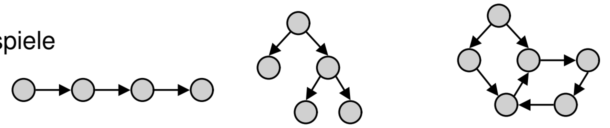
Kapitel 7: Dynamische Datenstrukturen

- Motivation
  - Länge eines Arrays ist nach der Erzeugung festgelegt.
  - hilfreich wären unbeschränkt große Datenstrukturen.
  - Lösungsidee: Verkettung einzelner Objekte zu größeren Strukturen.
- Beispiele



Liste Baum allgem. Graph

- Charakterisierung
  - Knoten werden zur Laufzeit (also dynamisch) erzeugt und verkettet.
  - Strukturen können dynamisch wachsen und schrumpfen.
  - Größe einer Struktur ist nur durch verfügbaren Speicherplatz beschränkt und muss nicht im vorhinein bestimmt werden.

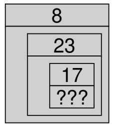
Beispiele

- Liste
  - Jeder Knoten (außer dem letzten) hat genau einen Nachfolger.
  - Jeder Knoten (außer dem ersten) hat genau einen Vorgänger.
- Baum
  - Ein Knoten kann mehrere Nachfolger haben („Verzweigungsgrad").
  - Jeder Knoten (außer der Wurzel) hat genau einen Vorgänger.
  - Modellierung von Hierarchien (Bsp. Teilestruktur eines Fahrzeugs).
  - Bsp. Binärbaum: Jeder Knoten hat höchstens zwei Nachfolger.
- Allgemeiner Graph
  - Knoten können beliebige Vorgänger und Nachfolger haben.
  - Folge: Es können Zyklen gebildet werden.
  - Modellierung von Netzen (Bsp. Straßennetz, Schienennetz).

Verkettung von Knoten

- Rekursive Definition eines Knotens

```
class Node {
    int val;           // Nutzinformation, hier: int
    Node next;         // rekursive Verkettung: Referenz
    Node (int v)       { val = v; } /* next == null; */
}
```

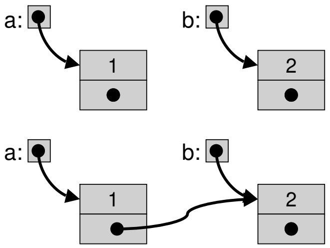


- Erzeugen von Knoten

```
Node a = new Node (1);
Node b = new Node (2);
```

- Verkettung der Knoten

```
a.next = b;
```

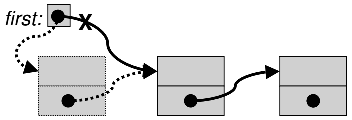


Einfügen in verkettete Liste

- Verankerung der Liste

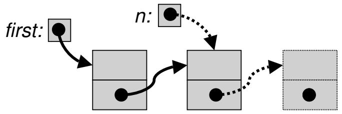
```
static Node first = null;
```

- Am Listenanfang einfügen




```
static void insert (int value) {
    Node n = new Node (value);
    n.next = first;
    first = n;
}
```

- Am Listenende einfügen




```
static void append (int value) {
    Node nn = new Node (value);
    if (first == null) {
      first = nn;
    } else {
      Node n = first;
      while (n.next != null) n = n.next;
      /*n.next == null*/ // n is last node
      n.next = nn;
    }
}
```

Durchlaufen einer Liste

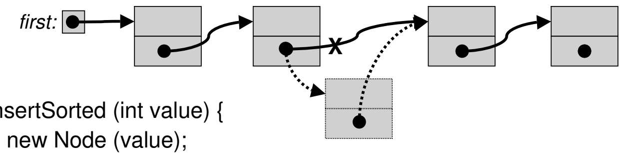
- Suchen, ob ein Wert enthalten ist

```
static boolean contains (int v) {
    Node n = first;                          // Starte am Listenanfang
    while (n != null && n.value != v) {      // null markiert Ende der Liste
      n = n.next;                            // Schritt zum nächsten Knoten
    } /* n == null || n.value == v */
    return (n != null);
}
```

- Ausgabe aller Listenelemente

```
for (Node n = first; n != null; n = n.next) {
    System.out.println (n.value);
}
```

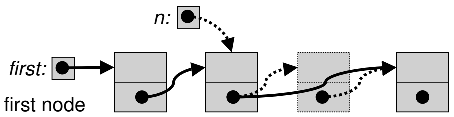

Sortiertes Einfügen in sortierte verkettete Liste




```
static void insertSorted (int value) {
    Node n = new Node (value);
    if (first == null) {
      first = n;
    } else {
      Node p = first;      // locate predecessor
      while (p.next != null && p.next.value < value) {
        p = p.next;
      } /*p.next == null || p.next.value >= value*/
      n.next = p.next;    // connect to successor
      p.next = n;         // connect with predecessor
    }
}
```


Entfernen aus einer Liste

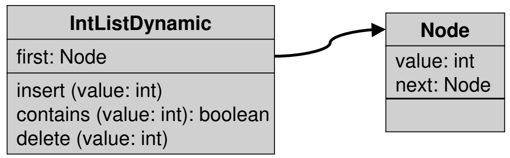
- Entferne den Knoten mit dem Wert v
  - Durchlaufe Liste bis zum Vorgänger des zu entfernenden Knotens.
  - Verknüpfe Vorgänger des zu entfernenden Knotens mit dessen Nachfolger.




```
static void delete (int v) {
    if (first != null) {
      if (first.value == v) {
        first = first.next;     // remove first node
      } else {
        Node n = first;       // locate predecessor
        while (n.next != null && n.next.value != v) {
          n = n.next;
        } /* n.next == null || n.next.value == v */
        if (n.next != null)
          n.next = n.next.next;
      }
    }
}
```


IntList als dynamische Liste

- Klassendiagramm



- Java-Code

```
class IntListDynamic implements IntList {
    Node first;
    public void insert (int value) { … }
    …
}
```

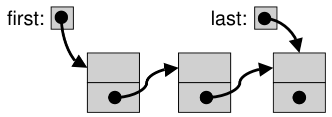
IntList als dynamische Liste (Java)

```
class Node {
    int value;
    Node next;
}

class IntListDynamic implements IntList {
    private Node first;

    public IntListDynamic () {
        first = null;
    }

    /** am Listenanfang einfügen */
    public void insert (int value) {
        Node n = new Node ();
        n.value = value;
        n.next = first;
        first = n;
    }

    public boolean contains (int value) {
        Node n = first;
        while (n != null && n.value != value) {
            n = n.next;
        } /* n == null || n.value == value */
        return (n != null);
    }

    public void delete (int value) {
        Node n = first, p = null;
        while (n != null && n.value != value) {
            p = n; n = n.next;
        } /* n == null || n.value == value */

        if (n != null) { /* n.value == value */
            if (n == first) first = n.next;
            else p.next = n.next;
        }
    }
}
```


Doppelt verankerte Liste

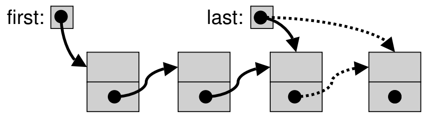
- Zusätzliche Variable für Listenende

```
Node first = null;
Node last = null;
```



- Einfügen am Listenende

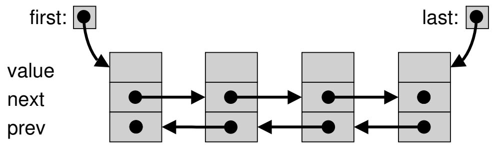
```
void append (int value) {
    Node n = new Node (value);
    if (first == null) first = n;
    else last.next = n;
    last = n;
}
```



Vorteil: kein Durchlauf nötig

Doppelt verkettete Liste

- Vorwärts- und Rückwärtsverkettung




```
class Node {
    int value;      // Nutzinformation
    Node next;      // Nachfolgerknoten (auch succ: successor)
    Node prev;      // Vorgängerknoten (auch pred: predecessor)
}
```

- Vorteil
  schnelles Entfernen am Anfang und am Ende der Liste.

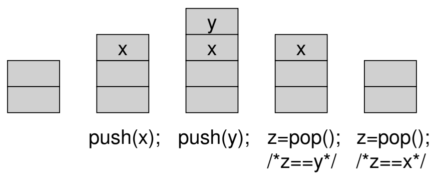
Beispiel Keller (Stapel, *Stack*)

- Spezifikation
  Kellerspeicher mit zwei Operationen:

push (v)	lege v auf Stapel ab.
v = pop ()	entferne oberstes Element, lege es in v.

LIFO-Prinzip: „Last In – First Out“

- Beispiel



Implementierung als verkettete Liste

```
public class Stack {
    private Node top = null;

    public void push (int v) {
        Node n = new  Node (v);
        n.next = top; top = n;
    }
    /** pop is valid only if not isEmpty */
    public int pop () {
        if (top != null) {
            Node p = top; top = top.next; return p.val;
        } else
            /* Fehlerbehandlung */
    }
    public boolean isEmpty () { return (top == null); }
}
```

Beispiel Warteschlange (*Queue*)

- Spezifikation
  Struktur mit zwei Operationen:
  put (v) füge v an Schlange an.
  v = get () entferne vorderstes Element, lege es in v.

  FIFO-Prinzip: „First In – First Out"

- Beispiel

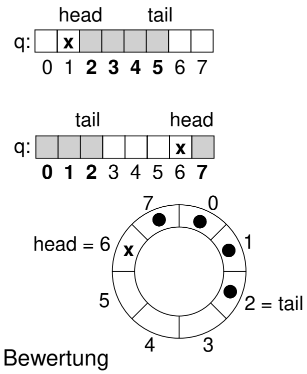
```
Queue q = new Queue();
q.put (x);
q.put (y);
z = q.get();     // z == x
z = q.get();     // z == y
```

Implementierung als verkettete Liste

```
public class Queue {
    private Node first = null;      // „head"
    private Node last = null;       // „tail"

    public void put (int v) {
        Node n = new  Node (v);
        if (first == null) first = n;
        else last.next = n;
        last = n;
    }
    /** valid only if queue is not empty */
    public int get () {
        if (first != null) {
            Node p = first; first = first.next;
            if (first == null) last == null;
            return p.val;
        } else
            /* Fehlerbehandlung */
    }
}
```


Warteschlange als zyklisches Array

- Schema



- Bewertung
  + spart Speicherplatz
  – beschränkte Länge

```
public class Queue {
    private int[] q = new int[capacity];     Konstruktor!
    private int head = 0, tail = 0;

    public void put (int v) {
        if ((tail+1) % q.length != head) {
            tail = (tail+1) % q.length;
            q[tail] = v;
        } else
            /* Überlaufbehandlung */
    }
    /** valid only if queue is not empty */
    public int get () {
        if (head != tail) {
            head = (head + 1)%q.length;
            return q[head];
        } else
            /* Unterlaufbehandlung */
    }
}
```


Geordnete Binärbäume

- Rekursive Definition
  Ein geordneter Binärbaum
  – ist entweder leer oder
  – besteht aus einer Wurzel, einem linken und einem rechten Teilbaum, die selbst geordnete Binärbäume sind.

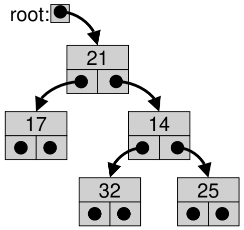
  Baum = ⟨⟩ | ⟨Wert, Baum, Baum⟩

- Beispiele
  ⟨⟩
  ⟨3, ⟨⟩, ⟨⟩⟩
  ⟨3, ⟨5, ⟨⟩, ⟨⟩⟩, ⟨⟩⟩
  ⟨21, ⟨17, ⟨⟩, ⟨⟩⟩, ⟨14, ⟨32, ⟨⟩, ⟨⟩⟩, ⟨25, ⟨⟩, ⟨⟩⟩⟩⟩

- Rekursive Datenstruktur (Java)

```
class Tree {
    int value;
    Tree left;
    Tree right;
}
```

- Beispiel



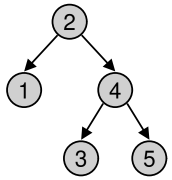
Baumdurchlauf

- Durchlaufreihenfolgen
  Präorder: Wurzel, linker Teilbaum, rechter Teilbaum
  Inorder: links – Wurzel – rechts
  Postorder: links – rechts – Wurzel

- Beispiel *Präorder-Durchlauf*

```
static void preorder (Tree t) {
    if (t != null) {
        System.out.println (t.value);
        preorder (t.left);
        preorder (t.right);
    }
}
```

- Beispiel



Präorder:	2, 1, 4, 3, 5
Inorder:	1, 2, 3, 4, 5
Postorder:	1, 3, 5, 4, 2