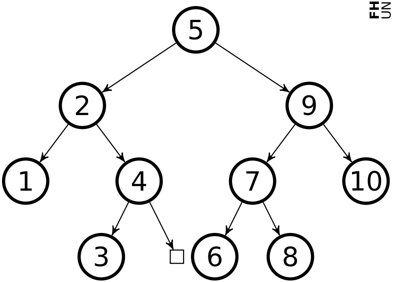
Function *Inorder(k)*
  if $k \neq Null$ **then**
    $Inorder(k \rightarrow left)$
    $Verarbeite(k \rightarrow key)$
    $Inorder(k \rightarrow right)$
end



Sortierte Verarbeitung beim BST

Preorder (WLR)

Ausgabe des Binärbaumes in Preorder-Reihenfolge:



Algorithm 2: Preorder

Function *Preorder(k)*
    if $k \neq Null$ **then**
        $Verarbeite(k \rightarrow key)$
        $Preorder(k \rightarrow left)$
        $Preorder(k \rightarrow right)$
end

TopDown-Verarbeitung

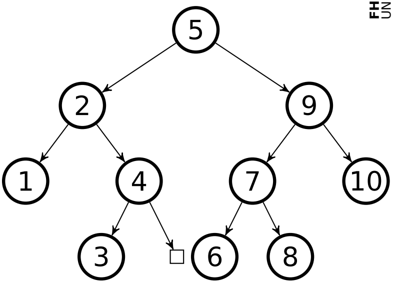
Postorder (LRW)

Ausgabe des Binärbaumes in Postorder-Reihenfolge:

Algorithm 3: Postorder

Function *Postorder(k)*
    if $k \neq Null$ **then**
        $Postorder(k \rightarrow left)$
        $Postorder(k \rightarrow right)$
        $Verarbeite(k \rightarrow key)$
end

BottomUp-Verarbeitung

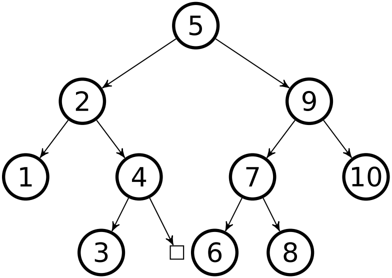


Levelorder

Hier: nicht rekursiver Algorithmus, verwendet Warteschlange. Levelorder-Reihenfolge des BST:

Algorithm 4: Levelorder

Function *Levelorder(k)*
    Queue q
    if $k \neq Null$ **then**
        $q.push(k)$
    while $(!q.IsEmpty())$ **do**
        $node \leftarrow q.pop()$
        $Verarbeite(node \rightarrow item)$
        $q.push(node \rightarrow left)$
        $q.push(node \rightarrow right)$
    end
end



Niveauweise Verarbeitung

Datenstruktur: Binärbaum

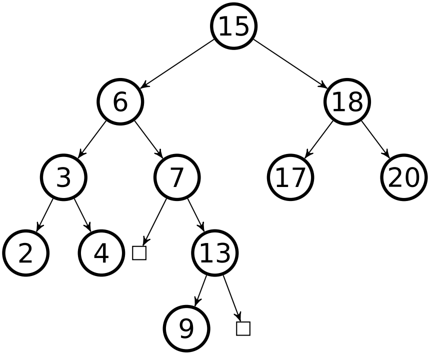
```
class BinaryTree {
   private:
      TreeNode *head;        // Ankerknoten des Binärbaums
      TreeNode *nullNode;    // Pseudoknoten für die Blätter
      void printInorder(TreeNode *k);      // Ausgabe in Inorder
      void printPreorder(TreeNode *k);     // Ausgabe in Preorder
      void printPostorder(TreeNode *k);    // Ausgabe in Postorder
      void printLevelorder(TreeNode *k);   // Ausgabe in Levelorder
   public:
      BinaryTree();     // Konstruktor
      ~BinaryTree();    // Destruktor: Löscht alle Knoten

      void put(int key, Value val); // neues Element einfügen
      Value get(int key)            // Knoten suchen
      bool remove(int key);         // Knoten entfernen

      // Traversierungsmethoden
      void printInorder();      // Ausgabe in Inorder
      void printPreorder();     // Ausgabe in Preorder
      void printPostorder();    // Ausgabe in Postorder
      void printLevelorder();   // Ausgabe in Levelorder
};
```

Kapitel 5 - Überblick

Suchen im BST

Die Suche nach einem Schlüssel ist die größte Anwendung von BST.

- Wie funktioniert die Suche?
- Wie groß ist der Aufwand für die Suche?



Suchen im BST

Algorithm 5: Rekursive Suche im BST

Function *TreeSearch(x, k)*

  if $x = Null$ *or* $k = x \rightarrow key$ **then**

    return x

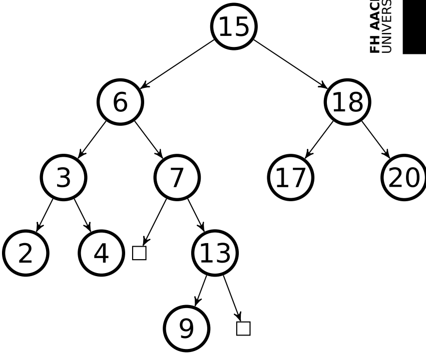
  if $k < x \rightarrow key$ **then**

    return $TreeSearch(x \rightarrow left, k)$

  else

    return $TreeSearch(x \rightarrow right, k)$

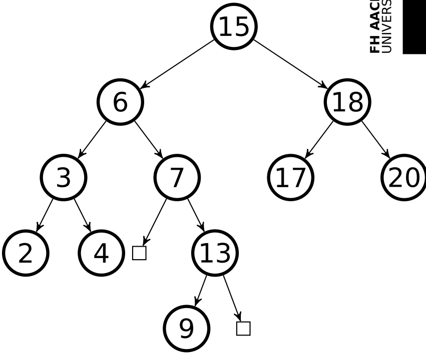
end



Suchen im BST

Algorithm 6: Iterative Suche im BST

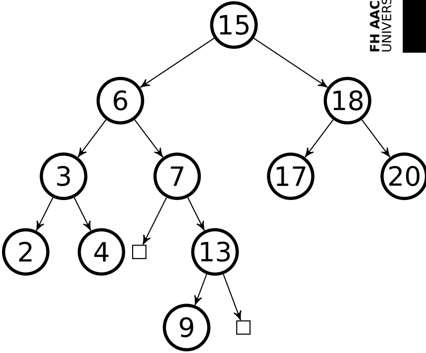
Function $IterativeTreeSearch(x, k)$
  while $(x \neq Null)$ and $(k \neq x \rightarrow key)$ **do**
    if $k < x \rightarrow key$ **then**
      $x = x \rightarrow left$
    else
      $x = x \rightarrow right$
  end
  return x
end



Suchen nach Minimum im BST

Algorithm 7: Minimum Suche im BST

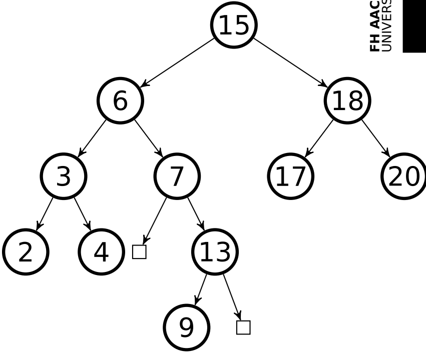
Function *MinimumTreeSearch(x)*
  while $(x \rightarrow left \neq Null)$ **do**
    $x = x \rightarrow left$
  end
  return x
end



Suchen nach Maximum im BST

Algorithm 8: Maximum Suche im BST

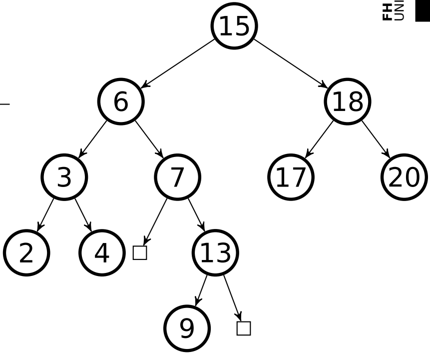
Function *MaximumTreeSearch(x)*
  while ($x \rightarrow right \neq Null$) **do**
    $x = x \rightarrow right$
  end
  return x
end



Suchen im BST

Mittlere Anzahl der Vergleiche für die Suche:

Knoten	Niveau	Anzahl Vergleiche
15	0	1
6, 18	1	2
3, 7, 17, 20	2	3
2, 4, 13	3	4
9	4	5



Einschub: Vollständiger Binärbaum

Definition (Vollständiger Binärbaum)

Beim vollständigen Binärbaum sind alle Knoten pro Niveau gefüllt außer im letzten Niveau. I.d.R. gilt auch hier, dass die Knoten im letzten Niveau von links nach rechts gefüllt sind.

Suchen im BST - Analyse

Fazit: Aufwand zur Suche

- **Best Case:** Ausgeglichener bzw. vollständiger Binärbaum. Bei $N = 2^h$ Knoten hat der Baum die Höhe h. Dann werden maximal $(h+1)$ Vergleiche benötigt. Dies entspricht: logarithmischen Aufwand: $h = log_2 N \sim O(log_2 N)$.
- **Worst Case:** Entarteter Baum. Dann ist der Binärbaum eine verkettete Liste. Das tiefste Blatt ist dann bei N Knoten mit $(N+1)$ Vergleichen gefunden. Das entspricht: linearer Aufwand $\sim O(N)$.
- **Allgemein:** Der Aufwand der Suche hängt von der Höhe h des Baumes ab. Im Worst Case muss der gesamte Pfad von der Wurzel bis zum tiefsten Blatt abgefragt werden. Dazu werden dann $(h+1)$ Vergleiche benötigt. Zusammengefasst: zwischen logarithmischen und linearen Aufwand mit $O(h)$.

Einfügen im BST

Was ist zu tun beim Einfügen eines Elementes im BST?

- Einfügen muß die Ordnung im BST erhalten.
- Neues Element wird immer als Blatt eingefügt.

Unterscheide beim Einfügen 2 Fälle:

- **Fall 1: Baum ist leer**
  Neues Element wird die Wurzel des BST.
- **Fall 2: Baum ist nicht leer**
  Suche durch den BST bis ein Nachfolger nicht belegt ist, d.h. Elternknoten zum neuen Element finden. Füge dann das neue Element als Nachfolger zu diesem Knoten ein.

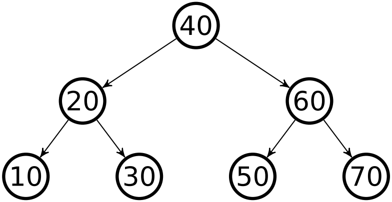
Einfügen im BST

Algorithm 9: Einfügen eines neuen Knoten z im BST

```
Function TreeInsert(T, z)
    y = Null
    x = root[T]
    while (x ≠ Null) do
        y = x
        if key[z] < key[x] then
            x = left[x]
        else
            x = right[x]
    end
    parent[z] = y * Vorgänger von z ist y *
    if y = Null then
        root[T] = z
    else
        if key[z] < key[y] then
            left[y] = z
        else
            right[y] = z
end
```

Einfügen eines neuen Elementes v in einen BST T. Erzeuge vorab neuen Knoten z und initialisiere diesen mit v:

$key[z] = v$
$parent[z] = Null$
$left[z] = Null$
$right[z] = Null$



Einfügen im BST - Analyse

Fazit: Aufwand zum Einfügen

Der Aufwand zum Einfügen eines neuen Knotens entspricht dem Aufwand zur Suche nach dem Elternknoten zum neuen einzufügenden Knoten. Zusätzlich fallen noch einige konstante Operationen für die Erzeugung des Knotens und die zusätzlichen Referenzen an. Dh. analog zur Suche entspricht der Aufwand der Gesamthöhe h des Baumes.

Zusammengefasst:
zwischen logarithmischen und linearen Aufwand mit $\sim O(h)$.

Löschen im BST

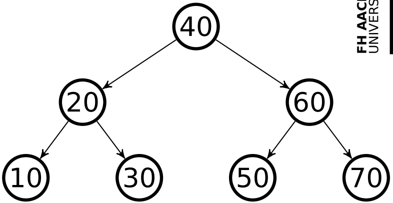
Was ist zu tun beim Löschen eines Elementes z im BST?

- Das zu löschende Element muss im BST gesucht werden.
- Man muss den Vorgänger des zu löschenden Knotens kennen.
- Was dann?

Unterscheide beim Löschen 3 Fälle:

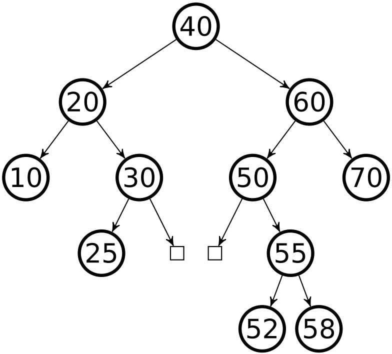
- **Fall 1:** z ist ein Blatt
- **Fall 2:** z hat 1 Nachfolger
- **Fall 3:** z hat 2 Nachfolger

Fall 1: Löschen eines Blattes



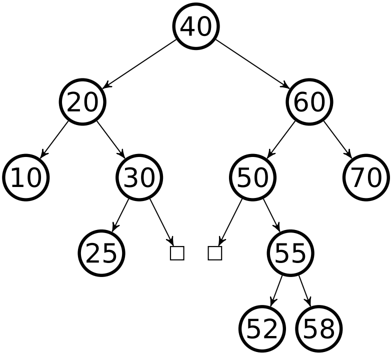
Fall 2: *z* hat 1 Nachfolger

Der zu löschende Knoten *z* hat nur 1 Nachfolger. Bsp: Knoten 30 und Knoten 50.



Fall 3: Knoten *z* mit 2 Nachfolger

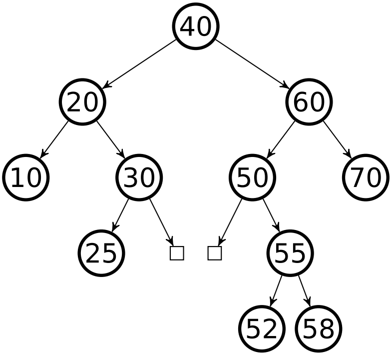
Der zu löschende Knoten *z* hat 2 Nachfolger.
Bsp: Löschen des Knotens 40.



Fall 3: Knoten *z* mit 2 Nachfolger

Der zu löschende Knoten *z* hat 2 Nachfolger.
Bsp: Löschen des Knotens 40.

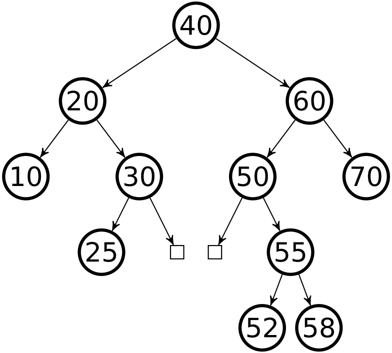
```
TreeNode *tmp = childParent;
tmp->left = child->right;
child->left = node->left;
child->right = node->right;
parent->left = child;
delete node;
```



Fall 3: Knoten *z* mit 2 Nachfolger

Der zu löschende Knoten *z* hat 2 Nachfolger.
Bsp: Löschen des Knotens 40.

- Suche Vorgänger von 40 (parent)
- Suche Minimum im rechten Teilbaum von 40. Das ist der Knoten 50.
- Entferne das Minimum im rechten Teilbaum und ersetze den zu löschenden Knoten durch das Minimum.
- Lösche Knoten 40.

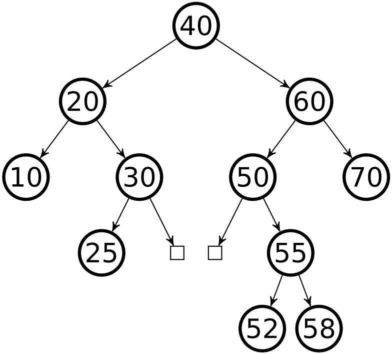


Pseudocode: Löschen im BST

Algorithm 10: Löschen im BST

```
Function TreeDelete(T, z)
    if (left[z] == Null or right[z] == Null)
      then
        y = z
    else
        y = TreeSuccessor(z)
    if (left[y] ≠ Null) then
        x = left[y]
    else
        x = right[y]
    if (x ≠ Null) then
        p[x] = p[y]
    if (p[y] == Null) then
        root[T] = x
    else
        if (y == left[p[y]]) then
            left[p[y]] = x
        else
            right[p[y]] = x
    if (y ≠ z) then
        key[z] = key[y]
        copy y's data into z
end
return y
```

Knoten z soll aus dem BST T gelöscht werden. Sei $p[z]$ der Elternknoten vom Knoten z.



Aufwand: Löschen im BST

Algorithm 11: Löschen im BST

```
Function TreeDelete(T, z)
    if (left[z] == Null or right[z] == Null) then
        y = z
    else
        y = TreeSuccessor(z)
    if (left[y] ≠ Null) then
        x = left[y]
    else
        x = right[y]
    if (x ≠ Null) then
        p[x] = p[y]
    if (p[y] == Null) then
        root[T] = x
    else
        if (y == left[p[y]]) then
            left[p[y]] = x
        else
            right[p[y]] = x
    if (y ≠ z) then
        key[z] = key[y]
        copy y's data into z
end
return y
```

Kapitel 5 - Überblick

Prof. Ingrid Scholl
FH Aachen
Fachbereich für Elektrotechnik und Informationstechnik
Graphische Datenverarbeitung und Grundlagen der Informatik
MASKOR Institut
Eupener Straße 70
52066 Aachen
T +49 (0)241 6009-52177
F +49 (0)241 6009-52190
scholl@fh-aachen.de
www.fh-aachen.de