Algorithmen & Datenstrukturen 2

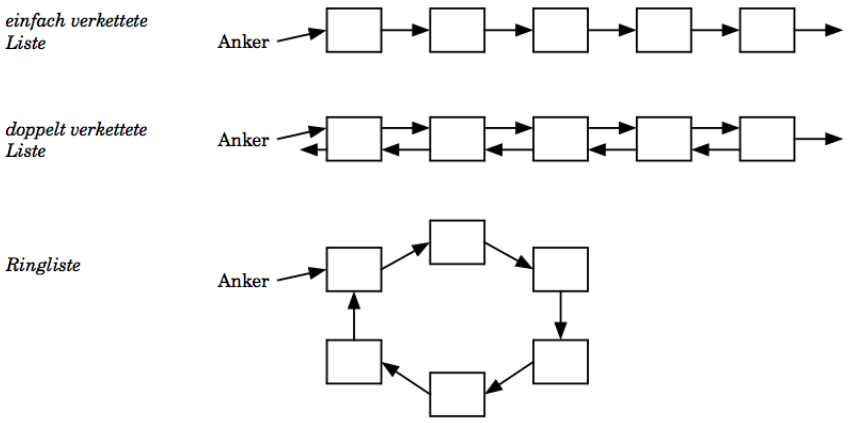
Jan Fässler

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; }
}
```

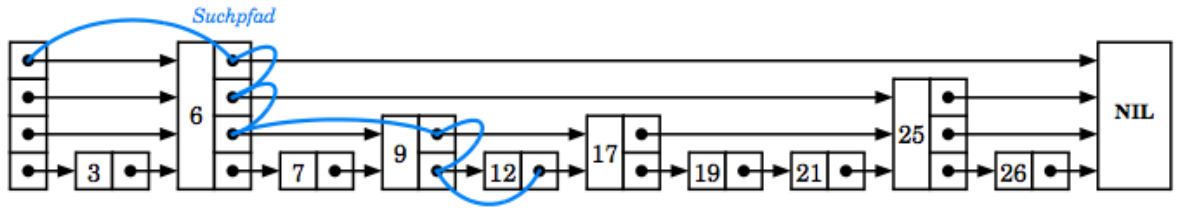
1.2.2 Merge Sort

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

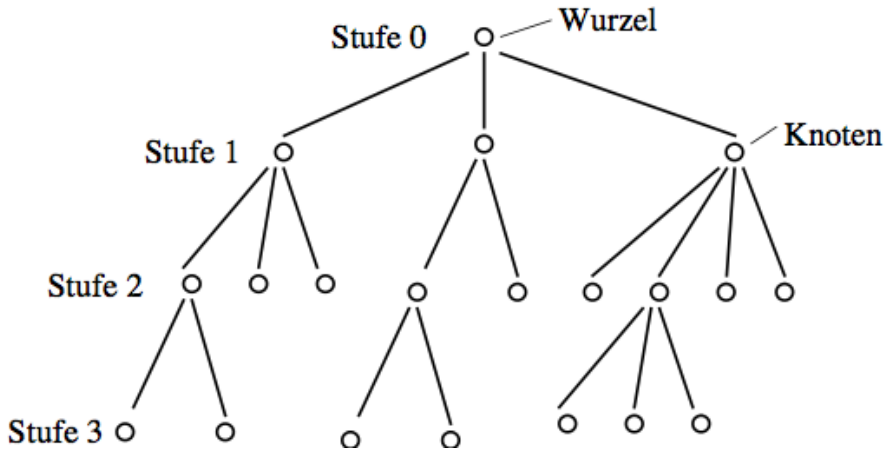
Listing 6: Skip List

```
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

2.2.2 Einfügen

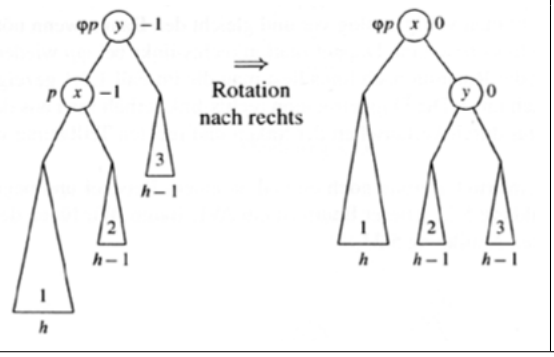
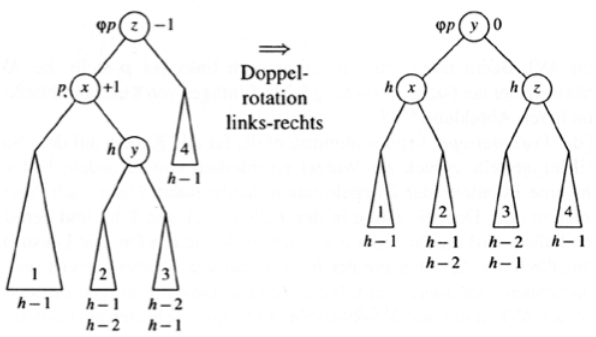
Fall 1: p hat 1 Sohn, einfügen an der leeren Stelle:
einf. Links: $bal(p) = +1 \rightarrow bal(p) = 0$
einf. Rechts: $bal(p) = -1 \rightarrow bal(p) = 0$
Höhe des Sohnes: $h(p) = const$ / Balance des Vater: $bal(v) = const$
$\Rightarrow$ **fertig**

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:**

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

2.2.3 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.

2.2.4 Beispiel

Listing 7: Insert & Delete from a AVL Tree

```
public T insert(int key, T value) {
    final Node<T> n = new Node<T>(key, value, null);
    return insert(this.root, n);
}
public T insert(Node<T> r, Node<T> n) {
    if (r == null) { size = 1; this.root = n; }
    else {
      if (n.key < r.key) {
        if (r.left == null) { // links
          r.left = n; n.parent = r; size++; balance(r);
        } else return insert(r.left, n);
      } else if (n.key > r.key) {
        if (r.right == null) { // rechts
          r.right = n; n.parent = r; size++; balance(r);
        } else return insert(r.right, n);
      } else {
        T oldValue = r.value; r.value = n.value; return oldValue;
      }
    } return null;
}
public T remove(int k) { return remove(k, root); }
private T remove(int key, Node<T> t) {
    if (t == null) return null;
    else {
      if (t.key > key)      return remove(key, t.left);
      else if (t.key < key) return remove(key, t.right);
      else                  return remove(t);
    }
}
private T remove(Node<T> t) {
    T oldValue = null; size--; Node<T> r;
    if (t.left == null || t.right == null) {
      if (t.parent == null) {
        this.root = null; return oldValue;
      } r = t;
    } else {
      r = successor(t);
      t.key = r.key; oldValue = t.value; t.value = r.value;
    } Node<T> p = (r.left != null?r.left:r.right);
```

```
      if (p != null)         p.parent = r.parent;
41    if (r.parent == null) this.root = p;
      else {
          if (r == r.parent.left) r.parent.left = p;
          else                    r.parent.right = p;
          balance(r.parent);
46    } return oldValue;
  }
  private Node<T> successor(Node<T> predec) {
      if (predec.right != null) { Node<T> r = predec.right;
        while (r.left != null) r = r.left;
51      return r;
      } else { Node<T> p = predec.parent;
        while (p != null && predec == p.right) {
          predec = p; p = predec.parent;
        } return p;
56    }
  }
  private void balance(Node<T> node) {
      int balance = node.balance();
      if (balance <= -2) {
61      if (height(node.left.left) >= height(node.left.right))
            node = rotateRight(node);
        else node = rotateLeftRight(node);
      } else if (balance >= 2) {
        if (height(node.right.right) >= height(node.right.left))
66          node = rotateLeft(node);
        else node = rotateRightLeft(node);
      }
      if (node.parent != null) balance(node.parent);
      else                     this.root = node;
71 }
  private static <X> Node<X> rotateLeftRight(Node<X> node) {
      node.left = rotateLeft(node.left); return rotateRight(node);
  }
  private static <X> Node<X> rotateRightLeft(Node<X> node) {
76    node.right = rotateRight(node.right); return rotateLeft(node);
  }
  private static <X> Node<X> rotateRight(final Node<X> r) {
      Node<X> pivot = r.left, left = pivot.left;
      if (r.parent != null) {
81        if (r.parent.left == r) r.parent.left = pivot;
          else                    r.parent.right = pivot;
      }
      pivot.parent = r.parent; r.left = pivot.right;
      if (r.left != null) r.left.parent = r;
86    pivot.right = r; r.parent = pivot;
      return pivot;
  }
  private static <X> Node<X> rotateLeft(final Node<X> r) {
      Node<X> pivot = r.right, right = pivot.right;
91    if (r.parent != null) {
        if (r.parent.left == r) r.parent.left = pivot;
        else                   r.parent.right = pivot;
      }
      pivot.parent = r.parent; r.right = pivot.left;
96    if (r.right != null) r.right.parent = r;
      pivot.left = r; r.parent = pivot;
      return pivot;
  }
```

2.3 Heaps / Priority-Queues

Die wichtigsten Operationen bei Warteschlangen (queues) sind der Zugriff auf das vorderste Objekt, das Einfügen eines Objektes am Ende der Warteschlange und das Entfernen des vordersten Objektes. Solche einfache Warteschlangen werden üblicherweise mit verketteten Listen realisiert.
Die wichtigsten Operationen von Prioritätswarteschlangen unterscheiden sich leicht von denjenigen der einfachen Warteschlangen. Der Zugriff auf das vorderste Objekt wird durch den Zugriff auf ein Objekt mit höchster Priorität ersetzt. Entsprechend wird das Entfernen des vordersten Objektes durch das Entfernen eines Objektes mit höchster Priorität ersetzt. Schliesslich wird ein neues Objekt nicht einfach am Ende eingefügt, sondern es muss an der richtigen Position gemäss seiner Priorität eingereiht werden. Dabei muss die richtige Position aber zuerst gesucht werden.

2.3.1 Bedingungen

Man unterscheidet Heaps in Min-Heaps und Max-Heaps. Bei Min-Heaps bezeichnet man die Eigenschaft, dass die Schlüssel der Kinder eines Knotens stets größer als der Schlüssel ihres Vaters sind, als Heap-Bedingung. Dies bewirkt, dass an der Wurzel des Baumes stets ein Element mit minimalem Schlüssel im Baum zu finden ist. Umgekehrt verlangt die Heap-Bedingung bei Max-Heaps, dass die Schlüssel der Kinder eines Knotens stets kleiner als die ihres Vaters sind. Hier befindet sich an der Wurzel des Baumes immer ein Element mit maximalem Schlüssel.

2.3.2 Element mit höchster Priorität entfernen

Das eigentliche Entfernen eines Objektes mit höchster Priorität entspricht dem Entfernen der Wurzel des Heaps.Dies hinterlässt im Allgemeinen zwei separate Heaps, nämlich den linken und den rechten Teilbaum der Wurzel. Die beiden Heaps werden zusammengeführt in dem der Knoten genommen wird, der auf dem untersten Niveau am weitesten rechts steht, und an der Stelle der Wurzel eingefügt wird.
Dadurch wird aber in der Regel die Heap Bedingungen verletzt. Um dies zu korrigieren lässt man die neue Wurzel versickern (**shift-down**) bis die Bedingungen wieder korrekt sind. Dabei bedeutet ein einzelner Versickerungsschritt das Vertauschen des Schlüssels eines inneren Knotens mit dem grösseren Schlüssel seiner beiden Söhne.
Der Aufwand des Versickerns hängt direkt von der Länge des Pfades und somit von der Höhe des Heap ab. Da die Höhe eines balancierten Binärbaumes logarithmisch in der Anzahl der Knoten *(= n)* ist, resultiert gesamthaft eine logarithmische Zeitkomplexität für das Entfernen eines Schlüssels mit höchster Priorität: *O(log n)*.

2.3.3 Einfügen eines Objektes mit gegebener Priorität

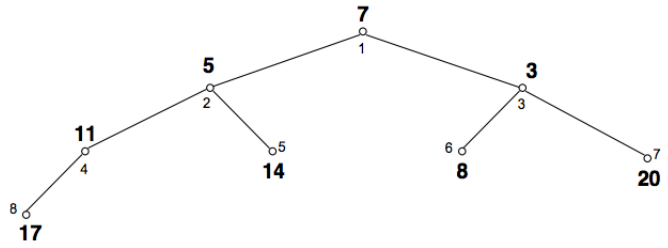
Das neue Objekt wird am letzten Knoten im Heap eingefügt. Dazu werden alle Knoten im Heap fortlaufend nummerriert. Beginnend bei der Wurzel mit 1, dann alle weiteren Niveaus der Reihe nach und innerhalb der Niveaus von links nach rechts. Auch hier ist es sehr warscheinlich, dass die Heap Bedinungen verletzt werden. Man wenden hier das genau gegenteilige Verfahren an (**shift-up**):
Erfüllt Knoten v die Heap-Bedingung nicht, so wird v mit demjenigen Sohn von v, welcher den grösseren Schlüssel besitzt, vertauscht.

2.3.4 Entfernen eines Objektes an beliebiger, aber gegebener Position

Auch bei dieser Operation wird die Nummerierung der Knoten benötigt. Falls der zu entfernende Knoten der letzte Knoten im Heap ist, kann dieser entfernt werden ohne die Heap Bedinungen zu verletzen. Andernfalls wird der zu löschende Knoten v mit dem letzten Knoten p des heaps vertauscht, damit v problemlos gelöscht werden kann.
m Allgemeinen wird durch das Vertauschen der beiden Knoten v und p die Heap-Bedingung verletzt, entweder hat p einen grösseren Schlüssel als sein neuer Vater oder p hat einen kleineren Schlüssel als einer seiner neuen Söhne. Im ersteren Fall wird auf dem Pfad von p zur Wurzel die **sift-up**-Operation angewandt und im letzteren Fall lassen wir p mittels **sift-down**-Operation versickern.

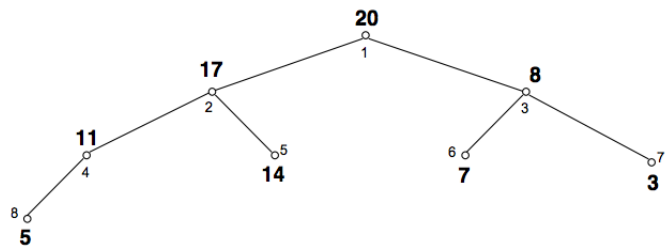
2.3.5 Aufbau eines Heaps

Mit den n Schlüsselwerten bauen wir zuerst einen balancierten Binärbaum auf, so dass alle Blätter auf höchstens zwei verschiedenen Niveaus auftreten und dass auf demjenigen Niveau, wo sowohl innere Knoten als auch Blätter auftreten können, kein innerer Knoten weiter rechts liegt als irgend ein Blatt. Im Allgemeinen wird jedoch die Heap-Bedingung dadurch nicht erfüllt. Bevor wir auf die Umstrukturierung eingehen, zeigen wir den balancierten Binärbaum, welcher aus der Schlüsselmenge 7, 5, 3, 11, 14, 8, 20, 17 anfänglich aufgebaut wird. Die Anzahl innerer Knoten ist n = 8.



Für jeden inneren Knoten, dessen beide Söhne beides Blätter sind, gilt trivialerweise, dass die Heap- Bedingung bereits erfüllt ist. Dies ist der Fall für alle Knoten mit einer Nummer grösser als $(n/2)$. Im oben stehenden Beispiel sind dies die Knoten 5 bis 8. Die restlichen Knoten mit den Nummern 1 bis $(n/2)$ werden nun in der Reihenfolge $(n/2)$, $(n/2) - 1$, ..., 1 mit der Operation **sift-down** versickert. Für oben stehendes Beispiel ergeben sich dadurch die folgenden Veränderungen und schliesslich der unten stehende Heap:

- Knoten 4 mit Schlüssel 11 versickern: 7, 5, 3, 17, 14, 8, 20, 11
- Knoten 3 mit Schlüssel 3 versickern: 7, 5, 20, 17, 14, 8, 3, 11
- Knoten 2 mit Schlüssel 5 versickern: 7, 17, 20, 11, 14, 8, 3, 5
- Knoten 1 mit Schlüssel 7 versickern: 20, 17, 8, 11, 14, 7, 3, 5



3 Hash-Verfahren

Die grosse Aufgabe bei Datenstrukturen sind die drei Operationen Einfügen, Suchen und Entfernen möglichst schnell anzubieten. Das schnellste bisher sind die AVL-Bäume, die alle drei Operationen in $O(log(n))$ anbieten. Im Gegensatz dazu steht das Array, welches alle drei Operationen in $O(1)$ anbietet. Es kann direkt auf einzelne Speicherzellen zugegriffen werden ohne umständliches Vergleichen und/oder Suchen. Arrays sind aber nicht dynamisch und es ist nicht möglich, ein unendlich langes Array zu haben, sondern es muss auf eine fixe Grösse limitiert werden. Die Hash-Datenstruktuen versuchen das schnelle Array mit der Unlimitiertheit der dynamischen Datenstrukturen zu verbinden.

3.1 Begriffe

Quellmenge
Die Menge aller Möglicher Nachrichten.

Zielmenge
Menge aller möglichen Hash-Werten. Sie ist im Allgemeinen viel kleiner als die Quellmenge.

Kollision
Nachrichten welche den selben Hash-Wert haben.

Hash-Funktion
Enthält die Berechnung, die es erlaubt, von einer beliebigen Nachricht einen Hash-Wert fixer Länge zu berechnen.

3.2 Hash-Funktionen

Divisions-Rest-Methode
Ein nahe liegendes Verfahren zur Erzeugung eines Hash-Wertes ist es, den Rest einer ganzzahligen Division von k durch m zu nehmen: $h(k) = k \ mod \ m$.
Für die Qualität dieser Hash-Funktion ist dann allerdings eine geschickte Wahl von m entscheidend. Eine gute Wahl ist eine Primzahl welche kein Teiler einer zweierpotenz ist.

Multiplikative Methode
Der gegebene Schllüsse wird mit einer irrationalen Zahl multipliziert; der ganzzahlige Anteil des Resultats wird abgeschnitten. Auf diese Weise erhält man für verschiedene Schlüssel verschiedene Werte zwischen 0 und 1. Für Schlüssel 1, 2, 3, ..., n sind diese Werte ziemlich gleichmässig im Intervall [0, 1] verstreut.

Jede Hash-Funktion aus H bildet alle denkbar möglichen Schlüsselwerte auf einen Index aus {0, 1, ..., m-1} ab. H heisst nun **universell**, wenn für je zwei verschiedene Schlüsselwerte x und y gilt:

$$\frac{|[h \in H : h(x) = h(y)]|}{|H|} <= \frac{1}{m}$$

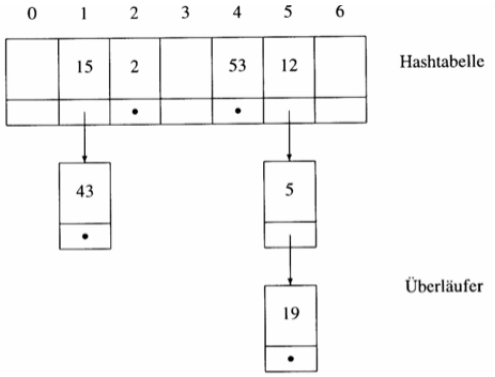
H ist also dann universell, wenn für jedes Paar von verschiedenen Schlüsseln höchstens der m-te Teil der Hash-Funktionen aus H zu einer Indexkollision für dieses Schlüsselpaar führen.

3.3 Verkettung der Überläufer

Das zu lösende Problem sind die Synonyme. Soll in ein Array, das bereits den Schlüssel k enthält, ein Synonym k' von k eingefügt werden, so ergibt sich eine Indexkollision. Der Platz h(k) = h(k') ist bereits besetzt und k', ein Überläufer, muss anderswo gespeichert werden. Eine einfache Art, Überläufer zu speichern, ist die, sie ausserhalb des Arrays abzulegen, und zwar in dynamisch veränderbaren Strukturen. So kann man etwa die Überläufer zu jedem Array-Index in einer linearen Liste verketten; diese Liste wird an den Array-Eintrag angehängt, der sich durch Anwendung der Hash-Funktion auf die Schlüssel ergibt.

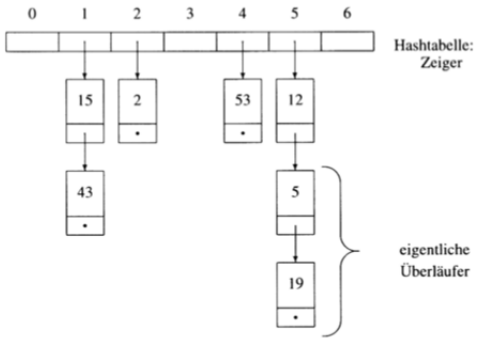
3.3.1 Separate Verkettung

Bei der separaten Verkettung der Überläufer ist jedes Element der Hash-Tabelle das Anfangselement einer Überlaufkette (verkettete lineare Liste). Angenommen wir hätten eine Klasse List mit einer inneren Klasse List.Element. Somit können wir ein Array von solchen Elementen als unsere Hash-Tabelle verwenden.



3.3.2 Direkte Verkettung

Bei der direkten Verkettung der Überläufer ist jedes Element der Hash-Tabelle eine eigenständige Liste. In der Hash-Tabelle werden also bloss Referenzen auf Listen gespeichert und die Datensätze in die Listen eingefügt.



3.4 Offene Hash-Verfahren

Mit der Idee von offenen Hash-Verfahren wird ein Ansatz verfolgt, die Überläufer innerhalb der Hash-Tabelle unterzubringen. Wenn also beim Versuch den Schlüssel k in die Hash- Tabelle an Position h(k) einzutragen festgestellt wird, dass t[h(k)] bereits belegt ist, so muss man nach einer festen Regel einen anderen, nicht belegten Platz finden, an dem man k unterbringen

kann. Da man von vornherein nicht weiss, welche Plätze belegt sein werden und welche nicht, definiert man für jeden Schlüssel eine Reihenfolge, in der alle Speicherplätze einer nach dem anderen betrachtet werden. Sobald dann ein betrachteter Platz frei ist, wird der Datensatz dort gespeichert. Die Magie liegt also darin, wie man abhängig vom jeweiligen Schlüssel, die Hash-Tabelle inspiziert. Diese Reihenfolge nennt sich Sondierungsfolge.
Um einen Schlüssel zu löschen ohne die Sondierungsreienfolge zu zerstören benötigt es einen kleineen Trick. Er wird nicht wirklich entfernt, sondern lediglich als entfernt markiert. Wird ein neuer Schlüssel eingefügt, so wird der Platz von k als frei angesehen; wird ein Schlüssel gesucht, so wird der Platz von k als belegt angesehen.

3.4.1 Schema

Sei s(j,k) eine Funktion von j und k so, dass $(h(k) - s(j,k))\ mod\ m$ für j = 0, 1, ..., m-1 eine Sondierungsfolge bildet, d.h. eine Permutation aller Hash-Adressen. Es sei stets noch mindestens ein Platz in der Hash-Tabelle frei.

3.4.2 Lineares Sondieren

Beim linearen Sondieren ergibt sich für den Schlüssel k die Sondierungsfolge
$h(k), h(k)-1, h(k)-2, h(k)-3, ..., 0, m-1, m-2, m-3, ..., h(k)+1$
Es wird einfach immer ein Array-Index kleiner versucht, bis der kleinste Index erreicht wird (also 0) und dann wird einfach vom höchsten Index an weiter gesucht.
Das Schema ist beim linearen Sondieren die Funktion $s(j,k) = j$.

3.4.3 Double-Hashing

Für die Sondierungsfolge wird eine zweite Hash-Funktion verwendet. Die gewählte Sondierungsfolge für Schlüssel k ist
$h(k), h(k) - h'(k), h(k) - 2 * h'(k), ..., h(k) - (m-1) * h'(k)$
wenn h'(k) die zweite Hash-Funktion bezeichnet. Damit wir keine Indizes errechnen, die kleiner 0 sind, wird das Resultat jeweils noch modulo m gerechnet.

3.4.4 Implementierung

Listing 8: Abstrakte Klasse

```
1  abstract public class OpenHashMap<T> implements HashMap<T> {
     private static enum Zustand { FREI, BELEGT, ENTFERNT };
     private class Element {
       private T m_data;
       private int m_key;
6      private Zustand m_zustand;

   public Element(int key, T data) {
         m_data = data;
         m_key = key;
11       m_zustand = Zustand.FREI;
       }
     }
     private Element[] m_HashTable;
     private int m_n;
16   public OpenHashMap(int size) {
       m_n = 0;
       m_HashTable = new OpenHashMap.Element[size];
```

```
    for (int i = 0; i < size; i++) m_HashTable[i] = new Element(-1, null);
  }
  protected int getTableSize() { return m_HashTable.length; }
  private Element find(int key) {
    int j = 0, i, hashValue = h(key);
    do {
      i = ((hashValue - s(j, key))%getTableSize() + getTableSize())%
          getTableSize();
      j++;
    } while (m_HashTable[i].m_zustand != Zustand.FREI && m_HashTable[i].
        m_key != key);
    if (m_HashTable[i].m_zustand == Zustand.BELEGT) {
      assert m_HashTable[i].m_key == key;
      return m_HashTable[i];
    } else return null;
  }
  abstract int h(int key);
  abstract int s(int j, int key);
}
```

Listing 9: Besipiel

```
public class LineareHashMap<T> extends OpenHashMap<T> {
  public LineareHashMap(int size) { super(size); }
  @Override
  int h(int key) {
    // TODO
  }
  @Override
  int s(int j, int key) {
    // TODO
  }
}
```

4 Graphen