TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Datenstruktur Liste

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Formale Definition von Listen

Eine **Liste über einem Typ** T ist eine total geordnete Multimenge mit Elementen aus T (bzw. eine Folge, d.h. eine Abbildung $\mathbb{N} \to T$).

Eine Liste heißt **endlich**, wenn sie nur endlich viele Elemente enthält.

Bemerkung:

- Wir betrachten hier zunächst nur Listen aus `int`-Werten.
- Um Listen abstrakt zu repräsentieren, verwenden wir hier die folgende Schreibweise: `[6,-3,84]`
- Die leere Liste wird dabei als `[]` repräsentiert.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Implementierung der Datenstruktur Liste in Java

Es gibt viele verschiedene Möglichkeiten Listen zu implementieren.

Wir betrachten hier zunächst drei Implementierungsvarianten:

1. als einfachverkettete Liste
2. als Array-Liste
3. als doppeltverkettete Liste

Dabei konzentrieren uns auf das Einfügen von neuen Elementen.

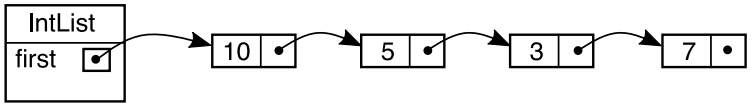
Einfachverkettete Listen

Einfachverkettete Listen

Bei einfachverketteten Listen wird für jedes Listenelement ein Verbund mit zwei Komponenten angelegt:

- zum Speichern des Elements
- zum Speichern der Referenz auf den Rest der Liste.

Die Liste mit den Elementen `[10,5,3,7]` erhält also folgende Repräsentation:



Implementierung: Listenknoten

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

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
    int getValue() {
     return value;
    }
    Node getNext() {
     return next;
    }
    void setNext(Node n) {
     next = n;
    }

}
```


Implementierung: Liste

```
public class IntList {
  private Node first;

  IntList() {
   first = null;
  }
  // fuegt ein Element am Ende der Liste ein
  void add (int elem) {
   Node newNode = new Node(element, null);
    if (first == null) {
      first = newNode;
    } else {
      Node n = first;
      Node nnext = n.getNext();
      while (nnext != null) {
        n = nnext;
      }
      n.setNext(newNode);
    }
  }
  // etc.
}
```

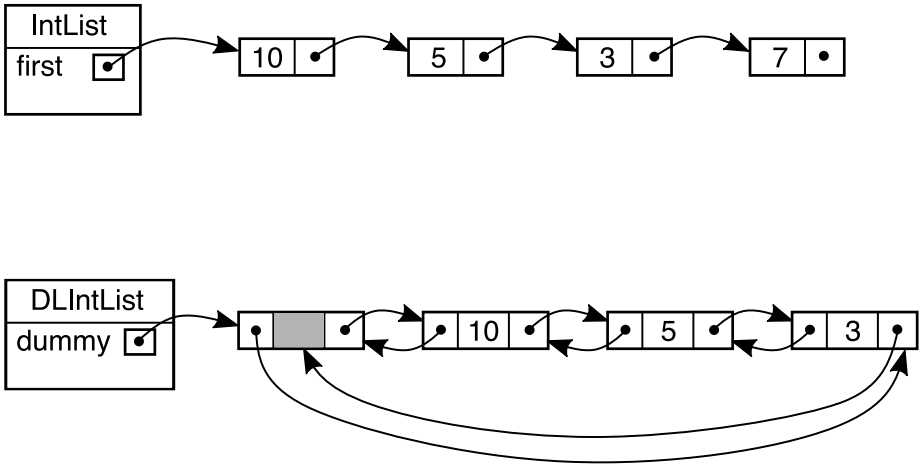
Doppeltverkettete Listen

Doppeltverkettete Listen

- Einfachverkettete Listen kann man nur effizient in eine Richtung durchlaufen, nämlich vom ersten Element zum letzten.

Idee: Doppelte Verkettung
Jeder Knoten zeigt nicht nur auf den Nachfolgerknoten (`Node`-Attribute `next`), sondern auch auf den Vorgängerknoten (zusätzliches `Node`-Attribute `prev`).

Skizze



Bemerkungen

- Problem: Das Einfügen am Ende der Liste erfordert zuerst vollständige Durchlaufen der Liste.
  - Lösung 1: Verwaltete Referenz auf das letzte Element der Liste (siehe Übung)
  - Lösung 2: Füge ein Dummy-Element bei der doppeltverketteten Liste ein
    - `next`-Referenz zeigt auf ersten Knoten der Liste
    - `prev`-Referenz zeigt auf letztes Element der Liste
    - Dies vermeidet Spezialfälle bei Implementierung der Methoden (z.B. beim Einfügen am Anfang der Liste)

Array-Listen

Grundidee: Array-Listen

- Elemente werden intern in einem Array gespeichert.
- Dazu wird zunächst ein Array mit einer bestimmten Anfangsgröße initialisiert.
- Das Array wird dann sukkzessive mit neuen Elementen gefüllt.
- Wenn das Array zu klein für neue Elemente ist, wird ein größeres erstellt und die alten Elemente werden in das neue Array kopiert.

Array-Liste: Konstruktor

```
public class IntArrayList {
    private int[] elementData;
    private int size;

    public IntArrayList(int initialArrayLength) {
     elementData = new int[initialArrayLength];
     size = 0;
    }
```


Array-Liste: Einfügen

```
// Element am Ende der Liste einfuegen
public void add(int element) {
    ensureCapacity(size + 1);
    elementData[size] = element;
    size++;
}

// stellt sicher, dass Array genug Platz hat
private void ensureCapacity(int minCapacity) {
    if (elementData.length < minCapacity) {
        // Groesse verdoppeln, mindestens auf minCapacity
        int newSize = Math.max(minCapacity,
                               2*elementData.length);
        elementData = Arrays.copyOf(elementData, newSize);
    }
}
```

Array-Liste: Element an Position

```
// liefert das Element an der gegebenen Position
public int get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return elementData[index];
}
```

- Beachte: Operationen müssen sicher stellen, dass nur der Teil des Arrays benutzt wird, der gültig ist.

Array-Liste: Suchen eines Elements

```
/** gibt die erste Position eines Elements in der Liste
  *  oder -1, wenn das Element nicht in der Liste ist */
public int indexOf(int element) {
    for (int i=0; i<size; i++) {
        if (elementData[i] == element) {
            return i;
        }
    }
    return -1;
}
```


Array-Liste: Entfernen eines Elements

```
// entfernt das 1. Vorkommen eines Elements aus der Liste
public void remove(int element) {
    int indexOf = indexOf(element);
    if (indexOf < 0) {
        // Element nicht vorhanden
        return;
    }
    // alle Elemente nach dem ersten Vorkommen
    // nach links verschieben:
    for (int i=indexOf; i<size-1; i++) {
        elementData[i] = elementData[i+1];
    }
    size--;
}
```

- Anmerkung: Löschen ist im Vergleich zu verketteten Listen aufwendig, insbesondere das Löschen am Anfang der Liste.

Iteratoren

Iteratoren

Iteratoren erlauben es, schrittweise über sogenannte Behälterdatenstrukturen (engl. *container*) wie Listen zu laufen, so dass alle Elemente der Reihe nach besucht werden.

Im Zusammenhang mit Kapselung sind sie unverzichtbar.

```
public class IntListIterator {
     //prueft, ob es noch weitere Eintraege gibt
     public boolean hasNext(){ ... }

     //liefert den naechsten Eintrag
     public int next() {...}
}
```


Beispiel: Iterator für einfachverkettete Liste I

Wir reichern die Klasse `IntList` mit Iteratoren an:

```
public class IntList {
  // Erweiterung um Iteratoren
  private Node first;
  ...
  public IntListIterator iterator() {
     return new IntListIterator(first);
  }
}
```


Beispiel: Iterator für einfachverkettete Liste II

```
1  import java.util.NoSuchElementException;
2
3  public class IntListIterator {
4    private Node current;
5    public IntListIterator(Node e) {
6      this.current = e;
7    }
8    //prueft, ob es noch weitere Eintraege gibt
9    public boolean hasNext(){
10     return current != null;
11   }
12   //liefert den naechsten Eintrag
13   public int next() {
14     if (current == null) {
15        throw new NoSuchElementException();
16     }
17     int res = current.getValue();
18     current = current.getNext();
19     return res;
20   }
21 }
```


Beispiel: Iterator für einfachverkettete Liste III

```
1  public class IntListTest {
2    public static void main(String[] args) {
3
4      IntList l = new IntList();
5      l.add(2);
6      l.add(-7);
7      l.add(34);
8
9      IntListIterator iter = l.iterator();
10     while(iter.hasNext()) {
11       StdOut.println(iter.next());
12     }
13   }
14 }
```

Bemerkung:

Der Iterator muss Zugriff auf die interne Repräsentation der Datenstruktur haben, über die er iteriert.

Iterator für Array-Liste I

Idee: Der Iterator hat eine Referenz auf das interne Array der Array-Liste. Das Attribut `position` gibt den `index` des nächsten Elements an.

```
public class IntArrayListIterator {
    private int[] elementData;
    private int size;
    private int position;

    IntArrayListIterator(int[] elementData, int size) {
        this.elementData = elementData;
        this.size = size;
        this.position = 0;
    }
```


Iterator für Array-Liste II

```
    public boolean hasNext() {
        return position < size;
    }

    public int next() {
        int elem = elementData[position];
        position++;
        return elem;
    }

}
```


Iterator für Array-Liste III

Beim Erstellen des Iterators (in der Klasse `IntArrayList`) wird das interne Array an den Iterator übergeben:

```
// liefert einen Iterator fuer die Liste
public IntArrayListIterator iterator() {
    return new IntArrayListIterator(elementData, size);
}
```

Bemerkung: Eine alternative Möglichkeit, die Kapselung zu gewährleisten, sind innere Klassen, die wir im weiteren Verlauf der Vorlesung noch behandeln werden.

Datenstruktur Baum

Baumartige Datenstrukturen

Nach Ottmann, Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Springer 2012

> *Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen.*

Anwendungen:

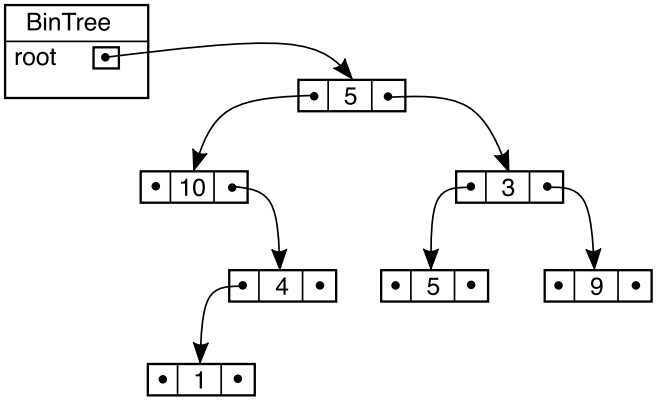
- Syntaxbäume
- Darstellung von Termen
- Dateisysteme
- Darstellung von Hierarchien (z.B. Typhierarchie)
- Implementierung von Datenbanken
- etc.

Begriffe

- In einem endlich verzweigten **Baum** hat jeder **Knoten** endlich viele **Kinder**.
- Üblicherweise sagt man, die Kinder sind von *links nach rechts geordnet*.
- Einen Knoten ohne Kinder nennt man ein **Blatt**, einen Knoten mit Kindern einen **inneren** Knoten oder **Zweig**.
- Den Knoten ohne Elternknoten nennt man **Wurzel**.
- Zu jedem Knoten k gehört ein **Unterbaum**, nämlich der Baum, der k als Wurzel hat.
- In einem **Binärbaum** hat jeder Knoten maximal zwei Kinder.

Markierte Bäume

- Ein Baum heißt **markiert**, wenn jedem Knoten k ein Wert/Markierung $m(k)$ zugeordnet ist.



Implementierung: Markierte Binärbäume

```
// Repraesentiert die Knoten eines Baums mit Markierungen
class TreeNode {
  private int mark;
  private TreeNode left, right;
  ...
}

// Repraesentiert einen markierten Binaerbaum
public class BinTree {
  private TreeNode root;
  ...
}
```

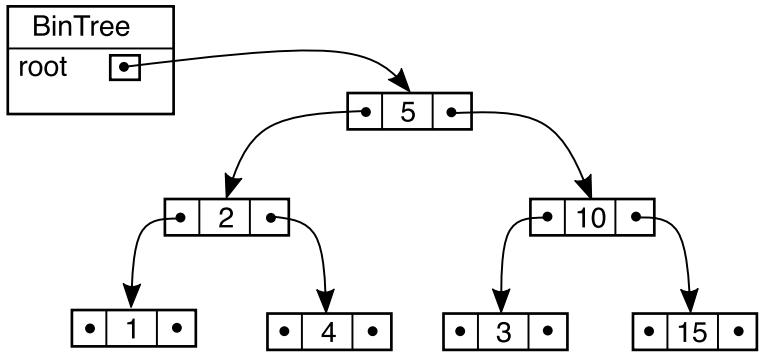
Definition: Sortiertheit markierter Binärbäume

Ein mit ganzen Zahlen markierter Binärbaum heißt **sortiert**, wenn für alle Knoten k gilt:

- Alle Markierungen der linken Nachkommen von k sind kleiner als oder gleich $m(k)$.
- Alle Markierungen der rechten Nachkommen von k sind größer als $m(k)$.

Aufgabe

Ist dieser markierte Binärbaum sortiert?



SortedBinTree: Suchen eines Eintrags I

```
// Repraesentiert einen sortierten markierten Binaerbaum
public class SortedBinTree {
  private TreeNode root;
  public SortedBinTree() {
   root = null;
  }
  // prueft, ob ein Element im Baum enthalten ist
  public boolean contains(int element) {
    return contains(root, element);
  }
  private boolean contains(TreeNode node, int element) {
    if (node == null) {
        return false;
    }
    if (element < node.getMark()) {
        // kleinere Elemente links suchen
        return contains(node.getLeft(), element);
    } else if (element > node.getMark()) {
        // groessere Elemente rechts suchen
        return contains(node.getRight(), element);
    } else {
        // gefunden!
        return true;
    }
  }
```


SortedBinTree: Suchen eines Eintrags II

SortedBinTree: Einfügen eines Eintrags I

Algorithmisches Vorgehen:

- Neue Knoten werden immer als Blätter eingefügt.
- Die Position des Blattes wird durch den Wert des neuen Eintrags festgelegt.
- Beim Aufbau eines Baumes ergibt der erste Eintrag die Wurzel.
- Ein Knoten wird
  - in den linken Unterbaum der Wurzel eingefügt, wenn sein Wert kleiner gleich ist als der Wert der Wurzel;
  - in den rechten, wenn er größer ist.

Dieses Verfahren wird rekursiv fortgesetzt, bis die Einfügeposition bestimmt ist.

SortedBinTree: Einfügen eines Eintrags II

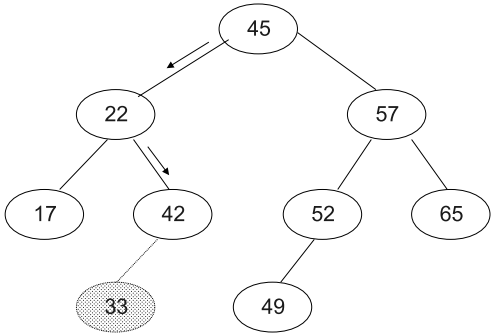
```
public void add(int element) {
   if (root == null) {
      root = new TreeNode(element);
   } else {
      add(root, element);
   }
}

private void add(TreeNode node, int element){
   if (element <= node.getMark()) {
      if (node.getLeft() == null) {
          node.setLeft(new TreeNode(e));
      } else {
          add(node.getLeft(), element);
      }
   } else {
     // Fall: node.getMark() < element
     if (node.getRight() == null) {
          node.setRight(new TreeNode(e));
      } else {
          add(node.getRight(), element);
      }
   }
}
```


Iterative Variante

```
public void addIteratively(int element) {
    if (root == null) {
        root = new TreeNode(element);
        return;
    }
    TreeNode node = root;
    while (true) {
        if (element <= node.getMark()) {
            if (node.getLeft() == null) {
                node.setLeft(new TreeNode(element));
                break;
            } else {
                node = node.getLeft();
            }
        } else {
           // Fall: element > node.getMark()
           // ... analog zum ersten Fall (jetzt rechts)
        }
    }
}
```


Beispiel

Einfügen von 33:

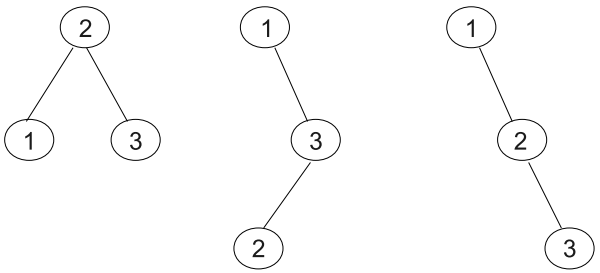


Bemerkungen

- Die Reihenfolge des Einfügens bestimmt das Aussehen des sortierten markierten Binärbaums:

  Reihenfolgen:

  2 ; 3 ; 1 1 ; 3 ; 2 1 ; 2 ; 3



- Bei sortierter Einfügereihenfolge entartet der Baum zur linearen Liste.

Zusammenfassung

- Klassische Datenstrukturen
  - Listen: Einfach-, doppeltverkettete Listen, Array-Listen
  - Bäume und ihre Eigenschaften
- Unterschiedliche Eigenschaften (u.a. Effizienz von Operationen)
- Iteratoren erlauben Kapselung von interner Repräsentierung