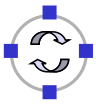
Programmierung und Algorithmen

Kapitel 10
Grundlegende Datenstrukturen



Überblick

Einführung Datenstrukturen

Kollektionen:
- Motivation
- Stapel (Stack)
- Listen
- Kollektionen in Java

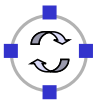
Parametrisierbare Datenstrukturen in Java

Anhang:
- Bäume
- Halden (Heap)
- Graphen



Datenstruktur - Begriff

Eine Datenstruktur ist eine **bestimmte Art, Daten zu verwalten** und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten **Operationen** verknüpft, um diesen Zugriff und diese Manipulation zu ermöglichen.



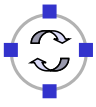
Datenstruktur vs. ADT

Abstrakter Datentyp
Spezifikation: Festlegung/Beschreibung von Eigenschaften und Verhalten



Implementierung

Datenstruktur
(Wieder-) verwendbare, ausführbare Umsetzung komplexer Datentypen



Datenstrukturen - Beispiele

Grundlegende Datenstrukturen:

- **Stack, Liste**, Warteschlange, …
- Zahlreiche mögliche Anwendungen
- Oft Grundlage für speziellere Datenstrukturen

Indexstrukturen:

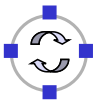
- **Bäume und Hash-Tabellen**
- Effizienter Zugriff auf große Datenmengen

Komplexe Netzwerke:

- Grundlage: **Graphen**
- Verschiedene interne Darstellungsformen

Anwendungsspezifische Datenstrukturen:

- Elektrische Schaltungen, Genom, Multimedia,. . .



Zielsetzungen für Datenstrukturen

Implementierung komplexer Datentypen

- Abbildung mit Hilfe einer Programmiersprache
- Wiederverwendbarkeit
- Speichereffizienz

Implementierung der Operationen

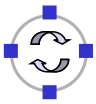
- Algorithmen aus Spezifikation
- Laufzeiteffizienz



Wiederverwendbarkeit

- **Meist: (Wieder-)verwendung von Datenstrukturen**
  - Nutzung existierender Implementierung in Form von Klassen, Klassenbibliotheken, Packages
  - Gegebenenfalls Erweiterung durch Vererbung oder Einbettung

- **Aber auch: eigenständige Implementierung**
  - Insbesondere für anwendungsspezifische Datenstrukturen
  - Ausnutzung der Anwendungssemantik führt meist zu effizienteren Lösungen



Motivation: Kollektionen

- Klassen zur Verwaltung einer Mehrzahl gleich strukturierter Objekte
- Ausprägungen: Felder, Listen, Stacks, Sets, . . .
- Beispiel: Realisierung als Feldtypen

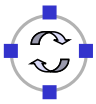
```
int[] feld
```

- Probleme:
  - Nutzung mit verschiedenen Elementtypen: Feld von **`int`**-Werten, von Strings, von rationalen Zahlen, . . .
  - Umständliche Handhabung: Anfügen, Einfügen an Position 0, . . .



Realisierungsvarianten

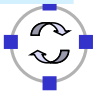
- **Gemeinsame Wurzelklasse `Object`:**
  - Elementtyp = Wurzelklasse
  - Problem: Typunsicherheit
  - Beispiel: Java, Smalltalk
- **Schablonen oder Templates**
  - Instantiierung mit konkreten Elementtyp
  - Typsicher
  - Generierung spezifischen Codes durch Compiler
  - Beispiel: C++, Java 5.0 (Generics)



Beispiel: Feld von Objekten

- Implementierung auf Basis des Java-Array-Typs

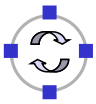
```
public class ObjFeld {
  private Object daten[] = null;
  public ObjFeld() {}
  public int laenge() {
    return (daten == null ? 0 : daten.length);
  }
  public Object element (int idx) {
    if (daten == null || idx < 0 ||
          idx >= laenge())
        throw new ArrayIndexOutOfBoundsException();
    else return daten[idx];
  } ...
}
```



Beispiel: Feld von Objekten /2

- Einfügen von Elementen

```
public void einfuegen (int pos, Object o) {
    int i, num = laenge();
    if (pos == -1) pos = num;
    Object ndaten[] = new Object[num + 1];
    for (i = 0; i < pos; i++)
        ndaten[i] = daten[i];
    ndaten[i++] = o;
    for (; i < num + 1; i++)
        ndaten[i] = daten[i - 1];
    daten = ndaten;
}
```



Feld von rationalen Zahlen

- Einfügen von Objekten der Klasse `RatNumber`

```
ObjFeld feld = new ObjFeld();
feld.einfuegen(-1, new RatNumber(1, 3));
feld.einfuegen(-1, new RatNumber(2, 3));
```

- Zugriff auf Elemente erfordert explizite Typkonvertierung (type cast)

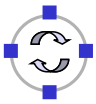
```
RatNumber rn = (RatNumber)feld.element(0);
```



Feld von rationalen Zahlen /2

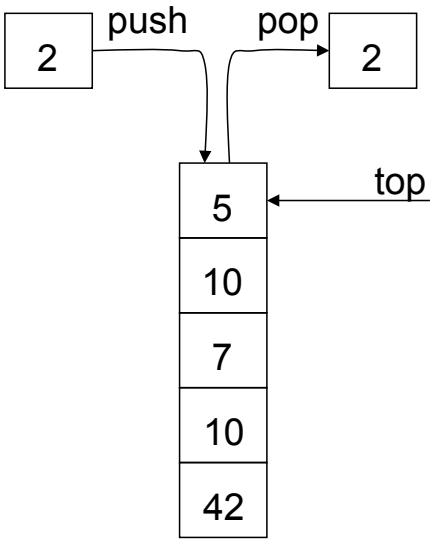
- Test auf Zugehörigkeit zu einer Klasse

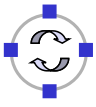
```
Object o = feld.element(0);
if (o instanceof RatNumber) {
    RatNumber rn = (RatNumber) o;
    ...
}
```



Kellerspeicher (Stack): Prinzip

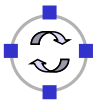
- LIFO-Prinzip: Last-In-First-Out-Speicher





Stack als Datentyp

- **`void`** `push(Object obj)` legt das Objekt *obj* als oberstes Element auf dem Stack ab
- `Object pop()` nimmt das oberste Element vom Stack und gibt es zurück
- `Object top()` gibt das oberste Element des Stacks zurück, ohne es zu entfernen
- **`boolean`** `isEmpty()` liefert **`true`**, wenn keine Elemente auf dem Stack liegen, andernfalls **`false`**



Stack als Datentyp /2

```
public class Stack {
    public void push(Object obj)
          throws StackException { ...}
    public Object pop()
          throws StackException { ...}
    public Object top()
          throws StackException { ...}
    public boolean isEmpty() { ...}
}
```



Nutzung des Stack

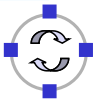
```
Stack stack = new Stack();
try {
      // Elemente auf den Stack ablegen
      stack.push("Eins");
      stack.push("Zwei");
      stack.push("Drei");
      // Elemente von dem Stack lesen und entfernen
      while (! stack.isEmpty()) {
          String s = (String) stack.pop();
          System.out.println(s);
      }
} catch (StackException exc) {
          System.out.println("Fehler: " + exc);
}
```



Implementierung des Stack

- Verschiedene Implementierungen möglich
- Hier über statisches Feld mit vorgegebener Kapazität
- Notwendig: Behandlung des Überlaufs (`StackException`)

```
public class Stack {
  Object elements[] = null; // Elemente
  int num = 0; // aktuelle Anzahl
  // Stack mit vorgegebener Kapazität
  public Stack(int size) {
    elements = new Object[size];
  }
...
}
```



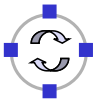
Implementierung des Stack /2

```
public void push(Object obj)
            throws StackException {
  if (num == elements.length)
     // Kapazität erschöpft
     throw new StackException();
  elements[num++] = obj;
}
public Object pop() throws StackException {
  if (isEmpty ()) // Stack ist leer
                  throw new StackException();
  Object o = elements[--num];
  elements[num] = null;
  return o;
}
```



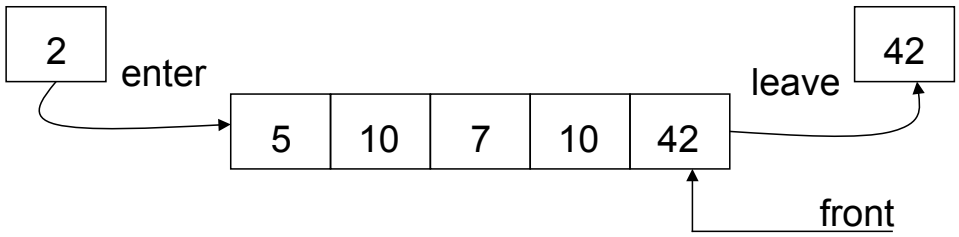
Implementierung des Stack /3

```
public Object top() throws StackException {
    if (isEmpty())
          throw new StackException();
    return elements[num - 1];
}
public boolean isEmpty() {
    return num == 0;
}
```

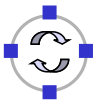


Warteschlange (Queue)

FIFO-Prinzip: „First in, first out.“



- Kann auf unterschiedliche Weisen implementiert werden:
  - Array-Implementierung mit zusätzlicher Speicherung der Positionen des ersten und letzten Elements (jeweils modulo der Array-Größe)
  - Implementierung über verkettete Liste

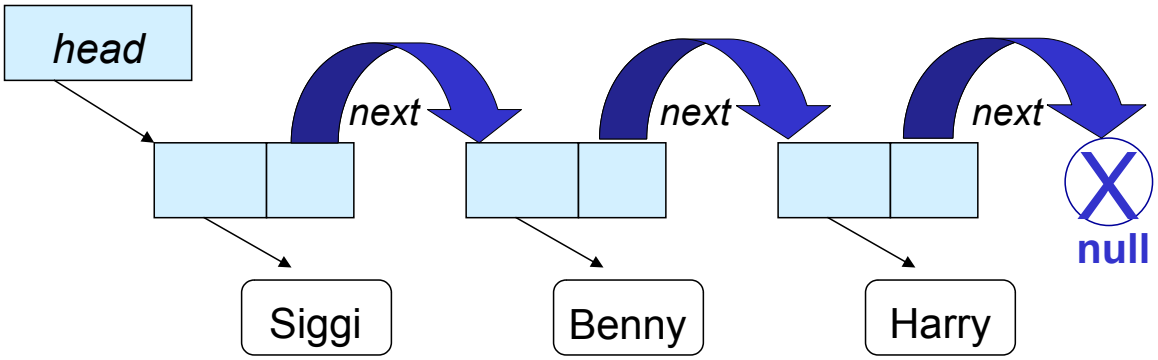


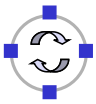
Verkettete Liste

- Bisherige Datenstrukturen: statisch
  - Können zur Laufzeit nicht wachsen bzw. schrumpfen
  - Dadurch keine Anpassung an tatsächlichen Speicherbedarf
- Ausweg: dynamische Datenstrukturen
- Beispiel: **verkettete Liste**
  - Menge von Knoten, die untereinander „verzeigert“ sind
  - Jeder Knoten besitzt Verweis auf Nachfolgerknoten sowie das zu speichernde Element
  - Listenkopf: spezieller Knoten `head`
  - Listenende: **`null`**-Zeiger



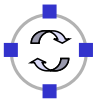
Verkettete Liste: Prinzip





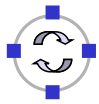
Verkettete Liste: Schnittstelle

- **`void`** `addFirst(Object obj)` fügt das Objekt `obj` als erstes Element in die Liste ein
- `Object getFirst()` liefert das erste Element der Liste
- `Object removeFirst()` entfernt das erste Element der Liste und gibt es gleichzeitig als Ergebnis zurück
- **`void`** `addLast(Object obj)` hängt das Objekt `obj` als letztes Element an die Liste an
- `Object getLast()` liefert das letzte Element der Liste
- `Object removeLast()` entfernt das letzte Element der Liste und gibt es gleichzeitig als Ergebnis zurück

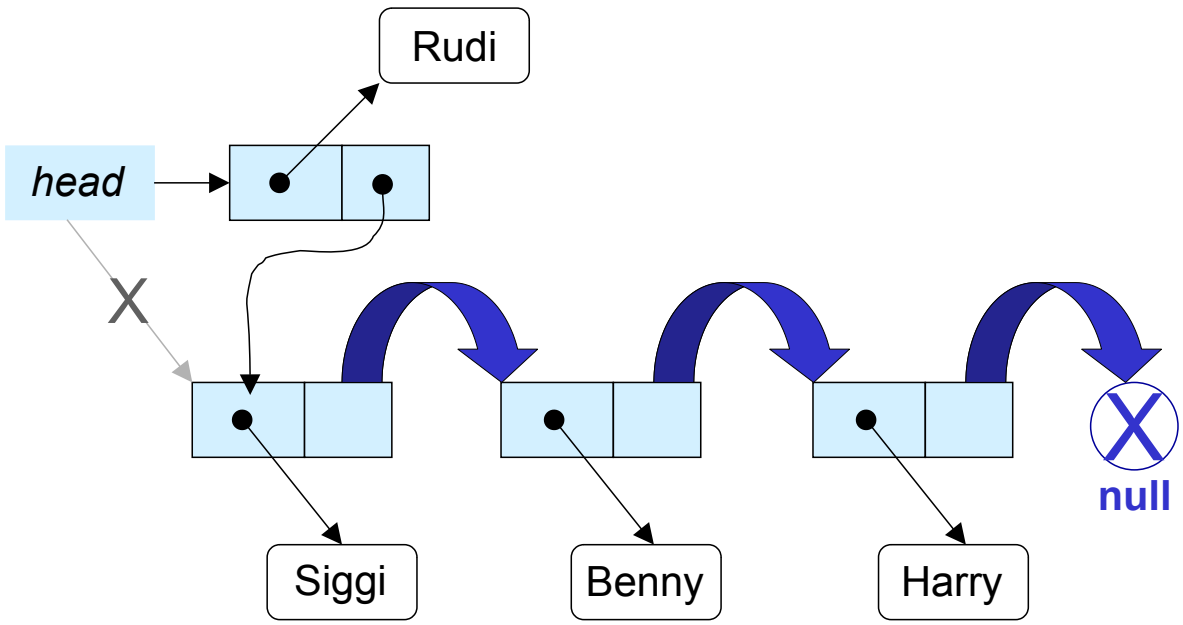


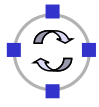
Verkettete Liste: Einfügen

1. Einfügeposition suchen
2. Neuen Knoten anlegen
3. Zeiger vom neuen Knoten auf Knoten an Einfügeposition
4. Zeiger vom Vorgänger auf neuen Knoten

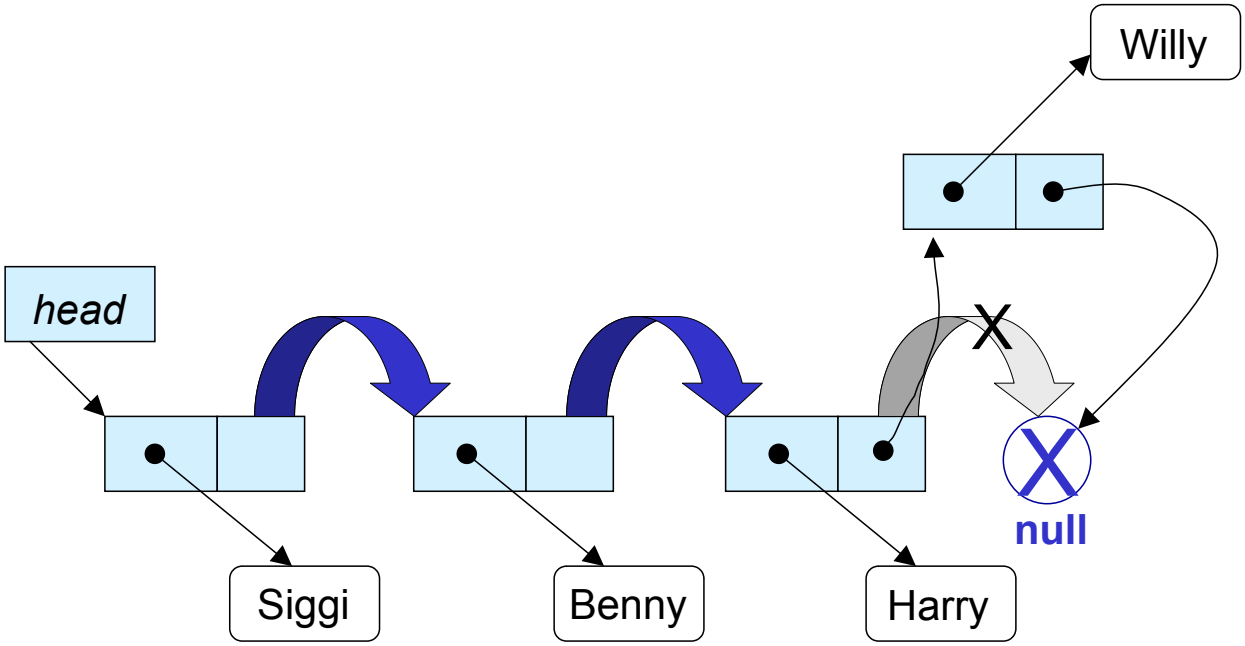


Verkettete Liste: Einfügen /2

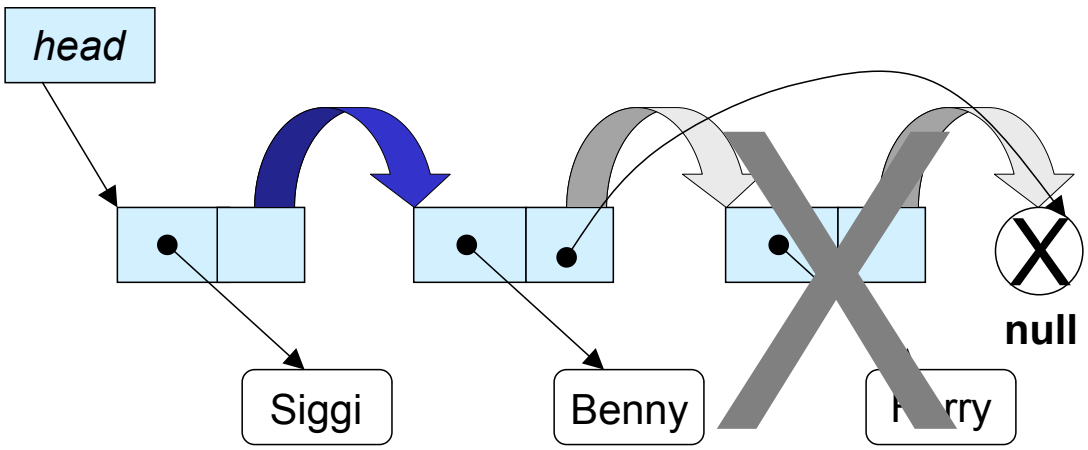




Verkettete Liste: Einfügen /3



Verkettete Liste: Löschen

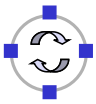


Komplexität der Operationen

Operation	Komplexität

`addFirst`
`getFirst`
`removeFirst`
	$O(1)$

`addLast`
`getLast`
`removeLast`
	$O(n)$



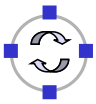
Implementierung: Knotenklasse

```
class Node {
  Object obj; Node next;
  public Node (Object o, Node n) {
    obj = o; next = n;
  }
  public Node() {
    obj = null; next = null;
  }
  public void setElement(Object o) { obj = o; }
  public Object getElement() { return obj; }
  public void setNext(Node n) { next = n;
  public Node getNext() { return next; }
}
```



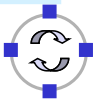
Implementierung: Initialisierung

```
public class List {
    class Node { ... }

    private Node head = null;

    public List() {
      head = new Node();
    }

    ...
}
```



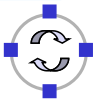
Implementierung: Einfügen

```
public void addFirst(Object o) {
   // neuen Knoten hinter head einführen
   Node n = new Node(o, head.getNext());
   head.setNext(n);
}
public void addLast(Object o) {
   Node l = head;
   // letzten Knoten ermitteln
   while (l.getNext() != null)
         l = l.getNext();
   Node n = new Node(o, null);
   // neuen Knoten anfügen
   l.setNext(n);
}
```



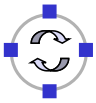
Implementierung: Löschen

```
public Object removeFirst() {
   if (isEmpty()) return null;
   Object o = head.getNext().getElement();
   head.setNext(head.getNext().getNext());
   return o;
}
public Object removeLast() {
   if (isEmpty()) return null;
   Node l = head;
   while (l.getNext().getNext() != null)
         l = l.getNext();
   Object o = l.getNext().getElement();
   l.setNext(null);
   return o;
}
```



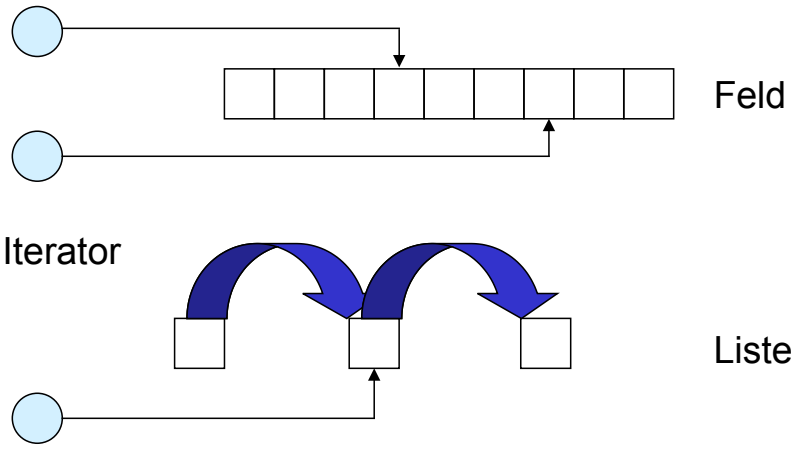
Liste: Nutzung

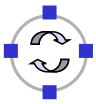
```
List liste = new List();
liste.addFirst("Drei");
liste.addFirst("Zwei");
liste.addFirst("Eins");
liste.addLast("Vier");
liste.addLast("Fünf");

while (! liste.isEmpty())
   System.out.println((String)
                      liste.removeFirst());
```



Navigation: Iterator-Konzept

- „Durchwandern“ von Kollektionen?
- Prinzipiell implementierungsabhängig: siehe Feld, Stack, Liste, ...
- Einheitliche Behandlung durch Iterator
  - Objekt als abstrakter Zeiger in Kollektion
  - Methoden zum Iterieren über Kollektion





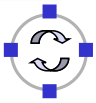
Iterator-Schnittstelle `java.util.Iterator`

- **`boolean`** `hasNext()` prüft, ob noch weitere Elemente in der Kollektion verfügbar sind
- `Object next()` liefert das aktuelle Element zurück und setzt den internen Zeiger des Iterators auf das nächste Element



Iterator: Implementierung für `List`

```
class List {
   class ListIterator implements Iterator {
      private Node node = null;
      public ListIterator() { node = head.getNext(); }
      public boolean hasNext() {
         return node != null; }
      public Object next() {
         if (! hasNext())
            throw new NoSuchElementException();
         Object o = node.getElement();
         node = node.getNext();
         return o; }
   }
}
```



Iterator: Nutzung

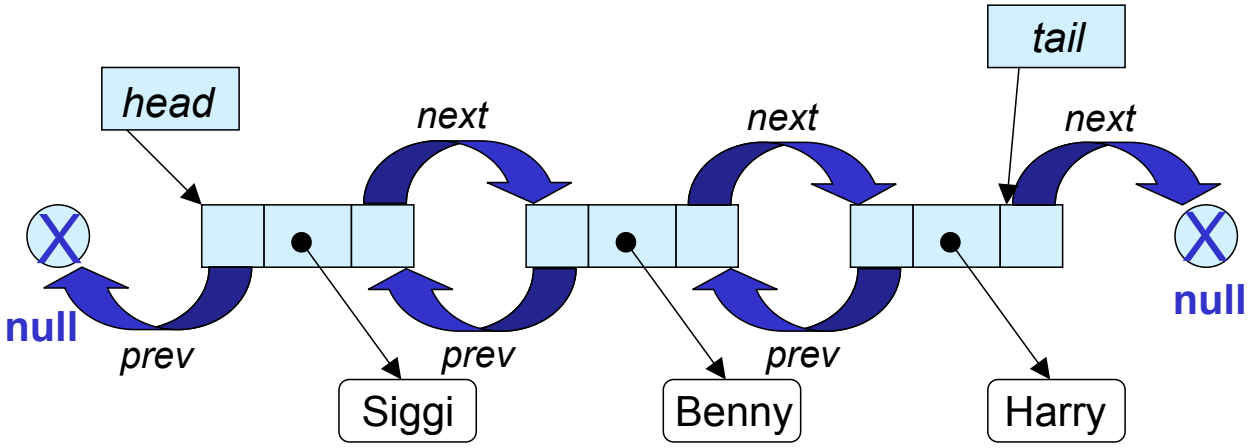
```
class List {
 ...
 public Iterator iterator() {
   return new ListIterator();
 }
}
```

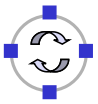
```
java.util.Iterator iter = liste.iterator();
while (iter.hasNext()) {
   Object obj = iter.next();
   // Verarbeite obj ...
}
```



Doppelt verkettete Listen

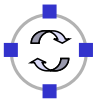
- Einfache Verkettung (nur „Vorwärtszeiger“) erlaubt nur Vorwärtsnavigation
- Rückwärtsnavigation durch zusätzlichen „Rückwärtszeiger“ ⇝ doppelt verkettet





Java-Kollektionen

- Java Collection Framework
- Bereitstellung wichtiger Kollektionen (Listen, Mengen, Verzeichnisse) mit unterschiedlicher Implementierung
- Navigation mittels Iteratoren
- Implementierung häufig benutzter Algorithmen (Suchen, Sortieren, kleines/größtes Element, . . .)
- Basis `java.util.Collection`



Schnittstelle für alle Kollektionsklassen

- Schnittstelle java.util.Collection
- Größe der Kollektion

```
int size()
boolean isEmpty()
```

- Suchen von Elementen

```
boolean contains(Object o)
boolean containsAll(Collection c)
```

- Navigation

```
Iterator iterator()
```



Schnittstelle /2

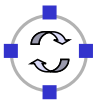
- Hinzufügen/Entfernen

```
boolean add(Object o)
boolean remove(Object o)
boolean addAll(Collection c)
boolean removeAll(Collection c)
```



Ausgewählte Kollektionen

- `java.util.List`
  Repräsentation von Listen (geordnet nach Einfügereihenfolge, Duplikate zulässig)
- `java.util.Set`
  Repräsentation von Mengen (ungeordnet/geordnet, ohne Duplikate)
- `java.util.Map`
  Verzeichnis (Dictionary) von Objekten (Schlüssel-Wert-Paare) für schnellen Zugriff über Schlüssel



java.util.List: Schnittstelle für Listen

- Zugriff über Position

```
Object get(int idx)
Object set(int idx, Object o)
void add(int idx, Object o)
Object remove(int idx)
```

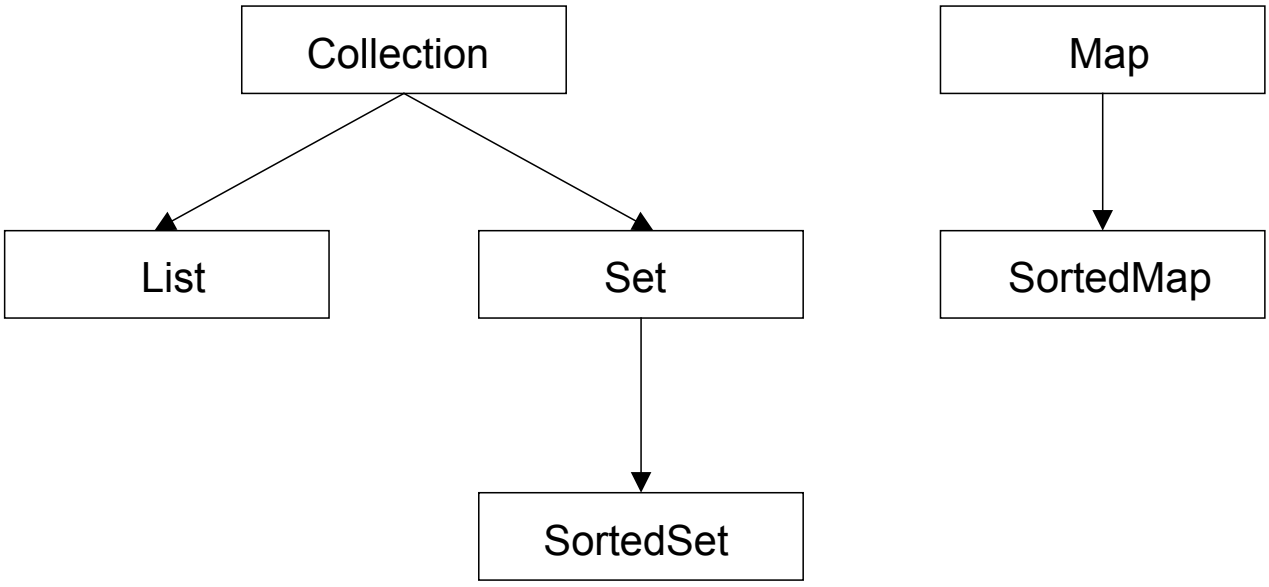
- Bestimmen der Position eines Objektes

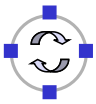
```
int indexOf(Object o)
```

- Konkrete Implementierungen
- Verkettete Liste: `LinkedList`
- Feld: `ArrayList`



Struktur des Java Collection Framework





Java Collection Framework: Implementierungen

Implementierung	Schnittstelle		
	`Set`	`List`	`Map`
Hashtabelle	`HashSet`		`HashMap`
Feld		`ArrayList`	
Baum	`TreeSet`		`TreeMap`
Liste		`LinkedList`	



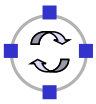
Algorithmen für Kollektionen

- Klasse java.util.Collections
- Sortieren von Listen

```
static void sort(List list)
```

- Suchen in Listen

```
static void binarySearch(List l, Object k)
```



Algorithmen für Kollektionen /2

- Umkehren

```
static void reverse(List list)
```

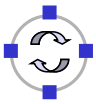
- Kleinstes/größtes Element

```
static Object min(Collection col)
static Object max(Collection col)
```



Anwendungsbeispiel

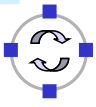
```
// Liste erzeugen und Objekte einfügen
LinkedList list = new LinkedList();
list.add(new RatNumber(1, 3));
list.add(new RatNumber(1, 4));
list.add(new RatNumber(1, 5));
// Sortieren
Collections.sort(list);
// Ausgeben
Iterator i = list.iterator();
while (i.hasNext())
    System.out.println(i.next());
```



Generics in Java

- Seit Version J2SE 5.0: direkte Unterstützung von parametrisierten Datentypen ⇝ Generics
- Notation: `Klasse<Elementtyp>`
- Speziell für Kollektionen und Iteratoren

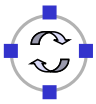
```
List<RatNumber> list =
                    new ArrayList<RatNumber>();
list.add(new RatNumber(1, 3));
list.add(new RatNumber(1, 4));

Iterator<RatNumber> iter = list.iterator();
while (iter.hasNext())
          RatNumber rn = iter.next();
```



Implementierung von Generics

- Angabe eines generischen Typen (”Platzhalter”) bei Implementierung

```
private class Node<T> {
    T obj;
    Node<T> next;
    ...
    public void setNext(Node<T> n) {
      next = n;
    }
}
```



Implementierung von Generics /2

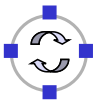
```
private class LinkedList<T> {

    private Node<T> head = null;

    public LinkedList() {
      head = new Node<T>();
    }
    ...
```



Implementierung von Generics /3

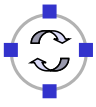
- Typsicherheit bei Verwendung

```
...
LinkedList<Student> l =
                    new LinkedList<Student>();
l.add(new Student(”Meier”));
l.add(new Student(”Schulze”));
l.add(new Student(”Schmidt”));
...
Student s = l.getFirst();
...
```



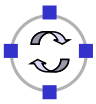
Zusammenfassung

- Datenstrukturen als Bausteine zur Datenverwaltung
- Statische Strukturen: Felder
- Dynamische Strukturen: Listen, Stack
- Vordefinierte Klassen in der Java-Bibliothek
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukture*n, Kapitel 13



Anhang: Weitere Datenstrukturen

- Weitere wichtige Datenstrukturen:
  - Bäume („Trees“)
    - Halden („Heaps“)
    - Binäre Suchbäume
  - Repräsentation von Graphen



Bäume (1)

- Listen und Felder können nur die Ordnung Ihrer Elemente in Ihrer Struktur repräsentieren (*lineare Struktur*).
  - Oft möchte man *hierarchische Stukturen* in einer Datenstruktur widerspiegeln. Bäume erlauben die Repräsentation solcher Strukturen.
  - Bäume können auch dafür eingesetzt werden, manche Operationen auf einer linearen Struktur effizienter zu realisieren.
- Ein *verwurzelter Baum* („rooted tree“) ist ein Baum mit einem Wurzelknoten („root“), der mit einigen Knoten verbunden ist, die die erste Hierarchieebene bilden, die wiederum jeweils mit Knoten der nächstniedrigen Hierarchiestufe verbunden sind.
- Hieraus ergeben sich Vater-Kind-Beziehungen zwischen den Knoten: der Vorgänger wird jeweils *Vaterknoten*, der Nachfolger jeweils *Kindknoten* genannt.
- Die maximale Anzahl von Kindern eines Knotens wird *„Grad“* des Baumes genannt („degree“). Ist der Grad = 2, so spricht man von einem *Binärbaum*.

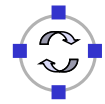


Bäume (2)

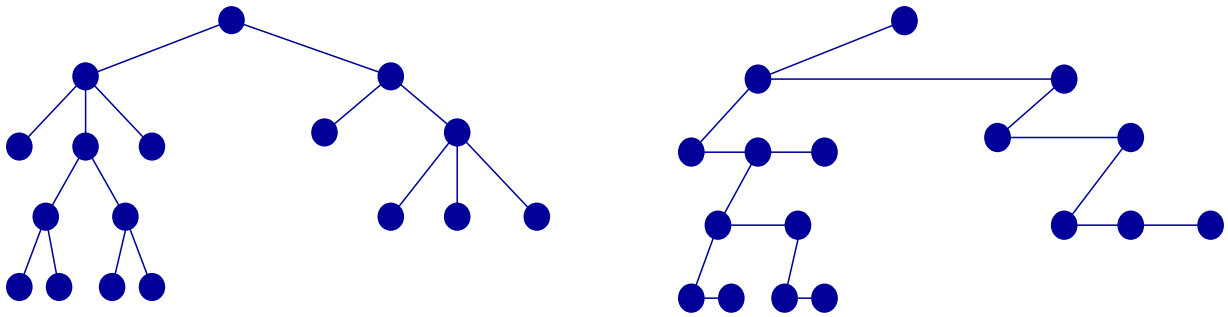
- Bei einem Binärbaum spricht man meist jeweils von *linken* und *rechten Kindknoten*
- Knoten, die selber keine Kindknoten mehr haben, heißen *Blätter*.
- Die *Höhe* eines Baumes ist die maximale Hierarchieebene eines Baumes, d.h. die maximale Distanz zwischen der Wurzel und einem Blatt.
- Jeder Knoten hat üblicherweise einen *Schlüssel* (z.B. Integer, String) und eventuell zusätzliche *Daten*. Solange keine Verwirrung entsteht, werden Schlüssel und Knoten oft miteinander identifiziert.

- Repräsentation von Bäumen in Programmen:
  - Explizite Repräsentation mit Knoten und Zeigern zwischen Knoten
  - Implizite Repräsentation mit Hilfe eines Feldes

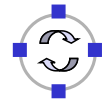
  Darstellung von Bäumen mit Knotengrad > 2?



Bäume (3)



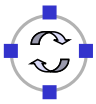
- Darstellung beliebiger Bäume als Binärbaum:
  - Der linke Nachfolger repräsentiert das erste Kind.
  - Der rechte Nachfolger repräsentiert den nächsten Geschwisterknoten.
- Nachteil:
  - Um auf ein bestimmtes Kind zuzugreifen, muss eine lineare Liste durchlaufen werden.



Bäume (4)

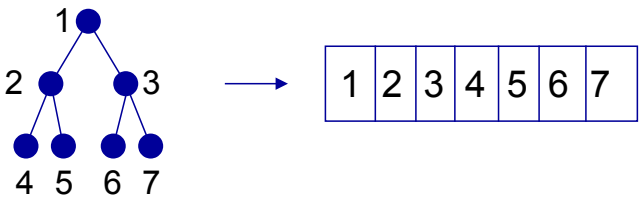
- Explizite Repräsentation

```
class Tree {
  Tree left;    // linker Teilbaum
  Tree right;   // rechter Teilbaum
  int key;      // Inhalt soll int sein.

  // Konstruktor
  Tree (int zahl)
  { this.left  = null;
    this.right = null;
    this.key = zahl;
  } // of Konstruktor
} // of class Tree
```

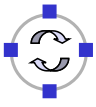


Bäume (5)

- Implizite Repräsentation:
  - Der Baum wird in einem Feld über dem Datentyp des Schlüssels repräsentiert (`int tree[n]`)
  - Die Wurzel wird in `tree[0]` gespeichert.
  - Ist ein Knoten im Eintrag `tree[i]` gespeichert, so werden seine beiden Kinder in den Einträgen `tree[2i+1]` und `tree[2i+2]` gespeichert.
  - Das Feld muss ausreichend groß dimensioniert sein, um das am weitesten unten (und rechts) stehende Kind aufnehmen zu können. Bei Binärbaum: $n = 2^{Höhe+1} - 1$

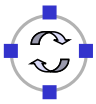


Achtung: nicht speichereffizient bei unbalancierten Bäumen!



Heaps (1)

- Ein *Heap* ist ein Binärbaum, dessen Knoten die folgende Eigenschaft erfüllen:
  - Der Schlüssel jedes Knotens ist größer oder gleich den Schlüsseln aller seiner Kindknoten
  - Alternativ kann die Heap-Eigenschaft auch lauten, dass der Schlüssel aller Knoten kleiner oder gleich der Schlüssel aller ihrer Kindknoten sein muss.
- Aufgrund der Transitivität der Größer- bzw. Kleiner-Beziehung folgt, dass der Schlüssel jedes Knotens jeweils größer oder gleich bzw. kleiner oder gleich den Schlüssel aller seiner (direkten und indirekten) Nachfolger ist.
- Heaps sind nützlich bei der Implementierung von sogenannten *Prioritätswarteschlangen*, einer abstrakten Datenstruktur mit den folgenden Operationen:
  - *Insert(x):* Füge x in die Prioritätswarteschlange ein
  - *Remove():* Entnehme das Element mit dem größten (bzw. kleinsten) Schlüssel.



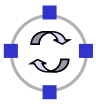
Heaps (2)

- Heaps können mit sowohl mit expliziter als auch mit impliziter Repräsentation implementiert werden:
  - Die explizite Repräsentation wird benötigt, wenn die maximale Anzahl *m* von Elementen nicht vor der Implementierung bestimmt werden kann.
  - Die implizite Repräsentation kann gewählt werden, wenn die maximale Anzahl von Elementen *m* bekannt ist und sie ist auch speichereffizient, da ein Heap balanciert gehalten werden kann.
- Wir zeigen hier die Implementierung der impliziten Repräsentation:
  - Die Knoten des Heaps werden somit in einem Feld *F[0, ..., m]* gespeichert
  - *Remove():*
    - Sei die aktuelle Anzahl von Elementen n, d.h. aktuell sind die Elemente des Heaps in dem Teilfeld F[0, ..., n-1] gespeichert.
    - Das größte Element ist immer in *F[0]* gespeichert. Ein Remove entfernt daher immer *F[0]*. Unsere Aufgabe besteht nun darin, ein Element des Teilfelds *F[1, ..., n-1]* in *F[0]* zu verschieben, so dass die Heap-Eigenschaft gewahrt bleibt und *n* zu dekrementieren.



Heaps (3)

- Remove():
  - Zunächst kopieren wir hierzu *F[n-1]* in *F[0]* und dekrementieren *n*.
  - Nun muss noch die Heap-Eigenschaft wiederhergestellt werden
  - Hierzu setzen wir zunächst p = 0 und c = 1 und betrachten dann jeweils die Schlüssel von F[p] und seiner beiden Kinder F[c] und F[c+1]:
    - Wenn F[p] größer oder gleich ist als seine beiden Kinder, ist die Heap-Eigenschaft bereits erfüllt und wir sind fertig.
    - Ansonsten wird zunächst das größere der beiden Kinder bestimmt (also ggf. c um 1 inkrementiert), danach F[p] mit F[c] vertauscht und anschließend geprüft, ob die Heap-Eigenschaft in dem Unterbaum ab F[c] erfüllt ist. Hierzu wird p = c und c = 2c + 1 gesetzt und erneut geprüft.
  - Die maximale Anzahl von Vergleichen ist hierbei $2\lceil \log_2 n \rceil$



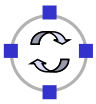
Heaps (4)

```
algorithm Remove(int F[], n)
  Eingabe: Ein Heap im Feld F und die Anzahl der Elemente n
  Ausgabe: das maximale Element top, modifizierter Heap F, n
  if n = 0 then print(„Empty Heap!“); fi
  top = F[0]; F[0] = F[n-1]; n = n-1; p = 0; c = 1;
  while c <= n-2 do
    if F[c] < F[c+1] then c = c + 1; fi // F[c] is biggest now
    if F[c] > F[p] then
      swap(F[p], F[c]);
      p = c; c = 2 * c + 1;
    else c = n - 1; // to stop the loop
    fi
  od
```



Heaps (5)

- Insert(x):
  - Zunächst fügen wir x am Ende des Arrays in F[n-1] ein.
  - Dann müssen wir es so lange mit seinen Vaterknoten vertauschen, wie es größer ist als sein Vaterknoten
  - Die maximale Anzahl der Vergleiche ist hierbei $\lceil \log_2 n \rceil$
- Es kann jede beliebige Sequenz von Insert- und Remove-Aufrufen in O(log n) pro Operation ausgeführt werden.
- Das sind allerdings auch die einzigen Operationen, die effizient mit einem Heap ausgeführt werden können.
- Bei der Suche nach einem bestimmten Element *x* hilft die Heap-Eigenschaft nicht (sehr), die Suche zu beschleunigen.
  - Ist das gesuchte Element größer als ein Knoten, kann es nicht mehr in dem Unterbaum ab diesem Knoten gespeichert sein. Dieses Wissen beschleunigt die Suche im allgemeinen Fall jedoch nicht wesentlich.



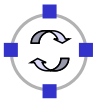
Heaps (5)

```
algorithm Insert(int x, F[], n)
  Eingabe: Element x, Heap F und Anzahl Elemente n
  Ausgabe: modifizierter Heap F und n
  if n = m - 1 then print(„Heap overflow!“); fi
  n = n + 1; F[n-1] = x; c = n-1; p = (c-1) DIV 2;
  while c > 0 do
    if F[c] > F[p] then
      swap(F[p], F[c]);
      c = p; p = (p - 1) DIV 2;
    else c = 0; // to stop the loop
    fi
  od
```



HeapSort (1)

- Heapsort ist ein weiterer effizienter Sortieralgorithmus:
  - In der Praxis ist er im mittleren Fall langsamer als Quicksort, jedoch nicht viel langsamer.
  - Allerdings ist seine Laufzeit von O(n log n) auch im schlechtesten Fall garantiert
  - Anders als Mergesort, ist er ein In-Place-Algorithmus, dass heisst er braucht nicht mehr Speicher als für die Speicherung der Elemente notwendig ist (Mergesort braucht doppelt soviel Speicher)



HeapSort (2)

```
algorithm HeapSort(F[], n)
  Eingabe: Feld F und Anzahl Elemente n
  Ausgabe: sortiertes Feld F und n
  buildHeap(F); // siehe folgende Folien
  for i = n-1 downto 1 do
    swap(F[0], F[i]);
    rearrangeHeap(i-1); // stelle Heap-Eigenschaft im Teilfeld
                        // F[0, ..., i-1] wieder her, d.h.
                        // heruntertauschen wie bei remove()
  od
```



Aufbau eines Heaps (1)

Problem *BuildHeap*

Gegeben: Feld F[0, ..., n-1] mit *n* Elementen in beliebiger Reihenfolge

Aufgabe: Ordne das Feld so um, dass die Heap-Eigenschaft erfüllt ist.

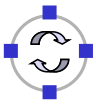
- Erster Ansatz: Bearbeite das Feld von links nach rechts (entspricht Top-Down bei grafischer Veranschaulichung)

  Induktionshypothese: Das Feld F[0, ..., i-1] ist ein Heap.

  Induktionsanfang: i = 1: trivial

  Induktionsschritt: i-1 ⇝ i

  - Im Induktionsschritt muss das i-te Element in den Heap eingefügt werden. Das leistet die Einfügeprozedur Insert wenn Sie mit Insert(F[i], F, i) aufgerufen wird.
  - Die Anzahl der Vergleiche ist im schlechtesten Fall jeweils $\lceil \log_2 i \rceil$
  - Insgesamt erfordert die Heap-Konstruktion so $\Theta(n \cdot \log_2 n)$ Schritte



Aufbau eines Heaps (2)

- Zweiter Ansatz: Bearbeite das Feld von rechts nach links (entspricht Bottom-Up bei grafischer Veranschaulichung)

  Induktionshypothese: Alle Bäume, die durch das Teilfeld F[i+1, ..., n-1] repräsentiert werden, erfüllen die Heap-Eigenschaft.

  Induktionsanfang: $i \in \{ \lfloor n/2 \rfloor + 1, ..., n-2 \}$: trivial
  Man beachte, das die Hälfte unserer Aufgabe durch Basisfälle der Induktion gelöst ist!

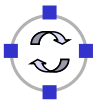
  Induktionsschritt: i+1 ⇝ i

  - Es reicht also, mit dem Induktionsschritt bei $\lfloor n/2 \rfloor$ zu beginnen.
  - Wir betrachten nun F[i]. Es hat maximal zwei Kinder F[2i+1] und F[2i+2]. Beide Kinder sind Wurzeln gültiger Heaps. Wir vergleichen F[i] nun mit dem größeren seiner Kinder:
    - Ist F[i] größer, so sind wir fertig mit F[i]
    - Ist F[i] kleiner, so tauschen wir es mit dem größeren Kind und tauschen es dann wie bei remove() in dem Teil-Heap herunter.



HeapSort (3)

- Der zweite Ansatz hat den Vorteil, dass erheblich weniger Arbeit zu leisten ist, da die Hälfte der Arbeit von dem Basisfall erledigt wird:
  - Eine detaillierte Analyse zeigt, dass der Aufwand für die Heap-Konstruktion mit dieser Methode O(n) beträgt (vergleiche [Manber89, S. 140]).
  - Der Grund dafür dass der Bottom-Up-Ansatz effizienter ist, liegt darin, dass es erheblich mehr Knoten im unteren Bereich des Baumes gibt, als es Knoten im oberen Bereich gibt.
  - Daher ist es effizienter, die Arbeit für die unteren Knoten zu minimieren.
- Insgesamt ergibt sich für Heap-Sort der Aufwand $O(n \log_2 n)$:
  - Der Aufwand für rearrangeheap(i) beträgt $\lceil \log_2 i \rceil$ Schritte und diese Prozedur wird (n-1) Mal aufgerufen (i durchläuft die Werte n-1 bis 0).



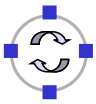
Binäre Suchbäume (1)

- Binäre Suchbäume (oft auch nur Binärbäume genannt) implementieren die folgenden Operationen auf effiziente Weise:
  - *Search(x):* Finde x der Datenstruktur bzw. stelle fest, dass x nicht in der Datenstruktur enthalten ist.
  - *Insert(x):* Füge x in die Datenstruktur ein.
  - *Delete(x):* Lösche x aus der Datenstruktur, wenn es darin enthalten ist.
- Die Grundidee von binären Suchbäumen ist es, dass jeder linke Kindknoten grundsätzlich kleiner ist als sein Vaterknoten und jeder rechte Kindknoten größer ist als sein Vaterknoten.
- *Search(x):*
  - Es wird zunächst x mit der Wurzel verglichen. Ist x = root.key, so ist der Eintrag gefunden.
  - Ist x kleiner, wird rekursiv im linken Teilbaum weitergesucht.
  - Ist x größer, wird rekursiv im rechten Teilbaum weitergesucht.



Implementierung von Binärbäumen (1)

```
class Node {
   private int key;
   private Node left, right;

   public Node Knoten(int x) {
    key = x ; left = null; right = null;
   }

   public int  getKey() { return key; }
   public void setKey(int x) { key = x; }
   public Node getLeft() { return left; }
   public void setLeft(Node n) { left = n; }
   public Node getRight() { return right; }
   public void setRight(Node n) { right = n; }
} // of class Node
```



Implementierung von Binärbäumen (2)

```
class BinaryTree {
  private Node root;

  public BinaryTree BinaryTree() {root == null; }

  private Node search(Node branch, int x);
  public Node search(int x);
  private void insert(Node branch, int x);
  public void insert(int x);
  private void delete(Node branch, int x);
  public void delete(int x);
} // of class BinaryTree
```



Implementierung von Binärbäumen (3)

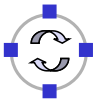
```
private Node BinaryTree::search(Node branch, int x) {
   if (branch.getKey() == x) return branch;
   if (x < branch.getKey())
      { if (branch.getLeft() == null)
           return null;
        else return search(branch.getLeft(), x); }
   else
      { if (branch.getRight() == null)
           return null;
        else return search(branch.getRight(), x); }
} // of Node BinaryTree::search(Node branch, int x)

public void BinaryTree::search(int x) {
   if (root == null ) return null;
   else return(search (root, x));
} // of BinaryTree::search(int x)
```



Binäre Suchbäume (2)

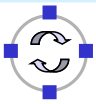
- *Insert(x):*
  - Ist der Baum leer, wird der Wert als Wurzelknoten eingefügt.
  - Ansonsten wird die richtige Stelle zum Einfügen gesucht
  - Die Suche stoppt bei einem Vaterknoten p, der:
    - kein linkes (bei x < p.key) bzw.
    - kein rechtes (bei x >= p.key) Kind hat.

    Der Schlüssel wird dann entsprechend in einen neu erzeugten linken bzw. rechten Kindknoten eingefügt.
  - Übrigens: Ist der Wert x bereits im Baum enthalten, wird er (bei der hier gezeigten einfachen Implementierung) einfach im rechten Teilbaum noch einmal an einer „passenden“ Stelle eingefügt.



Implementierung von Binärbäumen (4)

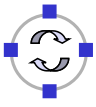
```
private void BinaryTree::insert(Node branch, int x) {
   if (key < branch.getKey())
      { if (branch.getLeft() == null)
           branch.setLeft(new Node(x));
        else insert(branch.getLeft(), x); }
   else
      { if (branch.getRight() == null)
           branch.setRight(new Node(x));
        else insert(branch.getRight(), x); }
} // of BinaryTree::insert(Node branch, int x)

public void BinaryTree::insert(int x) {
   if (root == null ) root = new Node(x);
   else insert(root, x);
} // of BinaryTree::insert(int x)
```



Binäre Suchbäume (3)

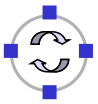
- *Delete(x):*
  - Löschen in Binärbäumen ist komplizierter als Suchen und Einfügen
  - Das Löschen von Blattknoten (also ohne Kinder) ist einfach: Es wird einfach der Zeiger im Baum auf den zu löschenden Knoten auf `null` gesetzt.
  - Das Löschen von Knoten mit nur einem Kind ist auch einfach: Es wird einfach der Zeiger im Baum auf den zu löschenden Knoten auf den (einzigen) Kindknoten gesetzt.
  - Soll ein Knoten mit zwei Kindern gelöscht werden, so vertauscht man zunächst seinen Schlüssel mit dem Schlüssel von entweder:
    - dem Knoten mit dem größten Schlüssel im linken Teilbaum, oder
    - dem Knoten mit dem kleinsten Schlüssel im rechten Teilbaum.

    Dieser Knoten hat maximal ein Kind (mit kleinerem bzw. größerem Schlüssel) und kann daher einfach gelöscht werden.



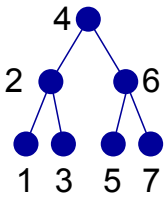
Implementierung von Binärbäumen (5)

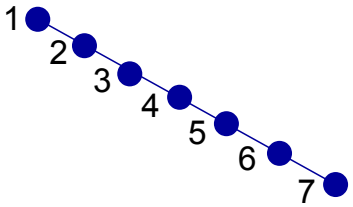
```
public void BinaryTree::delete(int x) {
  if (root != null) delete(root, x);
} // of void BinaryTree::delete(int x);

private void BinaryTree::delete(Node branch, int x) {
  // Your task... Euer Prof. ist ein Schuft! :o)
  // Tip: Mit Hilfsfunktionen wird es einfacher
} // of void BinaryTree::delete(Node branch, int x);
```



Binärbäume: Komplexität der Operationen (1)

- Die Laufzeit von Search(x), Insert(x) und Delete(x) hängt von der Gestalt des Baumes und dem Platz des Schlüssels x im Baum ab:
  - Im schlimmsten Fall muss hierfür der Baum bis hin zu dem „tiefsten“ Blattknoten durchlaufen werden.
  - Alle weiteren Schritte (Erzeugen von Knoten, Austausch von Schlüsseln beim Löschen etc.) benötigen nur konstanten Aufwand
- Ist der Baum „ausgeglichen“ („balanciert“), dann erfordern die Operationen im schlechtesten Fall $O(\log_2 n)$ Schritte.
- Ist der Baum zu einer Liste degeneriert, dann erfordern die Operationen im schlechtesten Fall $O(n)$ Schritte.

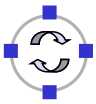






Binärbäume: Komplexität der Operationen (2)

- Eine detaillierte Analyse (siehe weitere Vorlesungen) zeigt:
  - Im schlechtesten Fall beträgt die Baumhöhe stets n.
  - Werden die Schlüssel in zufälliger Reihenfolge eingefügt und keine Löschoperationen ausgeführt so ist der Erwartungswert der Baumhöhe 2 ln(n) (ln bezeichnet den natürlichen Logarithmus)
  - Löschoperationen können auch bei zufälliger Reihenfolge zu hohen (und damit ineffizienten) Bäumen führen.
    - Das kann teilweise dadurch abgeschwächt werden, dass abwechselnd der größte Nachfolger im linken bzw. der kleinste Nachfolger im rechten Teilbaum für die Vertauschung gewählt wird.
  - Werden Einfüge- und Löschoperationen zufällig gemischt, kann sich im mittleren Fall eine Höhe von $O(\sqrt{n})$ ergeben.
  - Glücklicherweise existieren bessere Datenstrukturen für Bäume, die eine balancierte Gestalt bei den Operationen garantieren. (AVL-Bäume, Rot-Schwarz-Bäume etc. → weitere Vorlesungen)



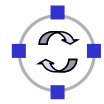
Repräsentation von Graphen (1)

- Wir erinnern uns zunächst an einige Zusammenhänge aus Kapitel 6:
  - Ein Graph G=(V, E) besteht aus einer Menge V von Knoten (vertices) und einer Menge E von Kanten (edges)
  - Jede Kante in E entspricht einem Paar von unterschiedlichen Knoten
  - Ein Graph kann gerichtet oder ungerichtet sein. Bei gerichteten Graphen ist die Reihenfolge (v, w) der Knoten bei Kanten wichtig. Eine Kante (v, w) können wir uns als Pfeil von v nach w vorstellen
  - Bei ungerichteten Graphen ist die Reihenfolge der Knoten (v, w) bei Kanten unwichtig, da Kanten in beiden Richtungen „verwendet“ werden können
- Frage: Wie repräsentiert man Graphen in einem Programm?

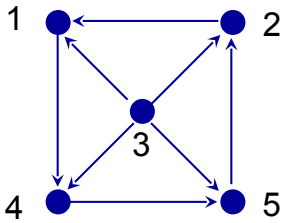


Repräsentation von Graphen (2)

- Wir nehmen an, dass die Knoten durch Ganzzahlwerte identifiziert werden (bei 1 beginnend, aufsteigend, ohne Lücken).
- Alternativen:
  - A.) Speichern in einer *Adjazenzmatrix* M: Existiert eine Kante vom Knoten i zum Knoten j, so steht eine 1 in M[i, j], sonst eine 0.
  - B.) Speichern in *Adjazenzlisten*: Es wird ein Feld K mit allen Knoten gespeichert. Der Eintrag K[i] zeigt auf eine verkettete Liste, die alle Nachfolger des Knotens i enthält.
  - C.) Speichern in einem *Knoten-Kanten-Feld*: Werden keine Kanten entfernt, so können die Knoten und Kanten wie folgt in einem Feld der Größe |V| + |E| gespeichert werden:
    - In den ersten |V| Einträgen des Feldes stehen die „Startpositionen“ im Feld auf die Kantenbeschreibungen zu den betreffenden Knoten.
    - Eine Kantenbeschreibung besteht aus dem Knotennamen, auf den die Kante zeigt.



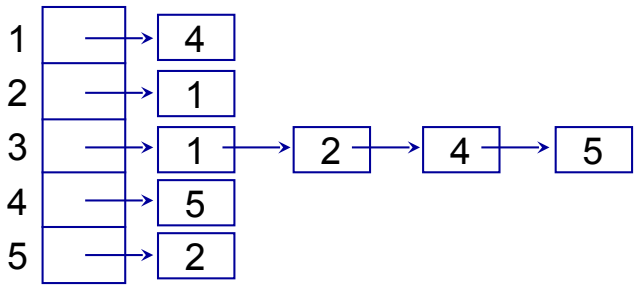
Repräsentation von Graphen (3)



A.)

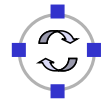
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	0
3	1	1	0	1	1
4	0	0	0	0	1
5	0	1	0	0	0

B.)



C.)

6	7	8	12	13	4	1	1	2	4	5	5	2
1	2	3	4	5	6	7	8	9	10	11	12	13



Zusammenfassung zum Anhang

- Bäume und Graphen erlauben die Repräsentation von komplexeren Strukturen als rein lineare Anordnungen:
  - Sie können auch bei „linear angeordneten“ Problemen, wie dem Sortieren von Daten eingesetzt werden, um eine effiziente Realisierung zu ermöglichen.
  - Teilweise sind Sie aufgrund der Gestalt des Problems angeraten, beispielsweise:
    - Freundesbeziehungen in einer Menge von Personen
    - Abhängigkeiten zwischen mehreren zu erledigenden Arbeitsschritten
  - Es gibt unterschiedliche Varianten, Bäume und Graphen in einem Programm zu repräsentieren.
    - Bäume: Feld oder explizit mit verzeigerten Knoten
    - Graphen: Adjazenzmatrix, Adjazenzlisten, Knoten-Kanten-Liste, oder auch mit einer Liste von Adjazenzlisten

