Teil 1:
Suchen

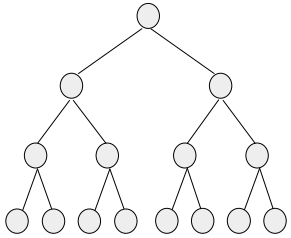
- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
- Binäre Suchbäume (Wiederholung aus Prog 2)
  - Bäume: Begriffe, Eigenschaften und Traversierung
  - Binäre Suchbäume
  - Gefädelte Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume

Warum Bäume?

Hashverfahren erlauben Suche, Einfügen und Löschen in konstanter Zeit,

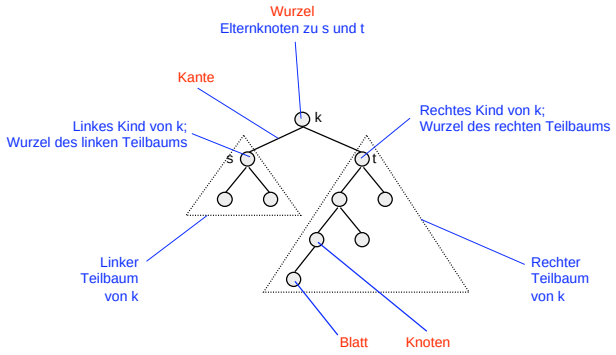
aber: Operationen, für die die Reihenfolge der Daten wichtig ist, werden nicht unterstützt.

Beispiele:
• finde Minimum, Maximum
• Analyse/Ausgabe der Daten in der korrekten Reihenfolge



Binäre Suchbäume erlauben sortierte Bearbeitung in O(n log n), Suche, Einfügen, Finden von Maxima und Minima und Löschen immerhin in durchschnittlich O(log n).

Bäume: Begriffe und Eigenschaften (1)



Binärbaum: Jeder Knoten hat 2 Kinder (linkes und rechtes Kind), oder 1 Kind (linkes oder rechtes Kind) oder keine Kinder.

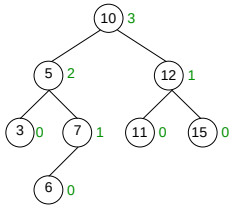
Baum: Jeder Knoten kann beliebig viele Kinder haben

Bäume: Begriffe und Eigenschaften (2)

Höhe eines Baums

Maximale Anzahl von Kanten von seiner Wurzel zu einem Blatt.

Beispiel



Höhe für jeden Teilbaum

Beachte

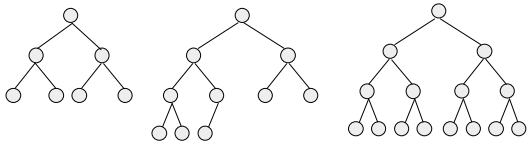
- Ein Baum, der nur aus einem Knoten besteht, besitzt die Höhe 0.
- Aus technischen Gründen wird die Höhe eines leeren Baums (d.h. Anzahl Knoten = 0) als -1 definiert.

Bäume: Begriffe und Eigenschaften (3)

Vollständiger Binärbaum

Ein vollständiger Binärbaum ist ein Binärbaum, bei der jeder Ebene (bis auf die letzte) vollständig gefüllt und die letzte Ebene von links nach rechts gefüllt ist.

Beispiele



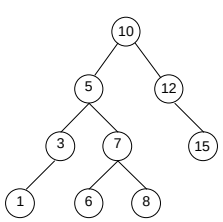
Eigenschaften:

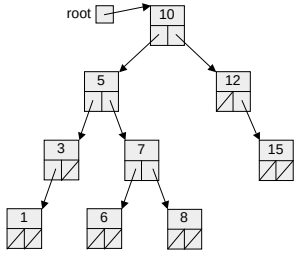
- Ein vollständiger Binärbaum mit n Knoten hat die Höhe $h = \lfloor \log_2 n \rfloor$.
- Vollständige Binärbäume sind (bei einer gegebenen Knotenzahl) Binärbäume mit einer minimalen Höhe.

Bäume: Begriffe und Eigenschaften (4)

Implementierung von Binärbäumen mit verketteten Knoten

Jeder Knoten hat jeweils einen Zeiger für das linke bzw. rechte Kind.






```
struct Node
{
      int data;
      Node* left;     // linkes Kind
      Node* right;    // rechtes Kind
};
```


Traversierung von Bäumen (1)

Ziel

Das Besuchen aller Knoten in einer bestimmten Reihenfolge
ist eine oft benötigte Operation.

Durchlaufreihenfolgen

- PreOrder: besuche Wurzel, besuche linken Teilbaum; besuche rechten Teilbaum;
- PostOrder: besuche linken Teilbaum; besuche rechten Teilbaum; besuche Wurzel;
- InOrder: besuche linken Teilbaum; besuche Wurzel; besuche rechten Teilbaum;
- LevelOrder: besuche Knoten ebenenweise

Bemerkungen

- Die Präfixe *Pre*, *Post* bzw. *In* bedeuten vorher, nachher und dazwischen.
  Gemeint ist damit der Zeitpunkt, zu dem die Wurzel besucht wird.
- InOrder-Durchlauf ist nur für Binärbäume sinnvoll.

Traversierung von Bäumen (2)

PreOrder-Traversierung

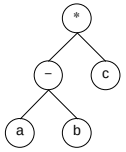
```
void preOrder(Node* p)
{
      if (p != 0)
      {
            bearbeite(p->data);
            preOrder(p->left);
            preOrder(p->right);
      }
}
```

```
// Definition eines Baums:
struct Node
{
      int data;
      Node* left;     // linkes Kind
      Node* right;    // rechtes Kind
};

Node* root;
…

// Aufruf von preorder
preOrder(root);
```

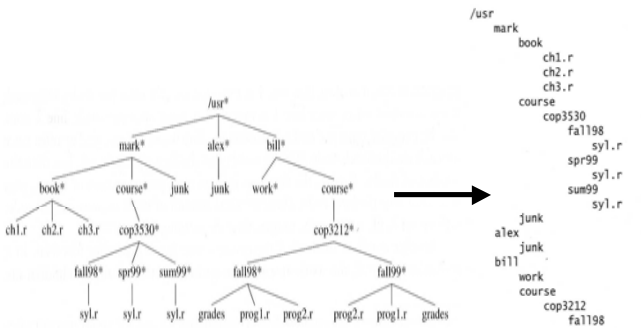
Beispiel



Bearbeitungsreihenfolge:

* - a b c

Directory-Liste durch Preorder-Traversierung



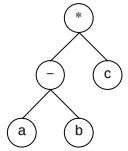
(Weiss, 1999)

Traversierung von Bäumen (3)

PostOrder-Traversierung

```
void postOrder(Node* p)
{
    if (p != 0)
    {
        postOrder(p->left);
        postOrder(p->right);
        bearbeite(p->data);
    }
}
```

Beispiel

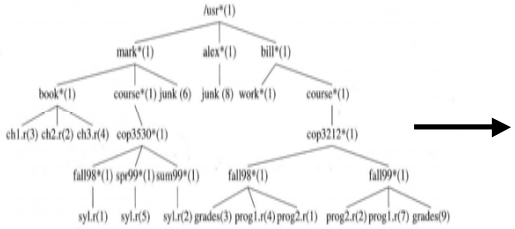


Bearbeitungsreihenfolge:

a b – c *

(Entspricht der sog. Postfix-Notation für arithmetische Ausdrücke)

Size(directory) durch Postorder-Traversierung



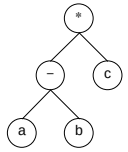
(Weiss, 1999)

Traversierung von Bäumen (4)

InOrder-Traversierung

```
void inOrder(Node* p)
{
    if (p != 0)
    {
        inOrder(p->left);
        bearbeite(p->data);
        inOrder(p->right);
    }
}
```

Beispiel



Durchlaufreihenfolge:

a – b * c

(als mathematischer Ausdruck interpretiert, erhält man (a - b) * c)

Traversierung von Bäumen (5)

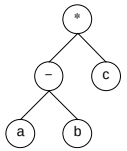
Level-Order-Traversierung

Die Knoten werden ebenenweise in einer Schlange (Queue) gespeichert und in einer while-Schleife abgearbeitet.

```
void levelOrder(Node* p)
{
    Queue<Node*> queue;
    queue.push(p);     // Ebene 0

    while ( !queue.empty() )
    {
        Node* q;
        // Schreibe vorderstes Element aus Schlange
        // nach q und lösche Element aus Schlange:
        queue.front(q);
        queue.remove();
        if (q != 0)
        {
            bearbeite(q->data);
            queue.insert(q->left);
            queue.insert(q->right);
        }
    }
}
```

Beispiel



Durchlaufreihenfolge:

* – c a b

Definition von binären Suchbäumen

Voraussetzung

Alle Knoten in einem Baum enthalten einen Schlüssel (Key) und Nutzdaten (value).

```
struct Node
{
    KeyType key;
    ValueType value;
    Node* left;    // linkes Kind
    Node* right;   // rechtes Kind
};
```

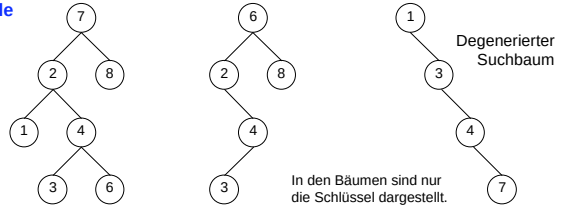
Definition

Ein binärer Suchbaum ist ein Binärbaum, bei dem für alle Knoten k folgende Eigenschaften gelten:

(1) Alle Schlüssel im linken Teilbaum sind kleiner als der Schlüssel im Knoten k

(2) Alle Schlüssel im rechten Teilbaum sind größer als der Schlüssel im Knoten k

Beispiele



In den Bäumen sind nur die Schlüssel dargestellt.

Suchen in binären Suchbäumen

Operation searchR

```
bool searchR(KeyType k,  ValueType& v,  const Node* p);
```

Suche nach einem Knoten mit Schlüssel k im Teilbaum p.

Falls gefunden, wird der im Knoten abspeicherte Datenwert v und der Rückgabewert true zurückgeliefert.

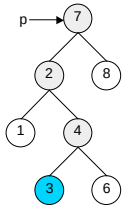
Falls nicht gefunden, wird der Rückgabewert false zurückgeliefert.

Algorithmus

```
bool searchR(KeyType k,  ValueType& v,  const Node*
p)
{
    if (p == 0)
        return false;
    else if (k < p->key)
        return searchR(k, v, p->left);
    else if (k > p->key)
        return searchR(k, v, p->right);
    else // k gefunden
    {
        v = p->value;
        return true;
    }
}
```

R steht für rekursiv

Beispiel



searchR(3,v,p)

Einfügen in binären Suchbäumen (1)

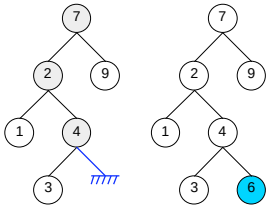
Idee

Um einen Schlüssel k einzufügen, wird zunächst nach dem Schlüssel k gesucht.

Falls der einzufügende Schlüssel k nicht bereits im Baum vorkommt, endet die Suche erfolglos bei einem 0-Zeiger.

An dieser Stelle wird dann ein neuen Knoten mit Schlüssel k eingefügt.

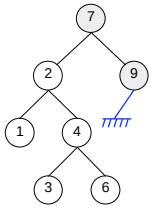
Beispiel 1: füge 6 ein



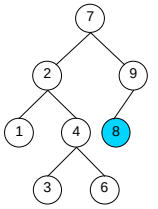
Suche von 6 endet bei 0-Zeiger

Ersetzte 0-Zeiger durch Zeiger auf Knoten 6

Beispiel 2: füge 8 ein



Suche von 8 endet bei 0-Zeiger



Ersetzte 0-Zeiger durch Zeiger auf Knoten 8

Einfügen in binären Suchbäumen (2)

Operation insertR

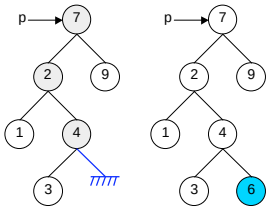
```
bool insertR(KeyType k, ValueType v, Node*& p);
```

Fügt im Teilbaum p neuen Knoten mit Schlüssel k und Datenwert v ein.
Falls Schlüssel schon vorhanden, wird kein neuer Knoten eingefügt.

Beachte:
Der Parameter p ist ein Ein/Ausgabeparameter und daher als Referenzparameter realisiert. Der Teilbaum wird gelesen und geändert.

Algorithmus

```
bool insertR(KeyType k, ValueType v, Node*& p)
{
    if (p == 0) {
        p = new Node;
        p->key = k; p->value = v;
        return true;
    }
    else if (k < p->key)
        return insertR(k,v,p->left);
    else if (k > p->key)
        return insertR(k,v,p->right);
    else    // k bereits vorhanden
        return false;
}
```



insert(6, v, p)

Löschen in binären Suchbäumen (1)

Idee

Um einen Schlüssel k zu löschen wird zunächst nach dem Schlüssel k gesucht.

Es sind dann 4 Fälle zu unterscheiden:

Fall „Nicht vorhanden“:

Schlüssel k kommt nicht vor:
dann ist nichts zu tun.

Fall „Keine Kinder“:

Der Schlüssel kommt in einem Blatt vor (keine Kinder):
dann kann der Knoten einfach entfernt werden.

Fall „Ein Kind“:

Der Knoten mit dem gefundenen Schlüssel hat genau ein Kind:
s. nächste Folie

Fall „Zwei Kinder“:

Der Knoten mit dem gefunden Schlüssel hat zwei Kinder
s. übernächste Folie

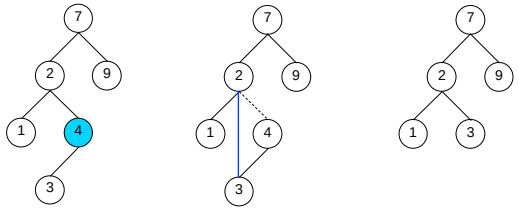
Löschen in binären Suchbäumen (2)

Fall „Ein Kind“

Der zu löschende Knoten k hat genau ein Kind.

Überbrücke den Knoten k, indem der Elternknoten von k auf Kind von k verzeigert wird (Bypass) und lösche k.

Beispiel: lösche Knoten mit Schlüssel 4



Suche 4 | Knoten 4 wird überbrückt und gelöscht | Knoten 4 wurde gelöscht

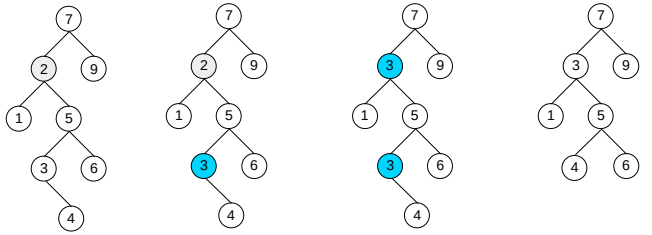
Löschen in binären Suchbäumen (3)

Fall „Zwei Kinder“ :

Der zu löschende Knoten k hat zwei Kinder.

1. Ersetze den Knoten k durch den kleinsten Knoten k_{min} im rechten Teilbaum von k.
2. Lösche dann k_{min}. Da der kleinste Knoten k_{min} im linken Teilbaum kein linkes Kind hat, kann das Löschen von k_{min} wie im Fall „Ein Kind“ bzw. „Keine Kinder“ behandelt werden.

Beispiel: lösche Knoten mit Schlüssel 2



Suche 2 | Suche kleinsten Knoten in rechten Teilbaum von 2 | Knoten 2 wurde durch kleinsten Knoten ersetzt | kleinster Knoten wurde gelöscht

Löschen in binären Suchbäumen (4)

```
bool removeR(KeyType k, Node*& p)
{
    if (p == 0)   // k nicht vorhanden
        return false;
    else if (k < p->key)
        return removeR(k,p->left);
    else if (k > p->key)
        return removeR(k,p->right);
    else if (p->left == 0 || p->right == 0) {
        Node* temp = p;
        if (p->left != 0)
            p = p->left;    // Bypass zu linkes Kind
        else
            p = p->right;   // Bypass zu rechtes Kind
        delete temp;
        return true;
    }
    else {
        // Min. im rechten Teilbaum suchen:
        Node* min = searchMinR (p->right);
        // Zu löschender Knoten durch Min. ersetzen
        p->data = min->data;   p->key = min->key;
        // Min. in rechtem Teilbaum löschen:
        return removeR (min->key, p->right);
    }
}
```

Operation removeR

Löscht im Teilbaum p Knoten mit Schlüssel k.

Der Parameter p ist ein Ein/Ausgabeparameter und daher ein Referenzparameter.

Knoten hat ein Kind oder kein Kind

Knoten hat zwei Kinder

Löschen in binären Suchbäumen (5)

Operation searchMinR

Sucht im Teilbaum p nach dem kleinsten Knoten und liefert Zeiger auf Minimum zurück.

```
Node* searchMinR(KeyType k,  const Node* p)
{
    if (p == 0)   // k nicht vorhanden
        return 0;
    else if (p->left == 0)   // Minimum gefunden
        return p;
    else
        return searchMinR (k, p->left);
}
```

Bemerkung

Beachten Sie, dass in der Operation removeR im Fall „Knoten hat zwei Kinder“ der Aufruf von searchMinR und der rekursive Aufruf von removeR bedeuten, dass zweimal vom rechten Kind p->right zum kleinsten Knoten gelaufen wird.

Diese Ineffizienz lässt sich jedoch beheben, indem searchMinR noch zusätzlich das Löschen des kleinsten Elements übernimmt. Der rekursive Aufruf von removeR ist dann überflüssig.

Diese Variante findet sich in der Quell-Code-Sammlung zur Vorlesung.

Gefädelte Suchbäume (1)

Problem

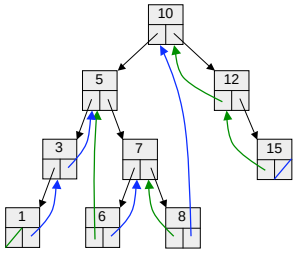
In binären Suchbäumen gibt es für einen Knoten im allgemeinen keinen effizienten Zugriff auf seinen InOrder-Vorgänger bzw. -Nachfolger.

Sollen Suchbäume beispielsweise für assoziative STL-Container eingesetzt werden, ist jedoch eine effiziente Vorwärts- und Rückwärtstraversierung mit Iteratoren notwendig.

Lösung

Ersetze in jedem Knoten den Links- bzw. Rechts-Zeiger mit dem Wert 0 durch einen Zeiger auf seinen InOrder-Vorgänger bzw. Nachfolger (Fädelung).

Ob ein Zeiger auf ein Kind oder ein Vor- bzw. Nachfolger zeigt, kann durch zusätzliche boolsche Variable vermerkt werden (siehe nächste Aufgabe).



Gefädelte Suchbäume (2)

Aufgabe 1.4

Ein gefädelter Suchbaum lässt sich beispielsweise durch folgenden Strukturdatentyp realisieren.

```
struct Node
{
    int data;
    Node* left;         // linkes Kind bzw. Vorgänger
    Node* right;        // rechtes Kind bzw. Nachfolger
    bool leftThread;    // true, falls left auf InOrder-Vorgänger zeigt
    bool rightThread;   // analog.
};
```

Schreiben Sie für gefädelte Suchbäume nicht-rekursive C++-Funktionen für folgende Aufgaben:

a) Bestimme den In-Order-Nachfolger zu einem Knoten.

b) Bestimme den In-Order-Vorgänger zu einem Knoten

c) Ausgabe aller Knoten in In-Order-Reihenfolge.

Analyse von binären Suchbäumen

Worst-Case

Da im schlechtesten Fall ein binärer Suchbaum mit n Knoten zu einem Baum der Höhe n-1 entarten kann (Bsp?), haben die Operationen zum Suchen, Einfügen und Löschen eine maximale Suchlänge von n.

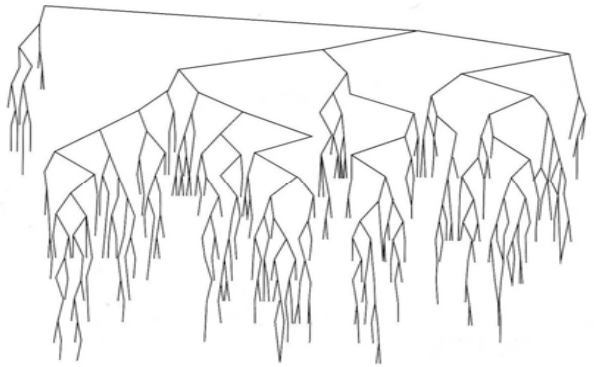
Damit: $\mathbf{T_{max}(n) = O(n)}$

Average-Case

In [Ottmann und Widmayer 2002] wird gezeigt, dass die durchschnittliche Laufzeit um eine Größenordnung besser ist. Es werden zwei Ergebnisse hergeleitet, die sich darin unterscheiden, welche Verteilung der Bäume angenommen wird:

- Bäume mit n Knoten entstehen durch eine Folge von Einfüge-Operationen von n unterschiedlichen Elementen. Es wird angenommen, dass jede der n! möglichen Anordnungen der Elemente gleichwahrscheinlich ist. Gemittelt wird dann über die n! viele auf diese Weise erzeugten Bäume. Man erhält dann für die Such-Operation eine mittlere Suchlänge von ungefähr 1.4 $\log_2 n$.
  Damit: $\mathbf{T_{mit}(n) = O(\log_2 n)}$
- Es wird angenommen, dass alle strukturell verschiedenen binären Suchbäume mit n Knoten gleichwahrscheinlich sind. Man erhält dann für die Such-Operation eine mittlere Suchlänge von ungefähr $\sqrt{\pi * n}$
  Damit: $\mathbf{T_{mit}(n) = O(\sqrt{n})}$

Zufällig erzeugter Baum

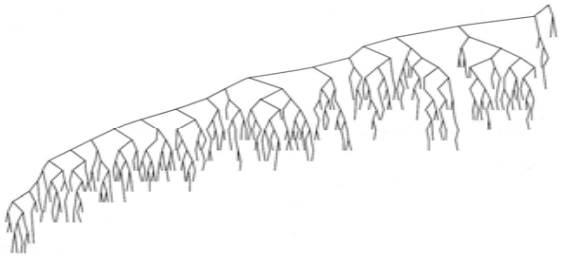


500 Zufallszahlen in binären Suchbaum eingefügt =>
Tiefe ist tatsächlich nahe an log n (aus Weiss, 1999)

Probleme bei binären Suchbäumen

Die Annahme von zufällig verteilten Schlüsseln bzw. gleichwahrscheinlichen Baumstrukturen ist nicht immer erfüllt:

- oft werden Daten unabsichtlich in geordeneter Form eingegeben
- *delete* favorisiert Bäume mit tieferen linken Unterbäumen, da ein gelöschter Knoten immer durch den minimalen Knoten des rechten Unterbaums ersetzt wird.



Binärer Suchbaum nach N^2 Paaren von insert und remove (aus Weiss, 99)