Algorithmen & Datenstrukturen 2

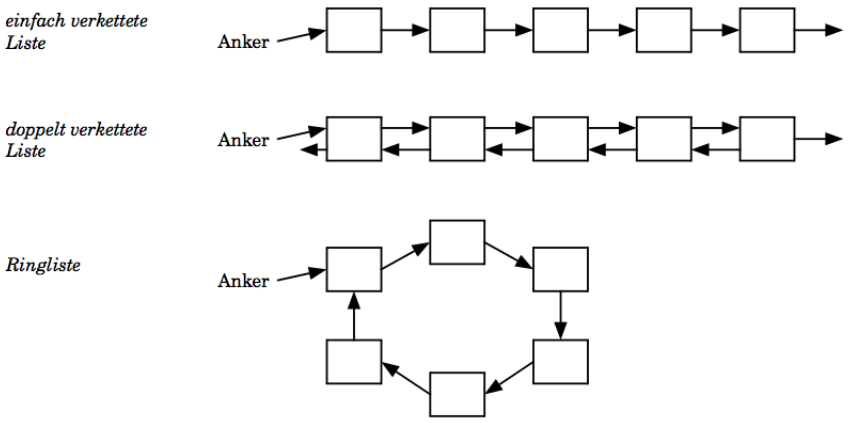
Jan Fässler

3. Semester (HS 2012)

Inhaltsverzeichnis

1 Listen

Eine verkettete Liste (linked list) ist eine dynamische Datenstruktur zur Speicherung von Objekten. Sie eignen sich für das speichern einer unbekannten Anzahl von Objekten, sofern kein direkter Zugriff auf die einzelnen Objekte benötigt wird. Jedes Element in einer Liste muss neben den Nutzinformationen auch die notwendigen Referenzen zur Verkettung enthalten.
Es gibt drei verschiedene Arten von Listen:



Listing 1: einfache Linked List

```
1 public class LinkedList<T> {
    private Element<T> head = null;
    private Element<T> last = null;
    public void add(T data) {
      last.next = new Element<T>(data);
6     last = last.next;
    }
    public void remove(T data) {
      Element<T> current = head;
      while (current != null && current.data != data) {
11      current = current.next;
        if (current != null && current.data == data) {
          current.last = current.next;
          current = null;
        }
16    }
    }
    public T getFirst() {
      return head.data;
    }
21  public T getLast() {
      return last.data;
    }
    public class Element<E> {
      public Element<E> next;
26    public Element<E> last;
      public E data;
      public Element(E input) {
        data = input;
      }
31  }
  }
```

1.1 Stack

Der Stack ist eine dynamische Datanstruktur bei der man nur auf das oberste Element des Stabels zugreifen (top), ein neues Element auf den Stabel legen (push) oder das oberste Element des Stapels entfernen (pop) kann.

Listing 2: Implementierung eines Stacks

```
public class Stack<T> extends LinkedList<T> {
  public T top() {
    return this.getLast();
  }
  public void push(T data) {
    this.add(data);
  }
  public T pop() {
    T last = this.getLast();
    this.remove(last);
    return last;
  }
  public boolean isEmpty() {
    return this.getFirst() == null ? true : false;
  }
}
```

1.2 Erweiterte Liste

Dies ist mal eine mögliche und vor allem nur teilweise Implementierung einer doppelt verlinkten Liste. Die Implementierung des Iterators und der sortierung sind ausgeklammert in Unterkapitel.

Listing 3: Liste mit Iterator

```
public class AdvancedComparableList<T extends Comparable<T>> implements
    Iterable<T> {
  private ListElement<T> head, foot;
  private int size = 0;
  public T getFirst() { return head.data; }
  public T getLast() { return foot.data; }
  public int size() { return size; }
  public boolean contains(T data) {
    boolean found = false;
    CListIterator<T> it = this.iterator();
    while (!found && it.hasNext()) if (it.next().equals(data)) found = true;
    return found;
  }
  public void add(T data) { add(size, data); }
  public void add(int index, T data) {
    if (index > size) throw new IndexOutOfBoundsException();
    else if (!this.contains(data)) {
      ListElement<T> newElement = new ListElement<T>(data);
      ListElement<T> current = head;
      if (size == 0) {
        head = newElement;
        foot = head;
      } else if (index == size) {
        newElement.last = foot;
        foot.next = newElement;
        foot = newElement;
      } else if(index == 0) {
```

```
        newElement.next = current;
        current.last = newElement;
        head = newElement;
      } else {
        for (int i=0; i<index; i++) current = current.next;
        newElement.next = current;
        newElement.last = current.last;
        if (current.last != null) current.last.next = newElement;
        current.last = newElement;
      }
      size++;
    }
  }
  public T remove() { return remove(size-1); }
  public T remove(T data) throws Exception {
    if (this.contains(data)) return remove(this.indexOf(data));
    else throw new Exception("Element "+data+" does not exist.");
  }
  public T remove(int index) {
    if (index >= size) throw new IndexOutOfBoundsException();
    T element = get(index);
    if (index == 0) {
      if (size > 1) head = head.next;
      else { head = null; foot = null; }
    } else if (index == (size-1)) {
      foot.last.next = null;
      foot = foot.last;
    } else {
      ListElement<T> current = head;
      for (int i=0; i<index; i++) current = current.next;
      current.last.next = current.next;
      if (current.next != null) current.next.last = current.last;
      current = null;
    }
    return element;
  }
  public int indexOf(T data) {
    int index = -1;
    if (this.contains(data)) { index++; while(!this.get(index).equals(data))
        index++; }
    return index;
  }
  public T get(int index) {
    T element = null;
    if (index >= 0 && index < size) element = this.iterator(index).next();
    return element;
  }
  public class ListElement<T extends Comparable<T>> implements Comparable<T>
      {
    public ListElement<T> next, last;
    public T data;
    public ListElement(T input) { data = input; }
    public int compareTo(T o) { return data.compareTo(o); }
    public String toString() { return String.valueOf(data) + ">"+String.
        valueOf(next); }
  }
}
```

1.2.1 Iterators

Die Schnittstelle java.util.Iterator, erlaubt das Iterieren von Containerklassen. Jeder Iterator stellt Funktionen namens next(), hasNext() sowie eine optionale Funktion namens remove() zur Verfügung. Der folgende ListIterator stellt auch noch Funktionen für rückwertsiterieren zur Verfügung, sowie die Möglichkeit den aktuellen Index abzufragen. Zudem kann damit noch direkt über den Iterator Elemente eingefügt oder ersetzt werden.

Listing 4: Iterators

```
public CListIterator<T> iterator() { return new CListIterator<T>(head,
    this); }
public CListIterator<T> iterator(int index) {
  if (index >= size) throw new IndexOutOfBoundsException();
  CListIterator<T> it = this.iterator();
  for (int i=0; i<index; i++) it.next();
  return it;
}
public static class CListIterator<E extends Comparable<E>> implements
    ListIterator<E> {
  private ListElement<E> nextElement, prevElement, lastReturned;
  private AdvancedComparableList<E> _list;
  private int index = 0;
  public CListIterator(ListElement<E> element, AdvancedComparableList<E>
      list) {
    _list = list;
    nextElement = element;
    prevElement = (element == null?null:element.last);
    ListElement<E> current = element;
    while (current != null && current.last != null) {
      index++;
      current = current.last;
    }
  }
  public boolean hasNext() {  return nextElement != null; }
  public E next() {
    index++;
    prevElement = nextElement;
    nextElement = nextElement.next;
    lastReturned = prevElement;
    return prevElement.data;
  }
  public boolean hasPrevious() { return prevElement != null; }
  public E previous() {
    index--;
    nextElement = prevElement;
    prevElement = prevElement.last;
    lastReturned = nextElement;
    return nextElement.data;
  }
  public int nextIndex() { return index; }
  public int previousIndex() { return index-1; }
  public void add(E data) { _list.add(previousIndex(), data); lastReturned
      = null; }
  public void remove() { _list.remove(previousIndex()); lastReturned =
     null; }
  public void set(E e) { lastReturned.data = e; }
}
```

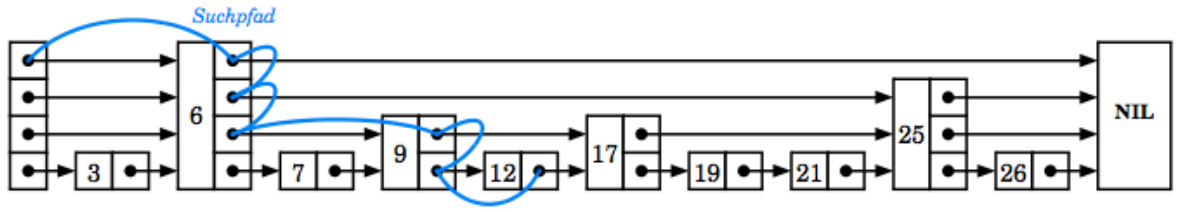
1.2.2 Merge Sort

Listing 5: Merge Sort

```
    /*     sorting      */
2   public void sort() { mergesort(0, this.size() - 1); }
    private void mergesort(int low, int high) {
      if (low < high) {
        if (high-low > 1) {
          int middle = (low + high) / 2;
7         mergesort(low, middle);
          mergesort(middle + 1, high);
          merge(low, middle, high);
        } else {
          if (this.get(low).compareTo(this.get(high)) > 0) {
12          T tmp = this.get(high);
            this.remove(high);
            this.add(low, tmp);
          }
        }
17    }
    }
    private void merge(int low, int middle, int high) {
      int iLeft = low, iRight = middle+1;
      while (iLeft <= high && iRight <= high) {
22      T right = get(iRight);
        if (get(iLeft).compareTo(right) > 0) {
          remove(iRight);
          add(iLeft, right);
          iRight++;
27      } else iLeft++;
      }
    }
```

1.3 Skip-Liste

Die Skip-Liste ist eine sortierte, einfach verkettete Liste, die uns aber ein schnelleres Suchen von Elementen in der Datenstruktur erlaubt. In einer sortierten, verketteten Liste müssen wir jedes Element einzeln durchlaufen bis wir das gewünschten Element gefunden haben. Wenn wir nun aber in der sortierten Liste auf jedem zweiten Element eine zusätzliche Referenz auf zwei Elemente weiter hinten setzen, dann reduziert sich die Anzahl zu besuchender Elemente auf einen Schlag um rund die Hälfte. Genau betrachtet müssen wir nie mehr als $(n/2)+1$ Elemente besuchen (n ist die Länge der Liste).



1.3.1 Beispiel

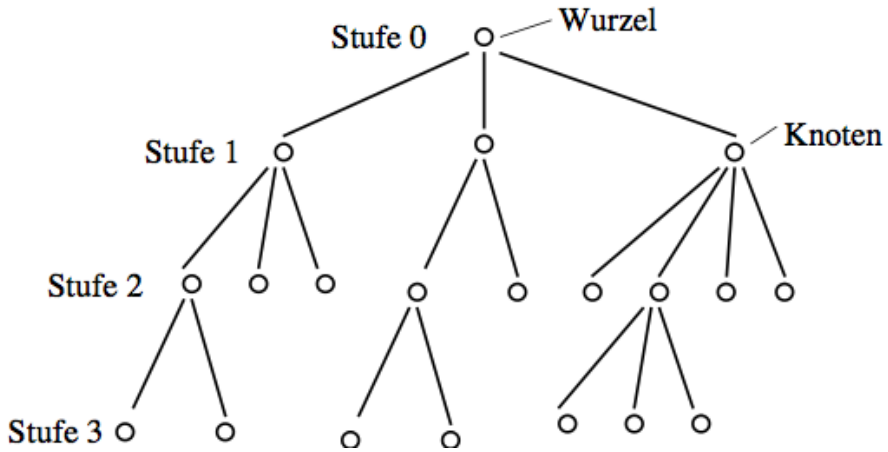
Listing 6: Skip List

```
1 public class SkipListe {
```

```
    private static double p = 0.7;
    private Element m_headAnchor; // kleinst moeglicher key, level = MaxLevel
    private Element m_tailAnchor; // groesst moeglicher key, level = MaxLevel
    private int m_maxLevel;       // speichert den maximalen Level
    public SkipListe(int maxLevel) {
      m_maxLevel = maxLevel;
      m_headAnchor = new Element(Integer.MIN_VALUE, null, m_maxLevel);
      m_tailAnchor = new Element(Integer.MAX_VALUE, null, m_maxLevel);
      for (int i = 0; i <= m_maxLevel; i++) m_headAnchor.setNext(i,
          m_tailAnchor);
    }
    public Object suchen(int key) {
      Element aktuell = m_headAnchor;
      for (int i = m_maxLevel; i >= 0; i--) {
        while (aktuell.getNext(i).getKey() <  key) aktuell = aktuell.getNext(i
            );
      }
      aktuell = aktuell.getNext(0);
      if (aktuell.getKey() == key) return aktuell.getData();
      else return Integer.MAX_VALUE;
    }
    public void einfuegen(int key, Object object) {
      Element[] update = new Element[m_maxLevel + 1];
      Element aktuell = m_headAnchor;
      for (int i = m_maxLevel; i >= 0; i--) {
        while (aktuell.getNext(i).getKey() <  key) aktuell = aktuell.getNext(i
            );
        update[i] = aktuell;
      }
      aktuell = aktuell.getNext(0);
      if (aktuell.getKey() == key) aktuell.setData(object);
      else {
        Element neuesElement = new Element(key,object,randomLevel());
        for (int i = 0; i <= neuesElement.getLevel(); i++) {
          neuesElement.setNext(i, update[i].getNext(i));
          update[i].setNext(i, neuesElement);
        }
      }
    }
    public void entfernen(int key) {
      Element[] update = new Element[m_maxLevel + 1];
      Element aktuell = m_headAnchor;
      for (int i = m_maxLevel; i >= 0; i--) {
        while (aktuell.getNext(i).getKey() <  key) aktuell = aktuell.getNext(i
            );
        update[i] = aktuell;
      }
      aktuell = aktuell.getNext(0);
      for (int i = 0; i <= aktuell.getLevel(); i++) update[i].setNext(i,
          aktuell.getNext(i));
      aktuell = null;
    }
    public int randomLevel() {
      int level = 0;
      while (level < m_maxLevel && Math.random() < p) level++;
      return level;
    }
  }
```

2 Bäume

Bäume sind verallgemeinerte Listenstrukturen. Ein Element, üblicherweise spricht man von Knoten (node), hat nicht, wie im Falle linearer Listen, nur einen Nachfolger, sondern eine endliche, begrenzte Anzahl von Söhnen. In der Regel ist einer der Knoten als Wurzel (root) des Baumes ausgeprägt. Das ist zugleich der einzige Knoten ohne Vorgänger. Jeder andere Knoten hat einen (unmittelbaren) Vorgänger, der auch Vater des Knotens genannt wird. Eine Folge p_0, ..., p_k von Knoten eines Baumes, die die Bedingung erfüllt, dass p_i+1 Sohn von p_i ist für $0 \leq i <$ k, heisst Pfad (path) mit Länge k, der p_0 mit p_k verbindet. Jeder von der Wurzel verschiedene Knoten eines Baumes ist durch genau einen Pfad mit der Wurzel verbunden.



2.1 Binäre Suchbäume

Ein binärer Suchbaum ist ein geordneter Baum mit Ordnung d = 2. In jedem Knoten wird ein Suchschlüssel so abgespeichert, dass alle Suchschlüssel des linken Teilbaums des Knotens kleiner und alle Suchschlüssel des rechten Teilbaums des Knotens grösser sind. Das heisst, dass an jedem Knoten alle kleineren Suchschlüssel über den linken Sohn und alle grösseren Suchschlüssel über den rechten Sohn erreicht werden.
Die so geordneten Knoten in einem balancierten binären Suchbaum erlauben ein schnelles Suchen. Der maximale Suchaufwand hängt direkt von der Höhe des Baumes ab und wächst nur logarithmisch mit der Anzahl der Knoten im Baum.

2.1.1 Traversieren

Das Durchlaufen kann auf mindestens drei verschiedene Arten erfolgen: Preorder, Inorder, Postorder.

Inorder
: 1. traversiere den linken Teilbaum des Knotens v; 2. besuche den Knoten v; 3. traversiere den rechten Teilbaum des Knotens v.

Preorder
: Bei Preorder wird zuerst Knoten v besucht, dann erst 99 der linke Teilbaum von v in Preorder und anschliessend noch der rechte Teilbaum von v in Preorder durchlaufen.

Postorder
: Hier wird zuerst der linke Teilbaum von v, dann der rechte Teilbaum von v und erst zum Schluss der Knoten v besucht.

2.2 Balancierte Bäume

Ein binärer Suchbaum ist AVL-ausgeglichen oder höhenbalanciert oder eben ein AVL- Baum, wenn für jeden Knoten v des Baumes gilt, dass sich die Höhe des linken Teilbaumes von der Höhe des rechten Teilbaumes von v höchstens um eins unterscheidet.

2.2.1 Berechnungen

h := Höhe = Maximal auftretende Tiefe
$bal(t) := h(t_r) - h(t_l)$ // $\in [-1, 0, 1] \Rightarrow$ AVL-Baum

TODO: wird hier etwa balance anders berechnet? Mir scheint dass beim Einfügen von $bal = h(t_l) - h(t_r)$ ausgegangen ist, denn wenn links eingefügt wird muss rechts etwas sein und dann wäre bal 1 und +1 wäre dann 2 und nicht 0. üblicherweise (also in Wikipedia etc) wird aber anders rum gerechnet.

2.2.2 Einfügen

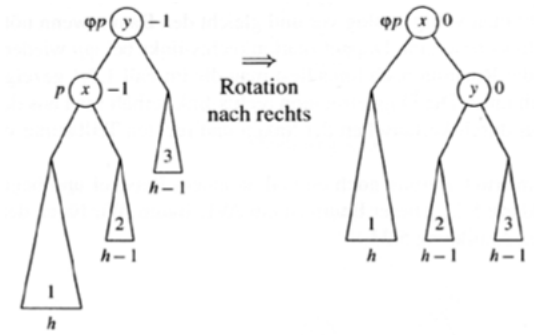
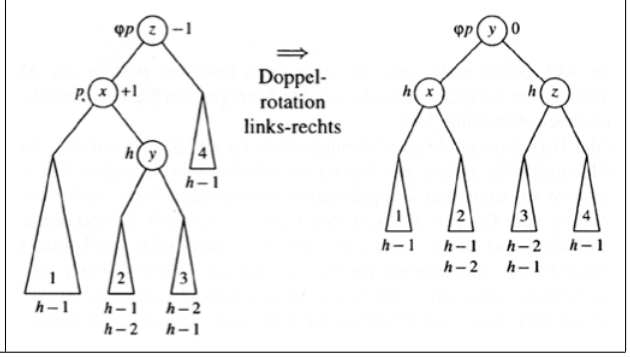
Fall 1: p hat 1 Sohn, einfügen an der leeren Stelle:
einf. Links: $bal(p) = +1 \rightarrow bal(p) = 0$
einf. Rechts: $bal(p) = -1 \rightarrow bal(p) = 0$
Höhe des Sohnes: $h(p) = const$
Balance des Vater: $bal(v) = const$
$\Rightarrow$ **fertig**

Fall 2: p hat keine Söhne, einfügen links/rechts
einf. links/rechts:$bal(p) = \pm 1$
Schieflage: $++h(p)$

Variante a.)
$bal(v) \neq 0$ / p ist kürzerer Ast
$\Rightarrow bal(v) = 0$
& $h(v) = const$
$\Rightarrow$ **fertig**

Variante b.)
$bal(v) = 0$
$\Rightarrow bal(v) = \pm 1$
& $++h(v)$
$\Rightarrow$ **weiter testen mit Vater/Opa/...**

Variante c.)
$bal(v) \neq 0$
& p ist längerer Ast
$\Rightarrow bal(v) = \pm 2! \Rightarrow$ **es muss balaciert werden:**

$sgn(bal(v)) = sgn(bal(p))$	$sgn(bal(v)) \neq sgn(bal(p))$
	

2.3 Löschen

Bei einem nicht ALV Baum ist der Aufwand für das löschen eines Elementes überschaubar:

I) p hat keinen Sohn
p entfernen

II) p hat einen Sohn
p entfernen und Sohn nachziehen

III) p hat zwei Söhne
p entfernen, durch nächstkleineres oder nächstgrösseres Element ersetzen
(dies kann eventuell zu einer weiteren Löschoperation vom Typ II weiter unten führen)

Bei einem ALV Baum muss nach dem löschen noch balanciert werden: