Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

9. Vorlesungswoche

Dynamische Datenstrukturen:

Listen, Bäume, Graphen, Schlangen, Keller, Mengen

Unterlagen:

Echtle, Goedicke; Einführung in die objektorientierte Programmierung mit Java, dpunkt-Verlag.

Doberkat, Dissmann; Einführung in die objektorientierte Programmierung mit Java, Oldenbourg-Verlag, 2. Auflage.

Folien nach V.Gruhn, Vorlesung Programmierung WS 99/00

Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Überblick

- Dynamische Datenstrukturen:
  - Strukturen, die je nach Bedarf und damit dynamisch wachsen und schrumpfen können (Unterschied zu Felder/Arrays!).
- Grundidee:
  - Dynamische Datenstrukturen bilden Mengen mit typischen Operationen ab
  - Einzelne Elemente speichern die zu speichernden / zu verarbeitenden Daten
  - Einzelne Elemente werden durch dyn. Datenstrukturen verknüpft

  also: Trennung von Datenverwaltung & Speicherung.
- Art der Elemente ist problemabhängig, variiert je nach Anwendung
- Für die Verknüpfung existieren typische Muster

  => dynamische Datenstrukturen: Liste, Baum, Graph, ...

  mit zugehörigen Zugriffsmethoden
- Objektorientierte Sicht: dynamische Datenstrukturen durch die Art der Verknüpfung der Elemente und die Zugriffsmethoden charakterisiert.

Wichtige dynamische Datenstrukturen

- Listen, lineare Listen, doppelt verkettete Listen
- Bäume, binäre Bäume, binäre Suchbäume
- Graphen, gerichtete Graphen, ungerichtete Graphen
- Stack, Schlangen
- Mengen

Fragen zur Organisation der Datenstrukturen:

– Wie wird eine Instanz der Struktur initialisiert, Daten eingefügt, modifiziert, entfernt ?
– Wie wird in den Strukturen navigiert ?
– Wie werden einzelne Werte in einer Struktur wiedergefunden ?
– Wie werden alle in einer Struktur abgelegten Werte besucht ?

Aus Sicht einer Menge: Vereinigung, Schnittmenge (mit einelementigen Mengen)

Zugriff über Referenzen

- Eigenschaften von Referenzen:

```
Geld betrag;
if (betrag != null)
  betrag.Drucke();
else
  betrag = new Geld(10, 25);
Geld preis;
preis = new Geld(48, 98);
preis = null;
```

Variable betrag: Referenz auf Objekt der Klasse Geld

Soll eine Referenz auf kein Objekt verweisen, wird der Wert null zugewiesen

Überprüfung der Referenz mittels Vergleichsoperator

Der Wert null für Referenzen ermöglicht ein Objekt explizit von einer Referenz zu lösen

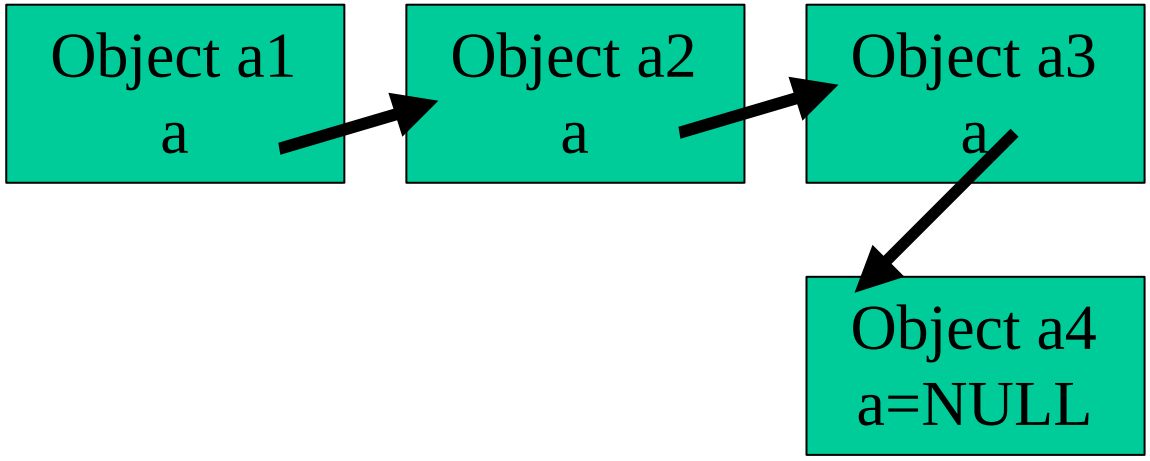
Referenz preis verweist auf Objekt der Klasse Geld

Zuweisung auf null vernichtet Verweis auf Objekt mit Werten 48 und 98

Grundlage für den Aufbau dynamischer Datenstrukturen

- Klassen enthalten Attribute, die Referenzen auf Objekte der eigenen Klasse darstellen.
- Diese Attribute schaffen die Möglichkeit, an eine Referenz ein weiteres Objekt der Klasse zu binden.
- Die einzelnen Objekte sind in der Lage, gemeinsam eine komplexe Struktur durch aufeinander verweisende Referenzen zu bilden.

```
class A {
  class A a ;
  ...
}
```



Allgemeines zu Listen

- Listen definieren eine Reihenfolge von Elementen, die gemäß dieser Reihenfolge miteinander verknüpft sind.
- Typische Zugriffsmethoden:
  - Einfügen am Anfang
  - Einfügen an bestimmter Stelle
  - Anfügen (d.h. Einfügen am Ende)
  - Ermittlung der Länge
  - Prüfen, ob die Liste leer ist
  - Prüfen, ob Element in Liste vorkommt
  - Ermittlung der Position eines Elements
  - Ermittlung des ersten Elements
  - Liefern der Liste ohne erstes Element
- Die Verfügbarkeit aller dieser Methoden variiert mit dem Anwendungszweck. Nicht immer sind alle Methoden realisiert, aber man redet dennoch von Listen

Bestandteile einer Liste

```
class Element {
   Element(int i) { wert = i; weiter = null; }
   private int wert;
   private Element weiter;
}
```

- Deklaration einer Klasse **`Element`** mit zwei privaten Attributen und einem Konstruktor
- Ein Objekt vom Typ Element enthält als Attribute eine ganze Zahl und einen Zeiger auf ein weiteres Objekt des Typs **`Element`**
- Jedes Objekt vom Typ **`Element`** besitzt eine Referenz auf ein weiteres Element, man kann sie miteinander verketten
- Die daraus entstehende Datenstruktur ist eine *Lineare Liste*

Schema der Klasse lineare Liste

```
class Liste {
  private Element kopf;

   Liste() { kopf = null; }
  Liste(int w) { kopf = new Element(w); }

   void FügeAn(int an) {
        ...
  }
  void FügeEin(int ein) {
        ...
   }
```

Erweiterung einer Liste

Eine lineare Liste kann auf verschiedene Arten konstruiert werden :

- neues Element an den Anfang, in die Mitte oder an das Ende einer bereits bestehenden Liste anhängen
- Zugriff auf die Liste wird durch eine Referenz realisiert, die auf das erste Element der Liste zeigt
- Enthält eine Liste keine Elemente, zeigt die Referenz auf **`null`**
- Besuchen eines Elements innerhalb der Liste erfordert eine Referenz von Element zu Element

Erweiterung einer Liste

Programm: **Einfügen am Ende einer linearen Liste**

```
void FügeAn(int neuerWert) {
  Element lauf = this;
  while (lauf.weiter != null)
       lauf = lauf.weiter;
  lauf.weiter = new Element(neuerWert);
}
```

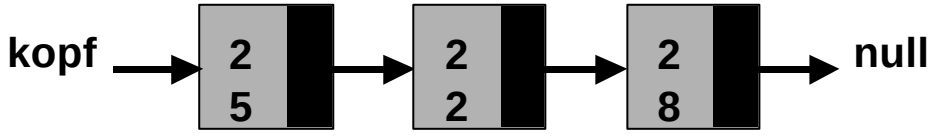
- Die Klasse **`Element`** wird um die Methode **`FügeAn`** ergänzt, die ein neues Element an das Ende einer Liste anhängt, die bereits aus wenigstens einem Element besteht.
- Die Liste wird durch die lokale Referenz **`lauf`** von Element zu Element durchlaufen, bis die Referenz **`weiter`** auf **`null`** verweist.
- Nun wird auf das neu erzeugte Objekt der Klasse **`Element`** verwiesen.
- Die Referenz **`this`** verweist immer auf das Objekt, für welches die Methode **`FügeAn`** aufgerufen wurde, daher verweist **`this`** nie auf den Wert **`null`**.

Initiales Anlegen einer linearen Liste

Anlegen einer linearen Liste:

- Mit dem Konstruktor von **`Element`** wird ein erstes Objekt geschaffen.
- Alle weiteren Listenelemente werden durch **`FügeAn`** für das erste Element angefügt.

```
Element kopf;
kopf = new Element(25);
kopf.FügeAn(22);
kopf.FügeAn(28);
```



Rekursives Anfügen

Listen erlauben elegante rekursive Funktionen:

Programm: Einfügen auf der Basis eines rekursiven Vorgehens

```
void RekFügeAn(int neuerWert) {
 if (weiter != null)
     weiter.RekFügeAn(neuerWert);
 else
     weiter = new Element(neuerWert);
}
```

Beachte:

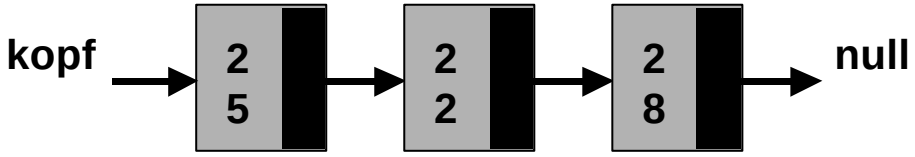
Die Ausführung der Methode **RekFügeAn** ist aufwendiger, als die bisher vorgestellten Methoden, da alle Aufrufe von **RekFügeAn** ineinander geschaltet sind und erst dann beendet werden, wenn das neue Element angefügt ist.

Konstruktion einer Liste durch weiteres Anfügen

Programm: **Methode `FügeEin`**

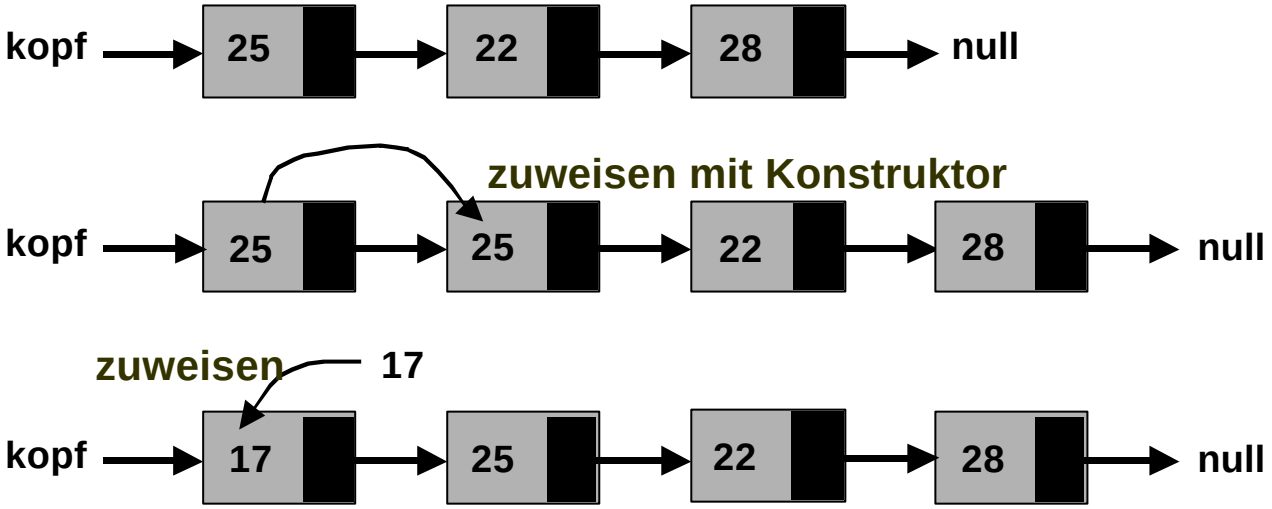
```
void FügeEin(int neuerWert) {
  Element neuesElement = new Element(wert);
  neuesElement.weiter = weiter;
  weiter = neuesElement;
  wert = neuerWert;
}
```

`kopf.FügeEin(17)` wird nun auf die Liste



angewendet

Konstruktion einer Liste durch Einfügen



Alles andere würde bedeuten, dass **kopf** umgesetzt wird.

Das aber würde bedeuten, dass ein Element den Zugriff auf sich abtritt.

Das geht nicht, also bleibt nur der Weg über das Einfügen an der zunächst zweiten Stelle.

Klasse lineare Liste

Umsetzen von linearen Listen mit der Klasse **Element** und den Methoden **FügeAn** und **FügeEin** :

- **Aufbau einer Liste aus einer „leeren“ Liste:**

  Da die Methoden **`FügeAn`** und **`FügeEin`** Bestandteile der Objekte der Klasse **`Element`** sind, können sie nur dann aufgerufen werden, wenn auch ein solches Element besteht. Daher muß das erste Element einer Liste immer über einen Konstruktor erzeugt werden, alle weiteren können dann über die Methoden hinzugefügt werden. Innerhalb eines Programms muß daher vor jedem Aufruf von **`FügeAn`** oder **`FügeEin`** überprüft werden, ob die Referenz auf die Liste auf **`null`** verweist. Übersichtlicher wäre es hingegen, in allen Situationen Elemente durch die Methoden **`FügeAn`** und **`FügeEin`** hinzufügen zu können.

Klasse lineare Liste

Änderung des ersten Elementes:

Die Implementierung der Methode **`FügeEin`** hat gezeigt, daß das Hinzufügen eines neuen Objektes als erstes Element der Liste nicht möglich ist; hierfür muß auf entsprechende Zuweisungen zurückgegriffen werden. Ebenso problematisch ist das Löschen des ersten Elementes einer Liste. Insbesondere das letzte Element einer Liste läßt sich nicht durch eine Methode der Klasse **`Element`** entfernen.

Effizienz der Methode `FügeAn`:

Die Methode **`FügeAn`** erfordert bei jedem Aufruf ein vollständiges Durchlaufen der Liste. Eine sehr viel effizientere Realisierung dieser Listenoperation wäre möglich, wenn neben dem ersten Element auch das letzte Element der Liste unmittelbar erreichbar wäre.

Verbesserungsmöglichkeiten

- Die Idee, eine Liste mit einer Referenz auf ihr erstes Element gleichzusetzen, wird dem Umgang mit der entstehenden Datenstruktur nicht gerecht.
- Der Wertebereich einer Klasse, die lineare Listen implementiert, sollte auch die leere Liste beinhalten und für diese eine korrekte Anwendung der Methoden garantieren.
- Die Ausführung von Methoden sollte durch zusätzliche Referenzen auf ausgewählte Elemente der Liste unterstützt werden.
- Als Ergebnis entstehen die folgenden, modifizierten Klassen **`Element`** und **`Liste`**

Klasse Element (verbessert)

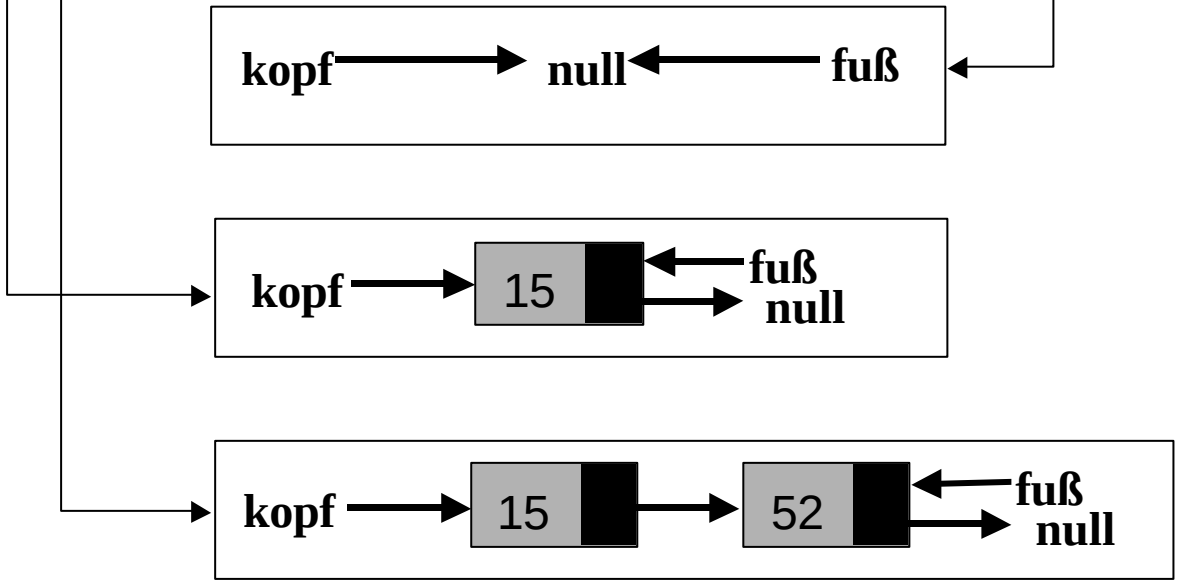
```
class Element {
  private int wert;
  private Element weiter;

  Element(int i) { wert = i; weiter = null; }
  Element(int i, Element e) { wert = i; weiter = e; }

  void SetzeWert(int i) {
       wert = i;
  }
  int GibWert() {
       return wert;
  }
  void SetzeWeiter(Element e) {
       weiter = e;
  }
  Element GibWeiter() {
       return weiter;
  }
}
```

Klasse lineare Liste (verbessert)

```
class Liste {
  private Element kopf, fuß;
  Liste() { kopf = fuß = null; }
  Liste(int w) { kopf = fuß = new Element(w); }
  void FügeAn(int an) {
        Element neu = new Element(an);
        if (fuß != null) {
              fuß.SetzeWeiter(neu);
              fuß = neu;
        }
        else
              kopf = fuß = neu;
  }
  void FügeEin(int ein) {
        kopf = new Element(ein, kopf);
        if (fuß == null)
              fuß = kopf;
        }
}
```

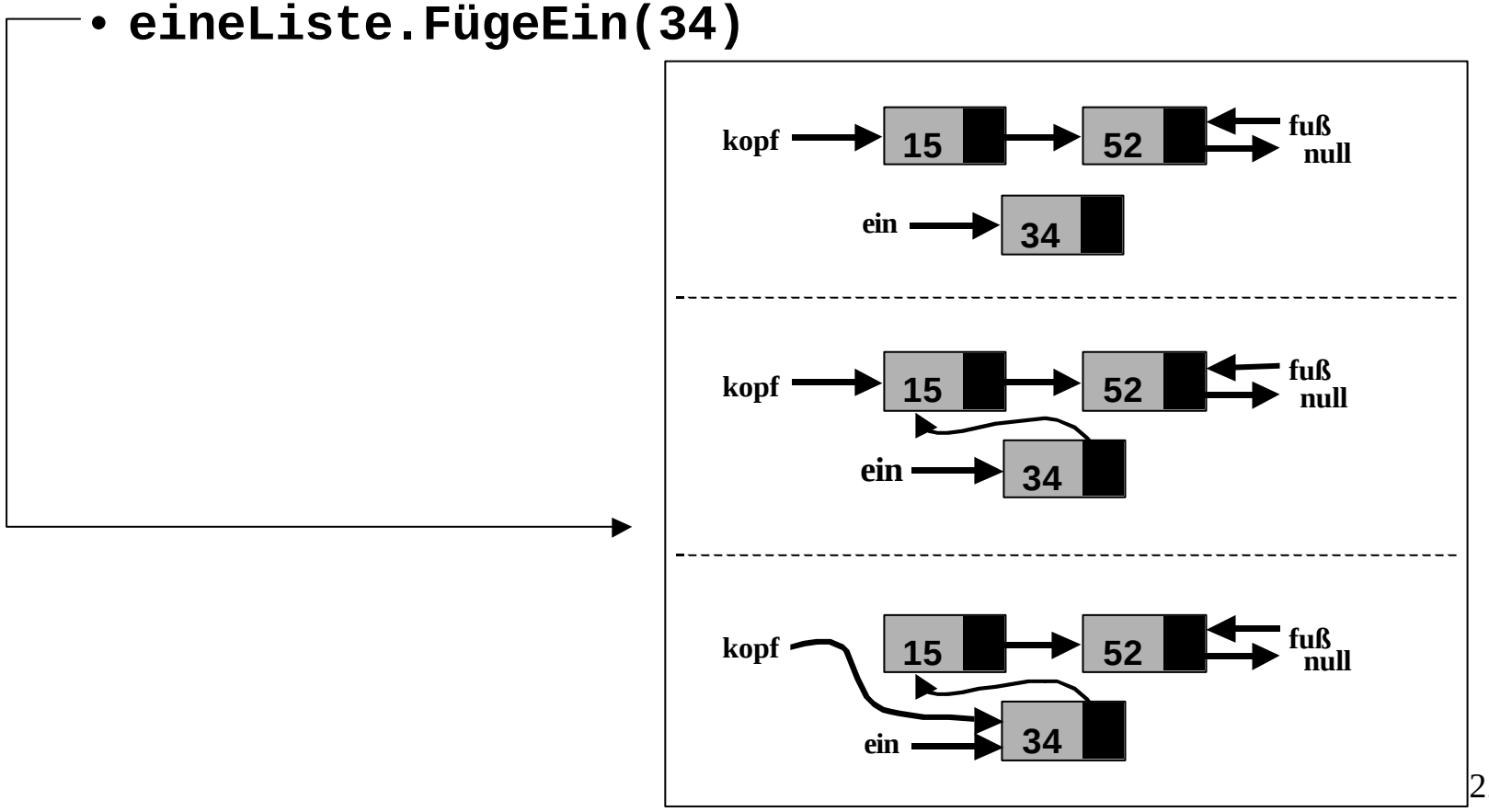
Beispiel Anfügen und Einfügen

- `Liste eineListe = new Liste();`
- `eineListe.FügeAn(15);`
- `eineListe.FügeAn(52);`
- `eineListe.FügeEin(34)`



Beispiel Anfügen und Einfügen

- `Liste eineListe = new Liste();`
- `eineListe.FügeAn(15);`
- `eineListe.FügeAn(52);`
- `eineListe.FügeEin(34)`



Anwendungsbeispiel

- Einordnen eines Werts in eine aufsteigend geordnete Liste
- Keine zwei Elemente haben die identische Belegung des Attributs **`wert`**
- Der Algorithmus ist auf natürliche Weise rekursiv

Idee:

- sei *x* der einzufügende Wert
- 1. Fall: *x* kleiner als 1. Element => Einfügen am Anfang
- 2. Fall: *x* grösser als 1. Element
  - suche passende Position in der Liste
  - trenne Liste in Anfangs- und Endteil auf
  - setze Element an den Anfang des Endteils und verbinde Teillisten
  - Falls *x* größer als letztes Element der Liste, ist x der Endteil

Anwendungsbeispiel

- Präzisierung des Algorithmus:
- Folgende Fälle sind zu unterscheiden:

`kopf == null:`

Einen Sonderfall bildet die Situation, daß die Liste leer ist, also noch kein Element enthält. Es muß ein erstes Element angelegt werden, das sicherlich eine geordnete, einelementige Liste bildet.

`kopf != null:`

Wir definieren eine private Methode **`Positioniere`**, die als Parameter den einzuordnenden Wert und eine Referenz auf den Anfang einer Teilliste übergeben bekommt. Als Ergebnis gibt **`Positioniere`** eine Referenz auf **`Element`** zurück, die auf die Teilliste verweist, in die *x* einsortiert ist.

Anwendungsbeispiel

Sei **`anfang`** die an **`Positioniere`** übergebene Teilliste und gelte:

`x < anfang.wert:`

Erzeuge ein neues Element und füge es am Anfang der bei **`anfang`** beginnenden Teilliste ein.

`x > anfang.wert:`

Füge x in die mit **`anfang.weiter`** beginnenden Restliste ein, indem hierfür **`Positioniere`** mit den entsprechenden Parametern erneut aufgerufen wird.

Anwendungsbeispiel

Beachte:

- Keine doppelten Einträge werden zugelassen.
- Die Referenz **`fuß`** verweist auch nach dem Einsortieren auf das letzte Element.
- Wenn **`Positioniere`** die leere Referenz **`null`** als Wert für den Parameter **`anfang`** übergeben bekommt, muß **`fuß`** korrigiert werden und auf das neu eingeordnete letzte Objekt gesetzt werden.

Einordnen mit Hilfe von `Positioniere`

***Programm:* Einordnen in eine geordnete Liste**

```
class Liste {
  private Element kopf, fuß;
  Liste() { kopf = fuß = null; }

  Liste(int w) { kopf = fuß = new Element(w); }
  …
  void OrdneEin(int i) {

       kopf   = Positioniere(kopf, i);
  }
```

Einordnen mit Hilfe von `Positioniere`

```
private Element Positioniere(Element anfang, int i) {
      if (anfang == null)

            fuß = anfang = new Element(i);
      else {
            if (i < anfang.GibWert()) {
                  anfang = new Element(i, anfang);

            }
            if (i > anfang.GibWert())
                  anfang.SetzeWeiter
                  (Positioniere(anfang.GibWeiter(), i));
      }

      return anfang;
   }
   …
}
```

Durchlaufen einer Struktur

- In vielen Anwendungen, die auf dynamischen Datenstrukturen basieren, besteht die Notwendigkeit, alle Elemente der Struktur genau einmal zu besuchen. Dies gilt für Listen wie für andere dynamische Strukturen.
- Dieses möglichst nur einmalige Besuchen aller Elemente nennt man Durchlaufen einer Struktur.
- Anwendungsbeispiele: Prüfen auf Vorhandensein, Einsortieren
- Konkrete Ausprägungen dieses Problems spielen in der theoretischen Informatik eine wichtige Rolle (Travelling Salesman Problem).

Durchlaufen einer Liste

Programm: **Rekursive Methode zum Drucken einer linearen Liste**

```
class Liste {
  …
  void RekDrucke() {
      RekDrucke(kopf);
  }
  private void RekDrucke(Element aktuell) {
      if (aktuell != null) {
            System.out.println(aktuell.GibWert());
            RekDrucke(aktuell.GibWeiter());
      }
  }
  …
}
```

Durchlaufen einer Liste

```
void IterDrucke() {
      Element aktuell = kopf;
      while (aktuell != null) {
            System.out.println(aktuell.GibWert());
            aktuell = aktuell.GibWeiter();
      }
  }
```

Durchlaufen einer Liste

Durchlauf einer Liste in umgekehrter Reihenfolge:

- Referenz **`fuß`** verweist zwar auf das letzte Element einer Liste, kann jedoch nicht von dort zum vorletzten Element gelangen.
- Für eine umgekehrte Ausgabe müssen alle Listenelemente gemerkt werden, während die Liste vom Anfang zum Ende durchläuft.
- Erst nach einmaligem Durchlaufen kann vom letzten bis zum ersten Element gedruckt werden.

Durchlaufen einer Liste

Programm: **Ausgabe einer Liste in umgekehrter Reihenfolge**

```
void  ReversivDrucke() {
   ReversivDrucke(kopf);
}
private void ReversivDrucke(Element aktuell) {
   if (aktuell != null) {
       ReversivDrucke(aktuell.GibWeiter());
       System.out.println(aktuell.GibWert());
   }
}
```

Doppelt verkettete Listen

- Ist der Durchlauf vom Ende einer Liste zu ihrem Anfang häufig benötigt, dann ist die lineare Verkettung von vorne nach hinten nicht der ideale Navigationspfad.
- Besser wäre es dann auch eine Rückwärtsverkettung zu haben.
- Auf Grund dieser Überlegung kommt man zu doppelt verketteten Liste (einmal von vorne nach hinten, einmal umgekehrt).
- Die lokale Klasse **`Element`** enthält eine zweite Referenz **`voran`**, die genau entgegengesetzt zu **`weiter`** gerichtet
  ist und somit für jedes Element innerhalb der Liste auf seinen direkten Vorgänger verweist.
- Mit doppelt verketteten Listen kann in beide Richtungen einer Liste navigiert werden und deshalb komplexe Operationen auf einer Liste unterstützen.
- In Java: **`LinkedList`** (Unterklasse von **`AbstractSequentialList`** in java.util)

Doppelt verkettete Listen

```
class DElement {
   …            // die bekannten Deklarationen der linearen Liste
   private DElement voran, weiter;
   void SetzeVoran(DElement e) { voran = e; }
   Element GibVoran() { return voran; }
}
class DListe {
   …            // die bekannten Deklarationen der linearen Liste
   void FügeAn(int an) {
        DElement neu;
        neu = new DElement(an);
        if (fuß != null) {
                fuß.SetzeWeiter(neu);
                neu.SetzeVoran(fuß);
                fuß = neu;
        } else
                kopf = fuß = neu;
   }
   void OrdneEin(int i) {

    kopf        = Positioniere(kopf, i);
}
```

Doppelt verkettete Listen

```
private DElement Positioniere(DElement anfang, int i) {
        if (anfang == null)

                anfang = new DElement(i);
                anfang.SetzeVoran(fuß);
                fuß = anfang;
        else {
                if (i < anfang.GibWert()) {
                        DElement neu = new DElement(i, anfang);


                        neu.SetzeVoran(anfang.GibVoran());
                        anfang.SetzeVoran(neu);
                        if (neu.GibVoran != null)
                                neu.GibVoran().SetzeWeiter(neu);
                        anfang = neu;
                }
                if (i > anfang.GibWert())
                        anfang.SetzeWeiter
                        (Positioniere(anfang.GibWeiter(), i));
        }


        return anfang;
   }
```

Doppelt verkettete Listen

```
void ReversivDrucke() {
      DElement aktuell = fuß;
      while (aktuell != null) {

      System.out.println(aktuell.GibWert());
            aktuell = aktuell.GibVoran();
      }
  }
}
```

Exkurs: java.util - Hilfsklassen

Umfaßt Interfaces und Klassen, um unterschiedliche Datenstrukturen zu realisieren, insbesondere:

- Interfaces:
  - `Enumeration` für Objekte, die mehrere Elemente generieren
  - `Set` für Mengen ohne doppelte Elemente
- Klassen:
  - `Vector`: Mehr Flexibilität als `[]`-Arrays
  - `Stack`: Stapel für allgemeine Objekte, abgeleitet von `Vector`
  - `LinkedList:` für doppelt verkettete Listen
  - `StringTokenizer`: Einfaches Zerlegen von Strings
  - `Random`: Zufallsgenerator
    (für Kryptographie: `java.security.SecureRandom`)
  - Verschiedene Kalender und Datumsklassen

java.util.Vector

Klasse `java.util.Vector`

- `public Object[] toArray()`
  - Wandelt Vector in ein Array vom Basistyp Object
- `public int size()`
  - Gibt Anzahl der enthaltenen Komponenten zurück
- `public boolean contains(Object elem)`
  - Prüft, ob ein Objekt enthalten ist. Gleichheit wird mit `equals()` überprüft.
- `public int indexOf(Object elem)`
  - Gibt Index eines Objektes zurück. Gleichheit wird mit `equals()` überprüft.
- `public Object elementAt(int index)`
  - Gibt Element an entsprechender Position zurück
- `public void insertElementAt(Object obj, int index)`
- `public void removeElementAt(int index)`
  - Fügt Objekt an Position ein, bzw. entfernt Objekt

java.util.Vector

Zugriff auf Elemente mit einem Index (analog zu Arrays)

Abarbeiten der Liste mit einem `Enumeration` - Objekt

```
public interface Enumeration {

   // liefert true, falls noch Elemente vorhanden sind
   public abstract boolean hasMoreElements();

   // liefert das nächste Objekt der Aufzählung
   public abstract Object nextElement();
}
```

Das Interface `Enumeration`

Methode `elements()` von `Vector` liefert ein `Enumeration`-Objekt

```
Vector v = new Vector();
Person p1 = new Person(“Urs“,“Müller“,“Hauptstr.
   1“,“12345“,“Berlin“);
Person p2 = new Person(“Max“,“Muster“,“Südwall
   1“,“44332“,“Dortmund“);
v.addElement(p1);
v.addElement(p2);
```

Explizites Typcasting durch vorangestellte runde Klammern:

Person p = (Person) el.nextElement();

```
for (Enumeration el = v.elements(); el.hasMoreElements(); ) {
   Person p = (Person) el.nextElement();
   String inhaber = p.toString();
   Bildschirm.gibAus(inhaber);
}
```

Methoden der Klasse `LinkedList`

- add (int index, Object element)
- addFirst (Object element)
- addLast (Object element)
- get (int index)
- remove (int index)
- remove (Object o) löscht das erste Auftreten des Elementes o
- getFirst()
- getLast()
- size() gibt die Anzahl der Elemente zurück

Zwischenstand

- Lineare Liste als klassische, einfache dynamische DS
  - Grundkonstruktion: Objekte haben Referenz auf Objekt der eigenen Klasse
  - typische Operationen: anlegen, finden von Elementen, Einfügen von Elementen, Durchlaufen aller Elemente, Löschen eines Elementes
  - unterschiedliche Varianten
    - einfache Liste, Liste mit Kopf & Fuß Attribut, doppelt verkettete Liste
  - Sonderfälle in Operationen: 1. Element, letztes Element
  - Operationen lassen sich auch leicht rekursiv formulieren
  - Aufwand für Operationen (worst case):
    - Einfügen am Anfang: O(1)
    - Einfügen am Ende: ohne Fuß-Attribut O(N), mit Fuß-Attribut O(1)
    - Suchen eines Elementes:
      - in unsortierter Liste O(N)
      - in sortierter Liste O(N), aber Abbruch vor Listenende bei fehlendem Element, ebenso: Einfügen eines Elementes in eine sortierte Liste
- Listen sind also bzgl Suchoperationen nicht sehr effizient
- allg. Verwaltung von beliebigen Objekten verwaltet Referenzen auf Objekte

Allgemeines zu Bäumen

- Bäume sind
  - gerichtete, azyklische Graphen. Es gibt keine Zyklen zwischen Mengen von Knoten.
  - hierarchische Strukturen. Man kommt von einer Wurzel zu inneren Knoten und letztlich zu Blättern.
  - verkettete Strukturen, die dynamisch wachsen und schrumpfen können.
- Binäre Bäume sind Bäume, in denen jeder Knoten maximal zwei Söhne hat.
- Beispiele für die Anwendung binärer Bäume
  - Heapsort.
  - binäre Suchbäume.

Allgemeines zu Bäumen

- Typische Zugriffsmethoden
  - einfügen einer Wurzel
  - einfügen eines inneren Knotens
  - entfernen der Wurzel
  - entfernen eines inneren Knotens
  - suchen
  - nach links/rechts navigieren

Binäre Suchbäume

Aufgabe:

suche ein Elemente x in einer geordneten Menge

Grundidee: rekursiver Ansatz.

- beschaffe mittleres Element y der geordneten Menge
- falls x = y: fertig
- falls x < y: wende Verfahren rekursiv auf Teilmenge kleinerer Elemente an
- falls x > y: wende Verfahren rekursiv auf Teilmenge größerer Elemente an

Beobachtung:

- in jedem Schritt wird die zu betrachtende Menge halbiert
- bei N Element also $\log_2(N)$ Schritte

Binäre Suchbäume

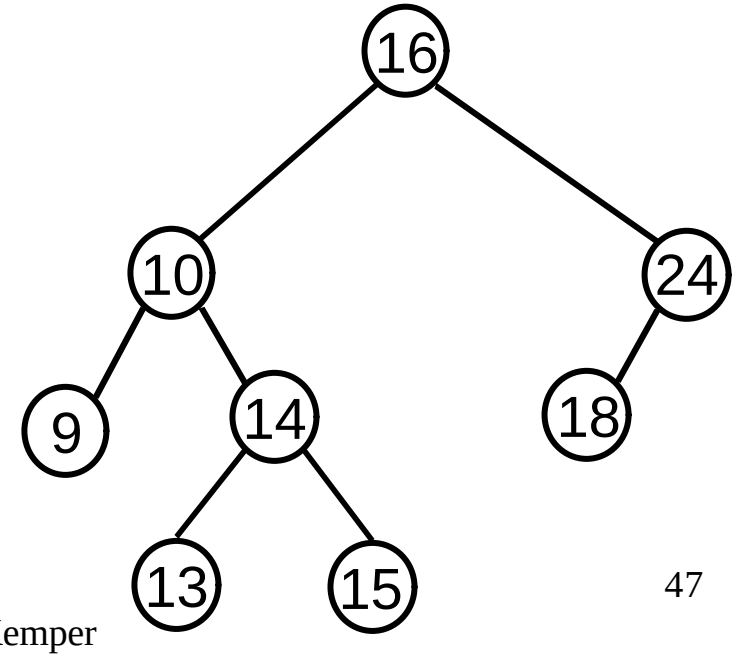
Definition:

Sei *B* ein binärer Baum, dessen Knoten mit ganzen Zahlen beschriftet sind. *B* heißt binärer Suchbaum, falls gilt:

B ist leer oder

- der linke und der rechte Unterbaum von *B* sind binäre Suchbäume,
- ist *w* die Beschriftung der Wurzel, so sind alle Elemente im linken Unterbaum kleiner als *w*, alle Elemente im rechten Unterbaum größer als *w*.

Beispiel:



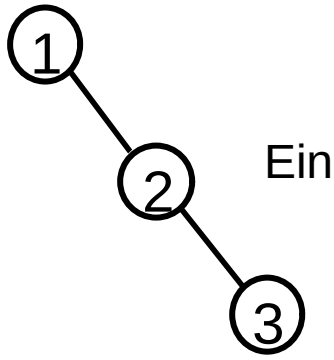
Unterstützt das Beispiel das Vorgehen der vorherigen Folie ?

Binäre Suchbäume

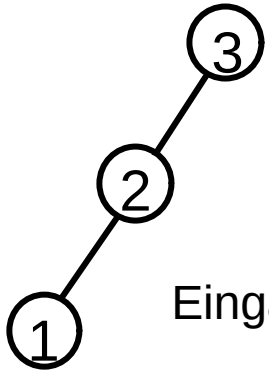
- Der Aufbau eines binären Suchbaums erfolgt durch wiederholtes Einfügen in einen leeren Baum.
- Die Reihenfolge der Werte, die in einen binären Suchbaum eingefügt werden, bestimmt die Gestalt des Baumes.
- Eine Menge von Werten kann bei unterschiedlichen Eingabereihenfolgen zu verschiedenen Repräsentationen als Baum führen.

Binäre Suchbäume

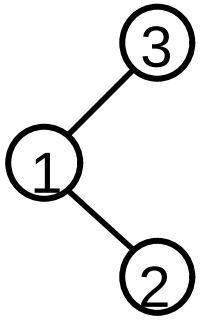
Beispiele:



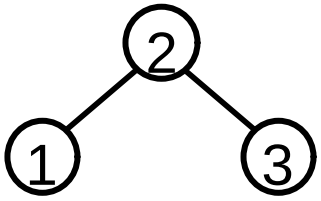
Eingabefolge 1 2 3



Eingabefolge 3 2 1



Eingabefolge 3 1 2



Eingabefolge 2 1 3 oder 2 3 1

Binäre Suchbäume - die Klasse Knoten

```
class Knoten {
  private int wert;
  private Knoten links, rechts;
  Knoten(int i) { wert = i; links = rechts = null; }
  void SetzeWert(int i) { wert = i; }
  int GibWert() { return wert; }
  void SetzeLlinks(Knoten k) { links = k; }
  Knoten GibLinks() { return links; }
  void SetzeRechts(Knoten k) { rechts = k; }
  Knoten GibRechts() { return rechts; }
};
```

Algorithmus für das Einfügen von Knoten

Gegeben seien ein binärer Suchbaum *B* und eine ganze Zahl *k*, die in *B* eingefügt werden soll. Es können vier Fälle auftreten:

- *B* ist leer: Erzeuge einen neuen Knoten, weise ihn *B* zu und setze *B.wert* auf *k*.
- *B* ist nicht leer und *B.wert* = *k*: Dann ist nichts zu tun, da keine doppelten Einträge vorgenommen werden sollen.
- *B* ist nicht leer und *B.wert* < *k*: Füge *k* in den rechten Unterbaum von *B* ein.
- *B* ist nicht leer und *B.wert* > *k*: Füge *k* in den linken Unterbaum von *B* ein.

Die Klasse Binary Search Tree (BST)

```
class BST {
   private Knoten wurzel;
   BST() { wurzel = null; }
   void FügeEin(int i) {
       wurzel = FügeEin(wurzel, i);
   }
```

Die Klasse Binary Search Tree (BST) - Einfügen

```
private Knoten FügeEin(Knoten aktuell, int ein) {
        if (aktuell == null)
                aktuell = new Knoten(ein);
        else {
                if (ein < aktuell.GibWert())
                        aktuell.SetzeLinks
                        (FügeEin(aktuell.GibLinks(), ein));
                if (ein > aktuell.GibWert())
                        aktuell.SetzeRechts
                        (FügeEin(aktuell.GibRechts(), ein));
        }
        return aktuell;
   }
}
```

Algorithmus für die Suche von Konten

Der am Beginn dieses Kapitels skizzierte Algorithmus für das binäre Suchen lässt sich nun mit der durch die Methode **`FügeEin`** aufgebauten Datenstruktur recht einfach realisieren.

Gegeben sind ein binärer Suchbaum *B* und eine Zahl *k*,
die in dem Baum *B* gesucht werden soll:

- ***B* ist leer: *k* kann nicht im Baum sein.**
- ***B* ist nicht leer, so betrachtet man die Fälle**
  - ***B.wert* = *k*:** ***k* ist gefunden, d.h. bereits in dem Baum *B* vorhanden.**
  - ***B.wert* < *k*:** **Suche im rechten Unterbaum von *B*.**
  - ***B.wert* > *k*:** **Suche im linken Unterbaum von *B*.**

Die Klasse Binary Search Tree (BST) - Suchen

```
class BST {
   ...
   boolean Suche(int i) {
       return Suche(wurzel, i);
   }
   private boolean Suche(Knoten aktuell, int i) {
       boolean gefunden = false;
       if (aktuell != null) {
             gefunden = (aktuell.GibWert() == i) ;
             if (aktuell.GibWert() < i)
                 gefunden =
   Suche(aktuell.GibRechts(), i);
             if (aktuell.GibWert() > i)
                 gefunden = Suche(aktuell.GibLinks(),
   i);
       }
       return gefunden;
   }
   ...
}
```

Suchen in binären Suchbäumen

Definition:

- Ist B ein binärer Baum, so definiert man die Höhe $h(B)$ von B rekursiv durch:

- $$h(B) := \begin{cases} 0, & \text{falls B leer ist} \\ 1 + max\{h(B_1), h(B_2)\}, & \text{falls } B_1 \text{ und } B_2 \text{ linker bzw. rechter Unterbaum von B sind} \end{cases}$$

Ist B ein binärer Suchbaum mit $h(B)$=n, so enthält B mindestens n und höchstens 2^n-1 Knoten.

- n, wenn der Baum zur Liste degeneriert ist,
- 2^n-1, wenn jeder von 2^{n-1}-1 inneren Knoten genau zwei Söhne und jedes von 2^{n-1} Blättern keine Söhne hat.

Schema der vollständigen Induktion über Bäumen

In rekursiven Datenstrukturen werden Aussagen häufig mittels vollständiger Induktion bewiesen.

Schema:

- Spezifikation der Behauptung S(T), wobei T ein Baum ist.

Beweis

- Induktionsanfang: zeige, dass S(T) für Bäume mit einem Knoten gilt.
- Induktionsannahme: T ist ein Baum mit Wurzel w und k ≥1 Unterbäume, T_1, ..., T_k. Annahme, dass $S(T_i)$ für i ∈ 1,2, ..., k gilt.
- Induktionsschritt: zeige, dass S(T) unter der Induktionsannahme gilt.

Suchen in binären Suchbäumen

Behauptung: In einem beliebigen binären Suchbaum B, braucht man für eine erfolglose Suche maximal h(B) Vergleiche

Beweis: durch vollständige Induktion nach dem Aufbau von B.

Induktionsanfang: Ist B leer, also h(B)=0, so ist kein Vergleich nötig.

Induktionsschritt: Wenn für zwei Bäume B1 und B2 mit der Höhe n gilt, dass in jedem eine erfolglose Suche in maximal n Vergleichen möglich ist, so gilt für den binären Suchbaum mit einer neuen Wurzel und deren Unterbäume B1, B2, dass maximal n+1 Vergleiche nötig sind.

Beweis des Induktionsschrittes: Durch Wurzelvergleich (ein Vergleich) wird ermittelt, ob im linken oder rechten Teilbaum weitergesucht werden muss. Die maximale Zahl der Vergleiche in einem Teilbaum ist n. Also, braucht man insgesamt maximal n+1 Vergleiche. In einem Baum der Höhe n+1 braucht man also maximal n+1 Vergleiche.

Suchen in binären Suchbäumen

Daraus ergibt sich:

Bei einer erfolglosen Suche in einem binären Suchbaum mit n Elementen sind mindestens *log n* (Basis 2) und höchstens *n* Vergleiche notwendig.

Der günstige Fall (log n Vergleiche) gilt in einem gleichgewichtigen Baum. Der ungünstige (n Vergleiche) gilt in einem vollständig degenerierten Baum, der beispielsweise immer dann entsteht, wenn die Elemente in sortierter Reihenfolge eintreffen.

Um diese Unsicherheit auszuräumen (und somit eine Laufzeit auf der Basis von log n Vergleichen sicherzustellen), werden balancierte, binäre Suchbäume benutzt.

Suchen in binären Suchbäumen

Eine Art balancierter, binärer Suchbäume sind die AVL-Bäume (nach ihren Erfindern Adelson, Velskii, Landis).

Def.: Ein AVL-Baum ist ein binärer Suchbaum, in dem sich für jeden Knoten die Höhen seiner zwei Teilbäume um höchstens 1 unterscheiden.

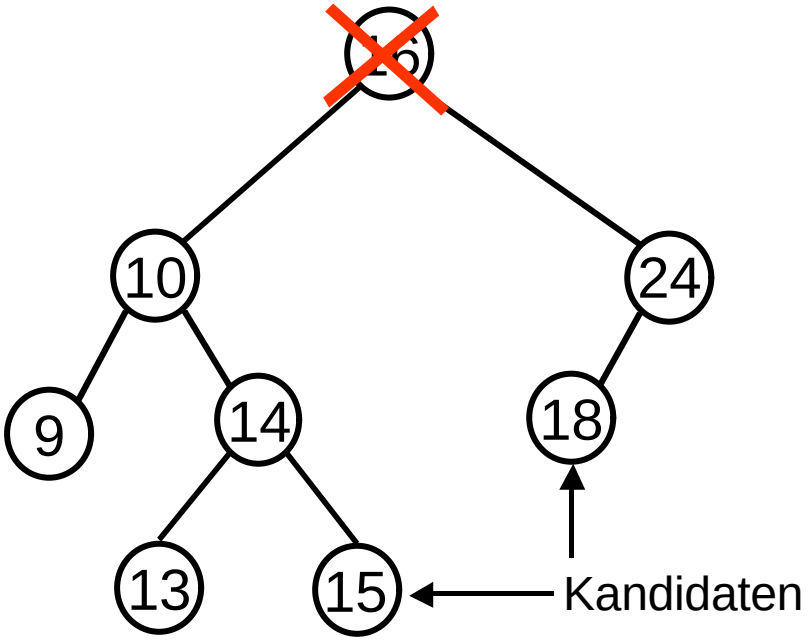
Einfüge- und Entferne-Operationen werden dann etwas aufwendiger, aber dafür ist die Suche auch in ungünstigen Fällen effizienter (vgl Literatur).

Beispiel: Entfernen der Wurzel aus einem binären Suchbaum

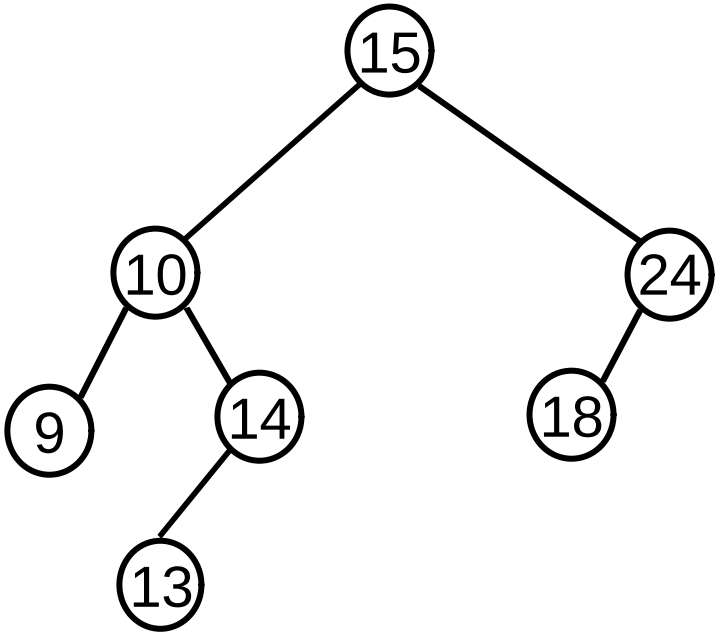
Algorithmus für das Entfernen

- Entfernen der Wurzel führt zur Konstruktion eines neuen binären Suchbaums.
- Darum: Finden eines Knotens, der an die Stelle der Wurzel gesetzt wird und die Kriterien für einen neuen binären Suchbaum erfüllt.
- Der Knoten muss größer als die Wurzel des linken Unterbaumes sein und kleiner als die Wurzel des rechten Unterbaumes.

Entfernen der Wurzel - Beispiel



Situation vor dem Löschen



Situation nach dem Löschen

Entfernen der Wurzel aus einem binären Suchbaum

Algorithmus für das Entfernen

- Es wird der Knoten mit der größten Beschriftung im linken Unterbaum genommen.
- Dieser Knoten wird entfernt und als Wurzel eingesetzt.
- Ist der linke Unterbaum einer Wurzel leer, nimmt man analog zur vorgestellten Methode das kleinste Element der rechten Wurzel.
- Ist der Unterbaum einer Wurzel leer, kann auch auf eine Umgestaltung des Baumes verzichtet werden: Wird die Wurzel entfernt, bildet der verbleibende Unterbaum wieder einen binären Baum.
- Wird ein innerer Knoten aus einem binären Suchbaum entfernt, stellt dieser Knoten die Wurzel eines Unterbaumes dar und diese Wurzel wird dann entfernt.

Durchlaufstrategien für binäre Suchbäume

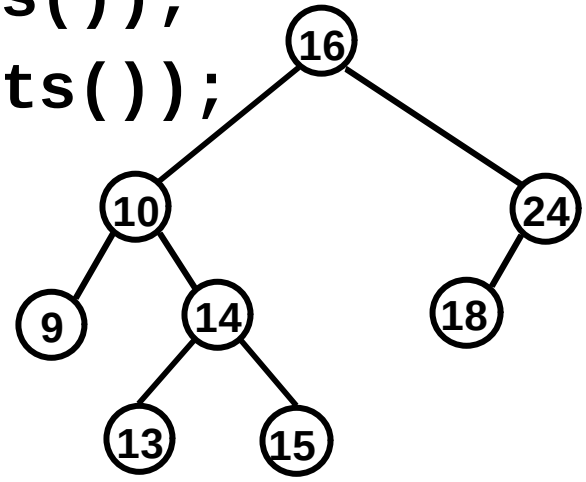
Tiefendurchlauf: Hier wird von einem Knoten aus in die Tiefe gegangen, indem einer der Söhne besucht wird und dann dessen Söhne usw. Erst wenn man die Blätter erreicht hat, beginnt der Wiederaufstieg.

- Preorder-Durchlauf
- Inorder-Durchlauf
- Postorder-Durchlauf

Breitendurchlauf: Mit dem Besuch eines Knotens werden auch seine Nachbarn besucht.
„Schichtweises Abtragen“

Tiefendurchlauf / Preorder

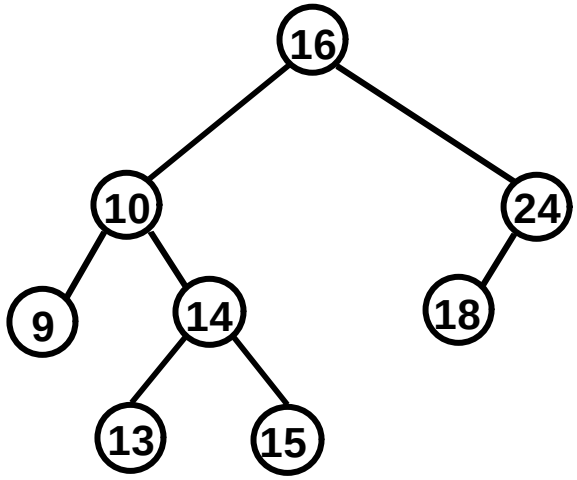
```
void PreOrder() {PreOrder(wurzel); }
   private void PreOrder(Knoten aktuell) {
     if (aktuell != null) {
      System.out.println(aktuell.GibWert());
      PreOrder(aktuell.GibLinks());
      PreOrder(aktuell.GibRechts());
     }
    }
```



Reihenfolge der besuchten Knoten: 16, 10, 9, 14, 13, 15, 24, 18

Tiefendurchlauf / Inorder

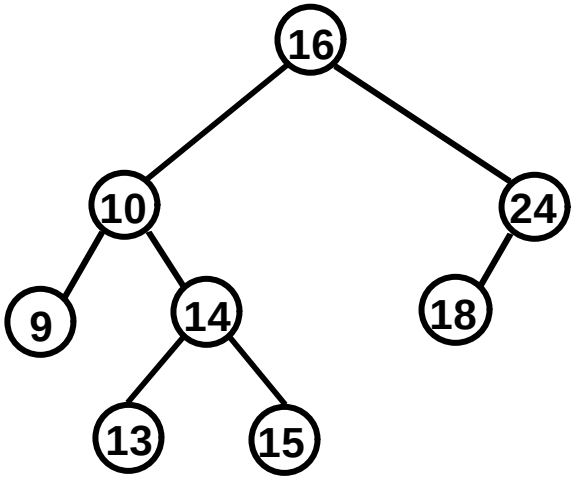
```
void InOrder() { InOrder(wurzel); }
private void InOrder(Knoten aktuell) {
  if (aktuell != null) {
     InOrder(aktuell.GibLinks());
     System.out.println(aktuell.GibWert());
     InOrder(aktuell.GibRechts());
  }
}
```



Reihenfolge der besuchten Knoten: 9, 10, 13, 14, 15, 16, 18, 24

Tiefendurchlauf / Postorder

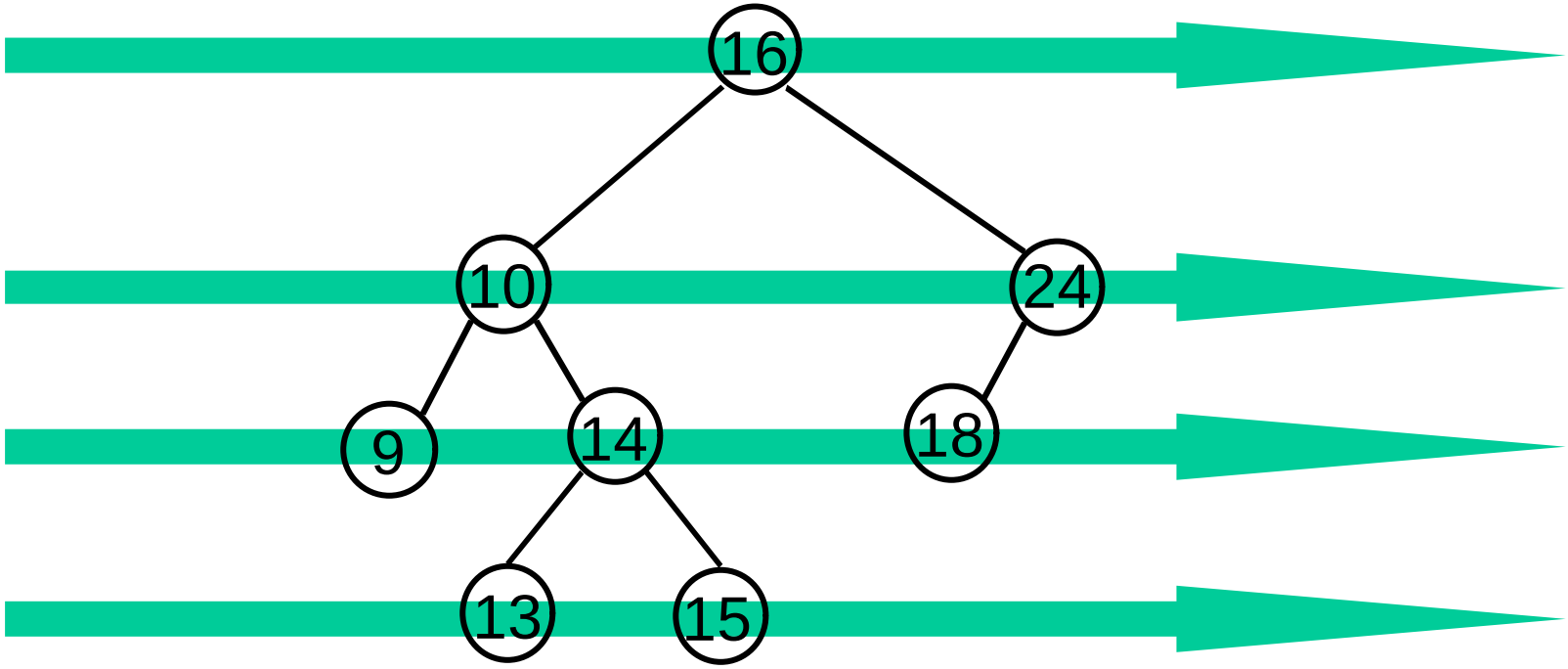
```
void PostOrder() { PostOrder(wurzel); }
private void PostOrder(Knoten aktuell) {
  if (aktuell != null) {
     PostOrder(aktuell.GibLinks());
     PostOrder(aktuell.GibRechts());
     System.out.println(aktuell.GibWert());
  }
}
```



Reihenfolge der besuchten Knoten:

9, 13, 15, 14, 10, 18, 24, 16

Breitendurchlauf



Reihenfolge der besuchten Knoten: 16, 10, 24, 9, 14, 18, 13, 15

Idee zur Realisierung des Breitendurchlaufs

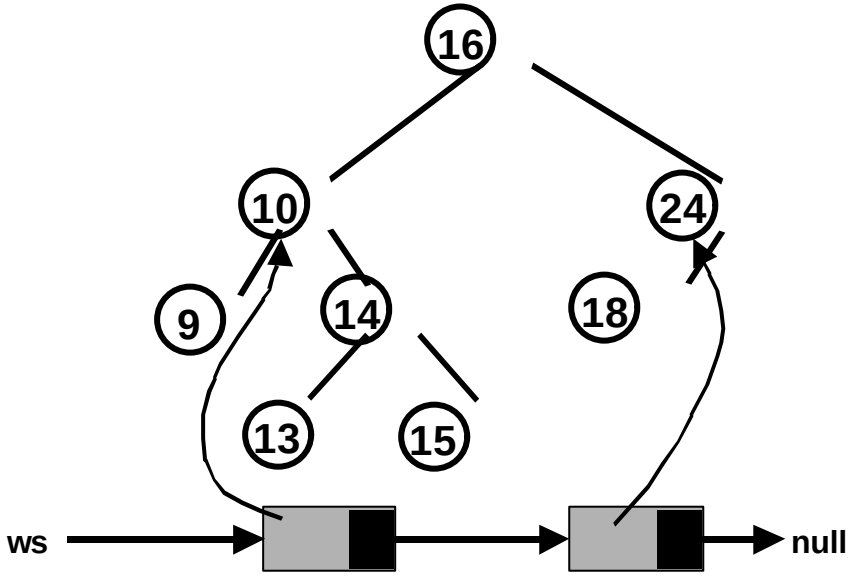
- Noch nicht besuchte Knoten in verketteter Liste zwischenspeichern
- nächster Knoten steht am Listenanfang.
- Knoten wird besucht,
  - Knoten aus der Liste entfernen
  - linker und rechter Sohn (falls vorhanden), in dieser Reihenfolge ans Ende der Liste angefügen
- Dies geschieht solange, bis die Liste leer ist.
- Die Liste wird mit der Wurzel des Baumes initialisiert.

Liste beschreibt eine *Warteschlange* für Knoten:

- Der Knoten am Anfang der Warteschlange wird als nächster ausgedruckt.

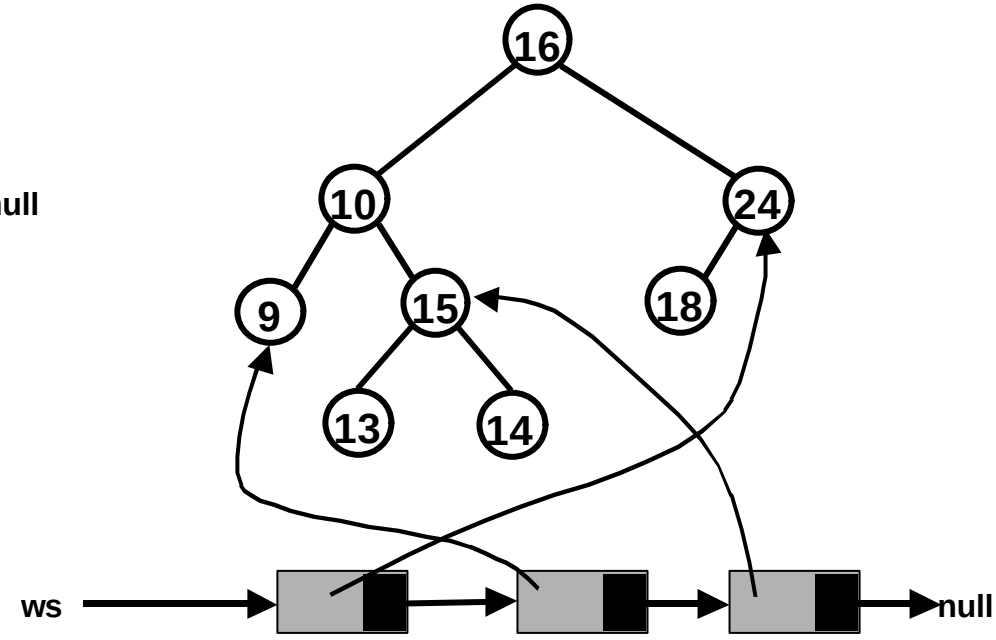
- Der Knoten am Ende der Warteschlange ist als letzter hinzugefügt worden.

Warteschlange für die Breitensuche



nach Besuch des Knotens 16, vor Besuch des Knotens 10

nach Besuch des Knotens 10



Breitendurchlauf durch einen binären Suchbaum

```
class BST {
   ...
   void Breitendurchlauf() {
        SohnListe ws = new SohnListe();
        Knoten aktuell;
        ws.FügeAn(wurzel);
        while ((aktuell = ws.Entferne()) != null) {
              System.out.println(aktuell.GibWert());
              if (aktuell.GibLinks() != null)
                   ws.FügeAn(aktuell.GibLinks());
              if (aktuell.GibRechts() != null)
                   ws.FügeAn(aktuell.GibRechts());
        }
   }
   ...
}
```

Die Klasse SohnElem

```
class SohnElem {
  private Knoten wert;
  private SohnElem weiter;
  SohnElem(Knoten k) { wert = k; weiter = null; }
  void SetzeWert(Knoten k) { wert = k; }
  Knoten GibWert() { return wert; }
  void SetzeWeiter(SohnElem s) { weiter = s; }
  SohnElem GibWeiter() { return weiter; }
}
```

Die Klasse SohnListe

```
class SohnListe {
  private SohnElem kopf, fuß;
  SohnListe() { kopf = fuß = null; }
  void FügeAn(Knoten an) {
     SohnElem neu = new SohnElem(an);
     if (fuß != null) {
          fuß.SetzeWeiter(neu);
          fuß = neu;
     } else
          kopf = fuß = neu;
  }
```

Die Klasse SohnListe

```
Knoten Entferne() {
      Knoten erster = null;
      if (kopf != null) {
            erster = kopf.GibWert();
            kopf = kopf.GibWeiter();
            if (kopf == null)
                  fuß = null;
      }
      return erster;
  }
}
```

Zusammenfassung

- Listen: ungünstig bzgl Suchaufwand O(N)
- Binäre Suchbäume
  - gerichtete, azyklische Graphen mit max 2 Nachfolgern je Knoten und max 1 Vorgänger je Knoten
  - Höhe des Baumes = max Länge einer Suche
    - degenerierter Baum: Suche in O(N)
    - balancierter Baum: Suche in $O(\log_2(N))$
  - Viele Varianten von Bäumen, um Suchaufwand und Aufwand für Einfüge/Entferne Operationen gering zu halten
    - AVL Bäume, 2-3 Bäume, 2-3-4 Bäume, Rot-Schwarz Bäume
  - Operationen auf Bäumen
    - Einfügen
    - Löschen
    - Suchen
    - Traversieren: Inorder/Preorder/Postorder, Breitendurchlauf
- im weiteren betrachten wir: Graphen