

Algorithmen & Datenstrukturen 2

Jan Fässler

3. Semester (HS 2012)

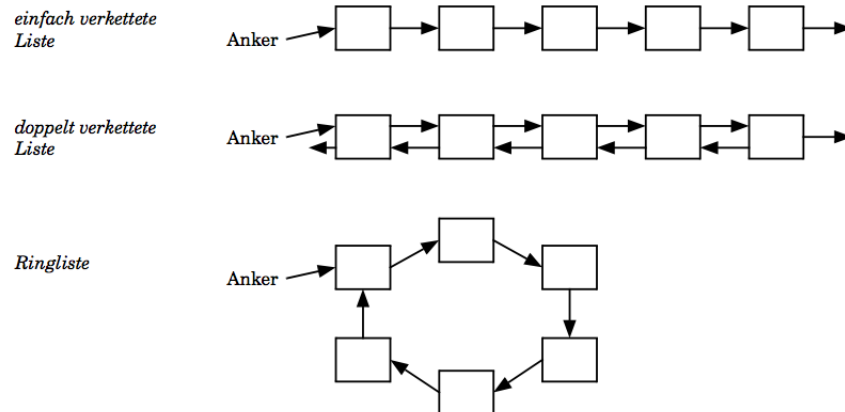
Inhaltsverzeichnis

1	Listen	1
1.1	Stack	2
1.2	Erweiterte Liste	2
1.2.1	Iterators	4
1.2.2	Merge Sort	5
1.3	Skip-Liste	5
1.3.1	Beispiel	5
2	Bäume	7
2.1	Binäre Suchbäume	7
2.1.1	Traversieren	7
2.2	Balancierte Bäume	8
2.2.1	Berechnungen	8
2.2.2	Einfügen	8
2.2.3	Löschen	9
2.2.4	Beispiel	9
2.3	Heaps / Priority-Queues	11
2.3.1	Bedingungen	11
2.3.2	Element mit höchster Priorität entfernen	11
2.3.3	Einfügen eines Objektes mit gegebener Priorität	11
2.3.4	Entfernen eines Objektes an beliebiger, aber gegebener Position	12
2.3.5	Aufbau eines Heaps	12
2.3.6	Sortierung	13
3	Hash-Verfahren	14
3.1	Begriffe	14
3.2	Hash-Funktionen	14
3.3	Verkettung der Überläufer	15
3.3.1	Separate Verkettung	15
3.3.2	Direkte Verkettung	15
3.4	Offene Hash-Verfahren	15
3.4.1	Schema	16
3.4.2	Lineares Sondieren	16
3.4.3	Double-Hashing	16
3.4.4	Implementierung	16
4	Graphen	18
4.1	Definitionen	18
4.2	Einfache Graphen	19
4.2.1	Vollständige Graphen	19
4.2.2	Kreise	19
4.2.3	Bipartite Graphen	19
4.2.4	Hyperwürfel	19
4.2.5	Planare Graphen	20
4.3	Gerichtete Graphen (Digraphen)	20
4.4	Speicherung von Graphen	21
4.4.1	Adjazenzmatrix	21
4.4.2	Inzidenzmatrix	21

4.4.3	Adjazenzlisten	22
4.4.4	doppelt verketteten Listen	22
4.5	Graphenalgorithmen	23
4.5.1	Topologisches Sortieren	23
4.5.2	Tiefensuche (DFS)	24
4.5.3	Breitensuche (BFS)	25
4.5.4	Kürzeste Pfade mit Dijkstra	25
4.5.5	Kürzeste Pfade mit Bellman und Ford	27
4.6	Spannbäume	28
4.6.1	Definitionen	28
4.6.2	Algorithmus von Prim	28
4.6.3	Beispiel des Prim Algorithmus	29
5	Spieltheorie	30
5.1	Minimax-Algorithmus	30
5.1.1	Beschreibung	30
5.1.2	Implementierung	30
5.2	Alpha-Beta-Suche	32
5.2.1	Beschreibung	32
5.2.2	Der Algorithmus	32
5.2.3	Implementierung	33

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 Datenstruktur 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() {
3      return this.getLast();
    }
    public void push(T data) {
        this.add(data);
    }
8   public T pop() {
        T last = this.getLast();
        this.remove(last);
        return last;
    }
13  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;
4   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;
9       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); }
14  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;
19          if (size == 0) {
                head = newElement;
                foot = head;
            } else if (index == size) {
                newElement.last = foot;
24          foot.next = newElement;
                foot = newElement;
            } else if (index == 0) {
```

```

        newElement.next = current;
        current.last = newElement;
29     head = newElement;
    } else {
        for (int i=0; i<index; i++) current = current.next;
        newElement.next = current;
        newElement.last = current.last;
34     if (current.last != null) current.last.next = newElement;
        current.last = newElement;
    }
    size++;
}
39 }

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.");
44 }

public T remove(int index) {
    if (index >= size) throw new IndexOutOfBoundsException();
    T element = get(index);
    if (index == 0) {
49     if (size > 1) head = head.next;
        else { head = null; foot = null; }
    } else if (index == (size-1)) {
        foot.last.next = null;
        foot = foot.last;
54     } 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;
59     current = null;
    }
    return element;
}

public int indexOf(T data) {
64     int index = -1;
    if (this.contains(data)) { index++; while(!this.get(index).equals(data))
        index++; }
    return index;
}

public T get(int index) {
69     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>
{
74     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); }
79 }
}

```

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 rückwärtsiterieren 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();
5    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;
10    private AdvancedComparableList<E> _list;
    private int index = 0;
    public CListIterator(ListElement<E> element, AdvancedComparableList<E>
        list) {
        _list = list;
        nextElement = element;
15    prevElement = (element == null?null:element.last);
        ListElement<E> current = element;
        while (current != null && current.last != null) {
            index++;
            current = current.last;
20    }
    }
    public boolean hasNext() { return nextElement != null; }
    public E next() {
        index++;
25    prevElement = nextElement;
        nextElement = nextElement.next;
        lastReturned = prevElement;
        return prevElement.data;
    }
30    public boolean hasPrevious() { return prevElement != null; }
    public E previous() {
        index--;
        nextElement = prevElement;
        prevElement = prevElement.last;
35    lastReturned = nextElement;
        return nextElement.data;
    }
    public int nextIndex() { return index; }
    public int previousIndex() { return index-1; }
40    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

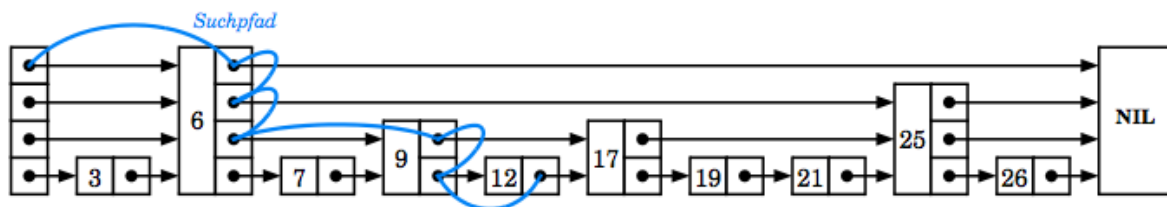
Listing 5: Merge Sort

```
/*      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;
               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 SkipList {
```



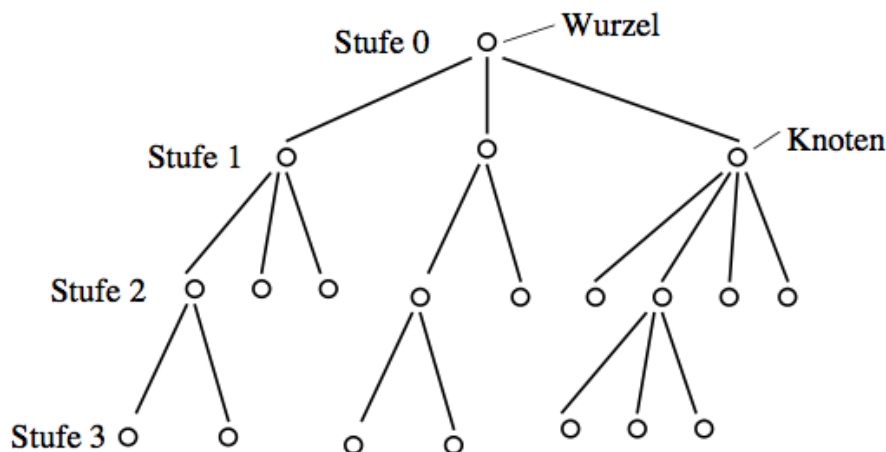
```

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
6 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);
11 }
    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);
16        }
        aktuell = aktuell.getNext(0);
        if (aktuell.getKey() == key) return aktuell.getData();
        else return Integer.MAX_VALUE;
    }
21 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);
26        update[i] = aktuell;
    }
    aktuell = aktuell.getNext(0);
    if (aktuell.getKey() == key) aktuell.setData(object);
    else {
31        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);
36        }
    }
}
    public void entfernen(int key) {
        Element[] update = new Element[m_maxLevel + 1];
        Element aktuell = m_headAnchor;
41        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);
46        for (int i = 0; i <= aktuell.getLevel(); i++) update[i].setNext(i,
            aktuell.getNext(i));
        aktuell = null;
    }
    public int randomLevel() {
        int level = 0;
51        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, \dots, 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 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: $bal(p) = \pm 1$

Schiefelage: $++h(p)$

Variante a.)

$bal(v) \neq 0$ / p ist kürzerer Ast $\Rightarrow bal(v) = 0$

& $h(v) = const \Rightarrow$ **fertig**

Variante b.)

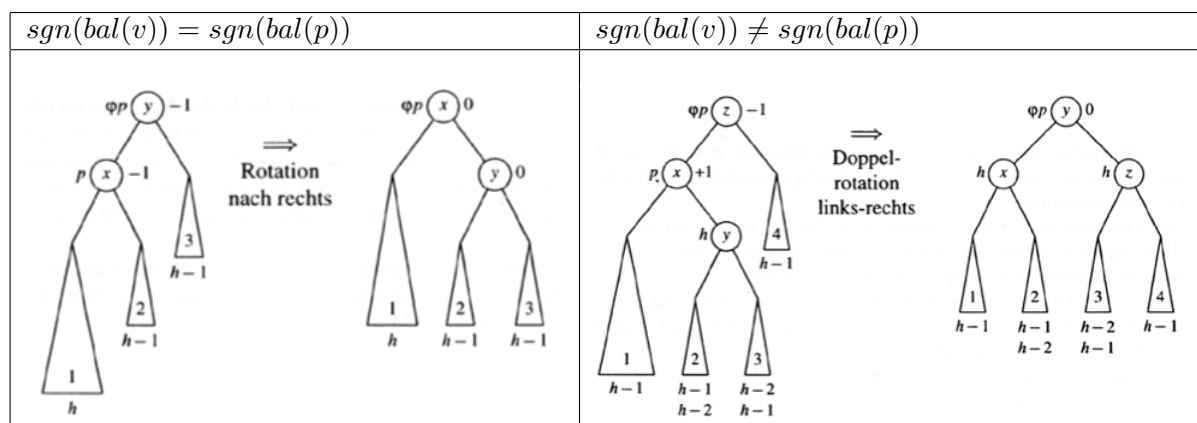
$bal(v) = 0 \Rightarrow bal(v) = \pm 1$

& $++h(v) \Rightarrow$ **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:**



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

```
1 public T insert(int key, T value) {
    final Node<T> n = new Node<T>(key, value, null);
    return insert(this.root, n);
}
2 public T insert(Node<T> r, Node<T> n) {
6     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);
11            } 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);
16            } else return insert(r.right, n);
        } else {
            T oldValue = r.value; r.value = n.value; return oldValue;
        }
    } return null;
}
21 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);
26            else if (t.key < key) return remove(key, t.right);
            else return remove(t);
        }
    }
    private T remove(Node<T> t) {
31        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;
36        } 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 nummeriert. 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 wahrscheinlich, dass die Heap Bedingungen verletzt werden. Man wendet 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.

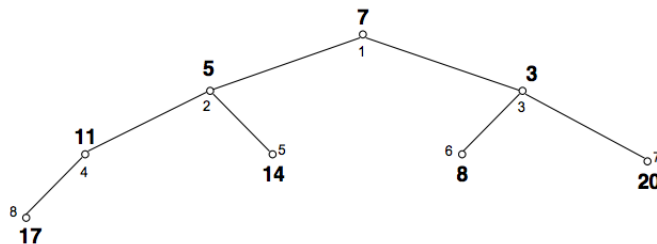
2.3.4 Entfernen eines Objektes an beliebiger, aber gegebener Position

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 Bedingungen zu verletzen. Andernfalls wird der zu löschende Knoten v mit dem letzten Knoten p des heaps vertauscht, damit v problemlos gelöscht werden kann.

Im 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.

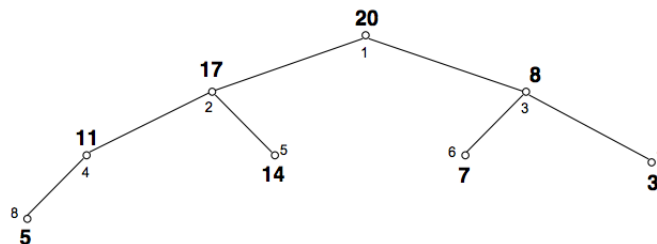
2.3.5 Aufbau eines Heaps

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



2.3.6 Sortierung

Listing 8: Sortierung

```
1 public class HeapSort {
    static int counter = 0;
    public static void main(String[] args) {
        int[] array = {3, 5, 1, 8, 2, 4, 7, 6, 10, 12, 1337};
        heapsort(array);
6    }
    private static int[] heapsort(int[] a) {
        int n = a.length;
        // bring the array into heap form
        for(int i = n/2-1; i >= 0; i--) siftDown(a, i, n-1);
11    // sort the heap
        for(int i = n-1; i >= 1; i --) {
            swap(a, 0, i);
            siftDown(a, 0, i-1);
        }
16    return a;
    }
    private static void swap(int[] a, int i, int j) {
        int temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
21    private static void siftDown(int[] a, int i, int m) {
        int j;
        while(2*i < m) {
            j = 2*i;
            if(j < m) {
26                // check if right is bigger
                if(a[j] < a[j+1]) j++;
                // check if the element has to be swapped
                if(a[i] < a[j]) { swap(a, i, j); i = j; }
                else { i = m; }
31            }
        }
    }
}
```

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-Datenstrukturen 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: $h(k) = k \bmod m$.

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 Schlüssel 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.

$$h(k) = \lfloor (kA \bmod 1) \times m \rfloor = \lfloor (kA - \lfloor kA \rfloor) \times m \rfloor, \quad 0 < A < 1, \quad m = \text{Anz. Adressen}$$

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

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

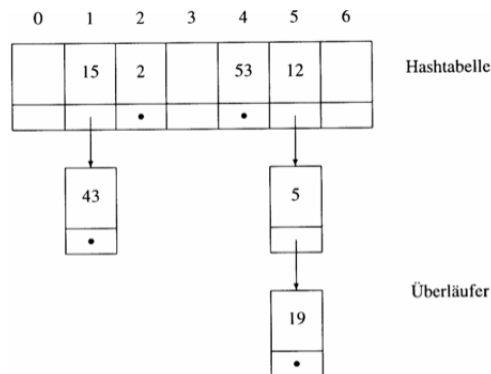
H ist also dann universell, wenn für jedes Paar von verschiedenen Schlüsseln höchstens der m -te Teil der Hash-Funktionen aus H zu einer Indexkollision für dieses Schlüsselpaar führen.

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 verketteten; diese Liste wird an den Array-Eintrag angehängt, der sich durch Anwendung der Hash-Funktion auf die Schlüssel ergibt.

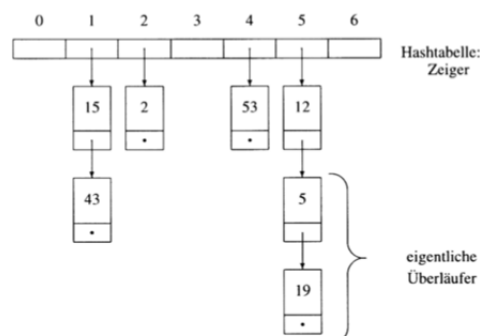
3.3.1 Separate Verkettung

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 Sondierungsreihenfolge zu zerstören benötigt es einen kleinen 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 $(h(k) - s(j,k)) \bmod m$ für $j = 0, 1, \dots, m-1$ 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

$h(k), h(k) - 1, h(k) - 2, h(k) - 3, \dots, 0, m - 1, m - 2, m - 3, \dots, h(k) + 1$

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

$h(k), h(k) - h'(k), h(k) - 2 * h'(k), \dots, h(k) - (m - 1) * h'(k)$

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 9: 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);
    }
21 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();
26         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];
31     } else return null;
    }
    abstract int h(int key);
    abstract int s(int j, int key);
}

```

Listing 10: Beispiel

```

public class LineareHashMap<T> extends OpenHashMap<T> {
    public LineareHashMap(int size) { super(size); }
    @Override
    int h(int key) {
5        // TODO
    }
    @Override
    int s(int j, int key) {
        // TODO
10    }
}

```

4 Graphen

4.1 Definitionen

Graph

Ein Graph besteht aus einer Knotenmenge V (vertices) und einer Kantenmenge E (edges). Dabei gilt:

- i $|V| = n > 0$
- ii $E = \{(u, v) | u, v \in V\}$ wobei u und v nicht verschieden sein müssen. Eine Kante ist also ein Paar von Knoten. Falls $u = v$ gilt, dann sprechen wir von einer Schleife oder Schlinge, also einer Kante, welche einen Knoten mit sich selber verbindet.
 $|E| = m \geq 0$

Notation

$V = \{1, 2, \dots, n\}$ und $E = \{(1, 2), (3, 2), \dots, (2, 4)\}$

Knoten

Der Grad eines Knoten in einem Graphen entspricht der Menge der Kanten, welche den Knoten als (einen) Anfangs- bzw. Endpunkt haben. Der Grad wird angegeben als **deg(v)**. Eine Kante ist zu ihren Anfangs- und Endknoten **inzident**. Alle Knoten, mit denen ein Knoten durch eine Kante verbunden ist, heissen **adjazente** Knoten. Diese adjazenten Knoten werden meist auch als Nachbarn bezeichnet.

Kanten

Eine Kante verbindet zwei Knoten miteinander. Dabei kann sie eine Richtung haben (**gerichtet**), oder aber auch **ungerichtet** sein. Wenn sie eine Richtung hat, wird sie als Pfeil gezeichnet und als Paar von Knoten (u, v) symbolisiert, wobei u der Startknoten und v der Zielknoten ist. Eine ungerichtete Kante wird als Verbindungslinie gezeichnet und als Menge u, v symbolisiert.

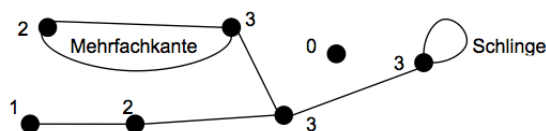
Neben der Richtung kann eine Kante auch noch ein Gewicht haben. Solche Kanten heissen **gewichtete** Kanten. Gewichte stellen zum Beispiel die Entfernung zweier Knoten oder die Bandbreite einer Internetleitung dar.

Sind zwei Knoten durch mehrere direkte Kanten verbunden, wird diese Gruppe als Mehrfachkante zusammengefasst.

Ist ein Knoten mit sich selber durch eine Kante verbunden, wird diese Kante als Schleife bezeichnet. (Für den Grad eines Knoten wird diese Kante zweimal gezählt.)

Zusammenhängende Graphen

Ein Graph ist zusammenhängend, wenn von jedem Knoten jeder andere Knoten über Kanten erreicht werden kann. Ist ein Graph nicht zusammenhängend, besteht er aus mehreren Komponenten. Das Tramnetz einer Stadt ist üblicherweise ein zusammenhängender Graph. Alle Tramnetze der Schweiz gemeinsam betrachtet, sind jedoch ein nicht zusammenhängender Graph.



Baum

Ein Baum ist ein spezieller zusammenhängender Graph. Er zeichnet sich dadurch aus, dass er keine Kreise und somit auch keine Schlingen enthält (siehe Unterkapitel Kreise).

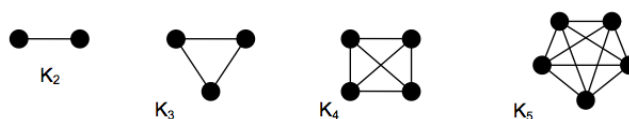
4.2 Einfache Graphen

Ein einfacher Graph ist ein ungerichteter Graph ohne Mehrfachkanten und ohne Schlingen. Dabei gilt:

- Die Summe aller Knotengrade ist gerade (Zweifachen der Anzahl Kanten m).
- Die Anzahl der Knoten mit ungeradem Knotengrad ist gerade.

4.2.1 Vollständige Graphen

Ein vollständiger Graph ist ein einfacher Graph. Er hat aber die zusätzliche Eigenschaft, dass von jedem Knoten aus alle andern Knoten direkt mit einer Kante verbunden sind. Solche Graphen können für die Darstellung eines Turniers verwendet werden, bei dem jedes Team gegen jedes andere Team spielen muss.



4.2.2 Kreise

Ein Kreis ist ein einfacher Graph, bei dem alle Knoten den Grad 2 haben. Von jedem Knoten kommt man auf zwei Wegen zu jedem anderen Knoten.



4.2.3 Bipartite Graphen

Ein einfacher Graph heisst **bipartit**, wenn die Knotenmenge in zwei disjunkte Teilmengen aufgeteilt werden kann und wenn alle Kanten nur Verbindungen zwischen den beiden Teilmengen herstellen. Es gibt also keine Kanten zwischen zwei Knoten aus der gleichen Teilmenge.

Ein bipartiter Graph heisst **vollständig**, wenn jeder Knoten aus der einen Teilmenge mit allen Knoten aus der andern Teilmenge verbunden ist. Wir bezeichnen einen vollständigen bipartiten Graph mit $(K_{a,b})$.



4.2.4 Hyperwürfel

Eine etwas spezielle Art eines Graphen ist der Hyperwürfel. Dieser Graph ist wiederum ein einfacher Graph. Seine Knoten haben jedoch ganz bestimmte Namen. Ein Hyperwürfel der Dimension d hat die Knotenmenge, welche aus allen binären Folgen der Länge d besteht. Formal heisst dies: $V = \{\{0,1\}^d\}$. Eine Kante verbindet immer genau dann zwei Knoten, wenn sich deren Binärfolge in nur einer Stelle unterscheiden. Z.B. werden die zwei Knoten mit den Binärfolgen 1011 und 1001 miteinander verbunden. Solche Hyperwürfel werden zum Beispiel in der Codierungstheorie verwendet.

4.2.5 Planare Graphen

Planare Graphen sind einfache Graphen, welche so gezeichnet werden können, dass sich keine zwei Kanten überschneiden.

Bäume sind planar.

Durch ihre spezielle Struktur können alle Bäume so gezeichnet werden, dass sich keine zwei Kanten überschneiden.

Eulersche Polyederformel

Sei G ein planarer, zusammenhängender, einfacher Graph mit n Knoten, m Kanten und f Gebieten. Dann gilt: $n - m + f = 2$.

Das Verhältnis von Gebieten zu Kanten

$$3f \leq 2m$$

Ein Gebiet wird von mindestens drei Kanten begrenzt. Eine Kante begrenzt immer zwei Gebiete. $3f$ bezeichnet alle Randkanten. Dabei ist aber jede Kante doppelt gezählt.

Kanten und Regionen in planaren Graphen

Für jeden einfachen planaren Graphen $G = (V, E)$ mit $|V| \geq 3$ gilt:
 $m \leq 3n - 6$ und $f \leq 2n - 4$

K_5 und $K_{3,3}$ sind nicht planar

K_5 oder $K_{3,3}$ sind die beiden kleinsten nicht planaren Graphen, was direkt aus dem Satz von Kuratowski folgt. Der Satz von Kuratowski sagt: *„Ein endlicher Graph ist genau dann planar, wenn er keinen Teilgraphen enthält, der durch Unterteilung von K_5 oder $K_{3,3}$ entstanden ist. Unterteilung bedeutet hier das beliebig oft wiederholbare (auch nullmalige) Einfügen von neuen Knoten auf Kanten. Mit Teilgraph ist hier ein Graph gemeint, der aus dem ursprünglichen Graphen durch Entfernen von Knoten bzw. Kanten entsteht.“* Somit erlaubt der Satz von Kuratowski zu entscheiden, ob ein Graph planar ist oder nicht.

4.3 Gerichtete Graphen (Digraphen)

Im Unterschied zu den allgemeinen Graphen besitzen die Kanten gerichteter Graphen eine Richtung. Das bedeutet, dass eine Kante zwischen zwei Knoten nur in einer Richtung durchlaufen werden darf.

Gerichtete Graphen finden ihre Anwendung in ganz verschiedenen Gebieten. In einem Projektplan werden die Aktivitäten mit einer gerichteten Kante verbunden um anzuzeigen, wie der zeitliche Ablauf sein muss. In einer Strassenkarte werden Einbahnstrassen mit einer Richtung versehen.

Eingangsgrad

Der Eingangsgrad (Indegree) eines Knoten entspricht der Anzahl gerichteter Kanten, die in den Knoten hineinführen: **$\text{indeg}(v)$**

Ausgangsgrad

Der Ausgangsgrad (Outdegree) eines Knoten entspricht der Anzahl gerichteten Kanten, die aus dem Knoten hinausführen: **$\text{outdeg}(v)$**

Die Summe der Eingangsgrade aller Knoten ist gleich der Summe der Ausgangsgrade aller Knoten: $\sum \text{indeg}(v) = \sum \text{outdeg}(v)$

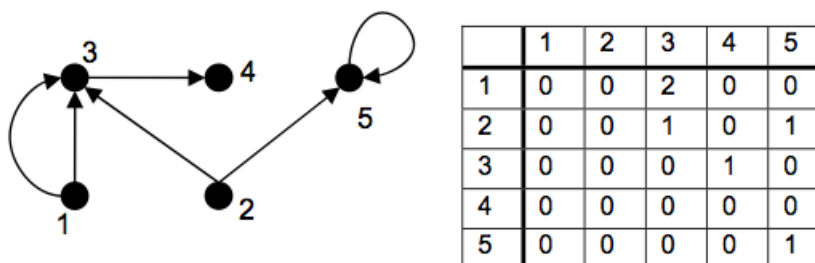
4.4 Speicherung von Graphen

Die Art der Speicherung der Graphen kann auf die Effizienz der auszuführenden Algorithmen einen Einfluss haben. Das ist eigentlich nichts Neues, wie Sie schon längst bei anderen Datenstrukturen gesehen haben. Aus diesem Grund ist es wichtig, Graphen speicher- und laufzeiteffizient abzuspeichern.

4.4.1 Adjazenzmatrix

Die Adjazenzmatrix ist eine Matrix der Größe $n \times n$, wobei n die Anzahl der Knoten bezeichnet. Sind zwei Knoten (u & v) mit einer gerichteten Kante verbunden wird in der u -ten Zeile und der v -ten Spalte eine 1 eingetragen. Sind die beiden Knoten mit mehr als einer Kante verbunden, wird die Anzahl der Kanten eingetragen. Für u und v gilt $1 \leq u \leq n$, $1 \leq v \leq n$.

Wenn der Graph gewichtet ist, dann wird das Gewicht der Kante in die Zelle eingetragen. Bei gewichteten Graphen ist es unüblich, dass Mehrfachkanten auftreten. Es kann nicht unterschieden werden, ob die Kanten gewichtet sind oder ob Mehrfachkanten aufgetreten sind.



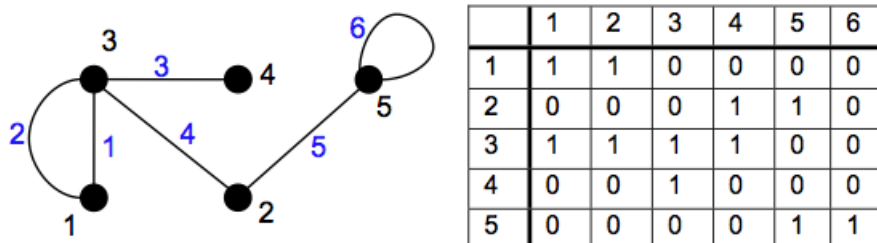
$$G = (V, E) \text{ mit } V = \{1, 2, 3, 4, 5\} \text{ und } E = \{(1, 3), (1, 3), (3, 4), (2, 3), (2, 5), (5, 5)\}$$

4.4.2 Inzidenzmatrix

Die Inzidenzmatrix ist eine $(n \times m)$ -Matrix. Dabei werden nicht nur die Knoten nummeriert, sondern auch die Kanten. Gehört eine Kante e mit Index j zu einem Knoten v , wird in der v -ten Zeile und der j -ten Spalte eine 1 eingetragen. Ansonsten wird eine 0 eingetragen. Für v und j gelten $1 \leq v \leq n$, $1 \leq j \leq m$.

Für gerichtete Graphen kann die Darstellung nicht einfach übernommen werden. Man muss definieren können, ob ein inzidenter Knoten der Ursprung oder das Ziel ist. Man könnte z.B. für Ursprung eine 1 eintragen und für Ziel eine -1.

Gewichte der Kanten können in der Inzidenzmatrix berücksichtigt werden indem der Wert des Gewichts in der Matrix eingetragen wird.



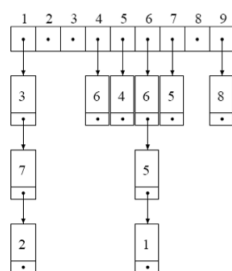
$$G = (V, E) \text{ mit } V = \{1, 2, 3, 4, 5\} \text{ und } E = \{\{1, 3\}, \{1, 3\}, \{3, 4\}, \{2, 3\}, \{2, 5\}, \{5\}\}$$

4.4.3 Adjazenzlisten

Wie bei der Adjazenzmatrix werden bei dieser Speicherung die Verbindungen zwischen zwei Knoten in den Vordergrund gestellt. Es wird also gespeichert, welcher Knoten mit welchem anderen Knoten benachbart ist.

Die Grundlage der Speicherung ist ein Knoten-Array. Dieses Array hat so viele Felder, wie es Knoten gibt. Jedem Knoten wird eine Position im Array zugeteilt. Im Knoten-Array werden mindestens die Anfangszeiger auf verkettete Listen gespeichert. In diesen Listen sind die Knoten gespeichert, zu welchen eine direkte Verbindung durch eine Kante besteht. Die Kante (u, v) aus dem Graph führt zu einem Eintrag in der Liste an der Stelle u im Array.

Diese Speicherung benötigt $\Theta(n + m)$ Speicherplatz. Adjazenzlisten unterstützen viele Operationen, z.B. das Verfolgen von gerichteten Kanten in Graphen. Andere Operationen dagegen werden nur schlecht unterstützt, insbesondere das Hinzufügen und Entfernen von Knoten

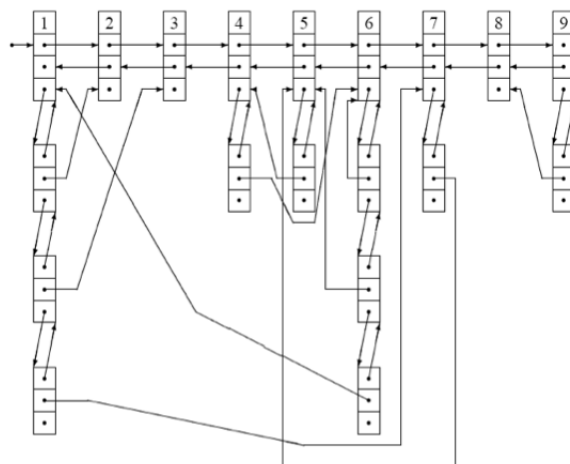


$$G = (V, E) \text{ mit } V = \{1, 2, \dots, 9\} \text{ und}$$

$$E = \{(1, 2), (1, 3), (1, 7), (4, 6), (5, 4), (6, 1), (6, 5), (6, 6), (7, 5), (9, 8)\}$$

4.4.4 doppelt verketteten Listen

Bei dieser Speicherung werden die Knoten und die Kanten als Objekte behandelt. Es wird ihnen also zugestanden, mehr Information als nur den Namen zu enthalten. Die Basis bildet eine doppelt verkettete Liste, welche die Knotenobjekte enthält. Jedes Knotenobjekt speichert dann eine Liste mit den von ihm ausgehenden Kanten-Objekten. Auch diese Liste ist doppelt verkettet. Ein Kantenobjekt enthält drei Referenzen. Zwei Referenzen werden für die doppelt verkettete Kantenliste benötigt. Die dritte Referenz zeigt auf den Knoten, auf den die gerichtete Kante zeigt. Zusätzlich können im Kantenobjekt noch weitere Informationen wie z.B. das Gewicht der Kante gespeichert werden.



$$G = (V, E) \text{ mit } V = \{1, 2, \dots, 9\} \text{ und}$$

$$E = \{(1, 2), (1, 3), (1, 7), (4, 6), (5, 4), (6, 1), (6, 5), (6, 6), (7, 5), (9, 8)\}$$

4.5 Graphenalgorithmen

4.5.1 Topologisches Sortieren

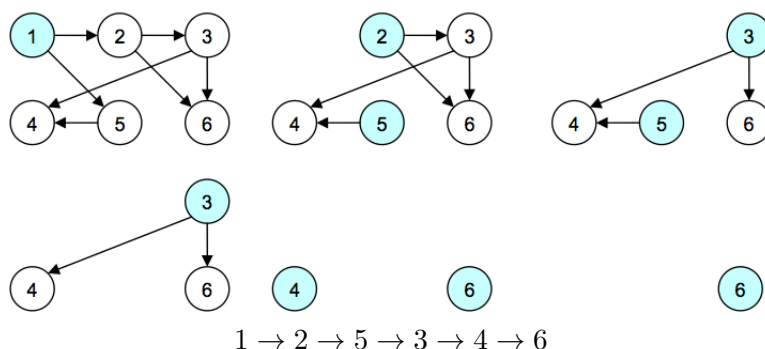
Eine topologische Sortierung basiert auf einer vergleichenden Relation. In einem gerichteten Graphen besteht durch die Kantenrichtung eine solche vergleichende Relation zwischen je zwei benachbarten Knoten. Die Kantenrichtung kann dabei gelesen werden wie zum Beispiel: "grösser als". In ungerichteten Graphen besteht diese vergleichende Relation nicht und daher kann keine topologische Sortierung erreicht werden.

Ein Zyklus oder Kreis ist eine Folge von Kanten bei der Anfangs- und Endpunkt dieselben sind. Eine Schlinge ist der kleinstmögliche Zyklus mit genau einem Knoten. Nicht für alle Digraphen kann eine topologische Sortierung erzeugt werden. Die zentrale Eigenschaft von topologisch sortierbaren Digraphen ist, dass sie keine Zyklen enthalten, also zyklensfrei sind.

Satz: Jeder zyklensfreie Graph hat eine topologische Sortierung.

Die Idee des Algorithmus ist es, mit Knoten zu arbeiten, welche keine eingehenden Kanten haben, d.h. die $\text{indeg}(v) = 0$ besitzen. Diese Knoten können nicht zu einem Zyklus gehören, da sie von keinem anderen Knoten erreicht werden können. In einem topologisch sortierbaren Graphen muss es mindestens einen Knoten v mit $\text{indeg}(v) = 0$ geben. Dieser Knoten wird markiert und bildet den Anfang der topologischen Sortierung. Alle markierten Knoten werden konzeptionell der Reihe nach aus dem Graphen entfernt. Dadurch erniedrigt sich der Eingangsgrad anderer Knoten und es gibt wieder einen oder mehrere neue Knoten mit $\text{indeg}(v) = 0$.

Beispiel:



Listing 11: TopologicalSort

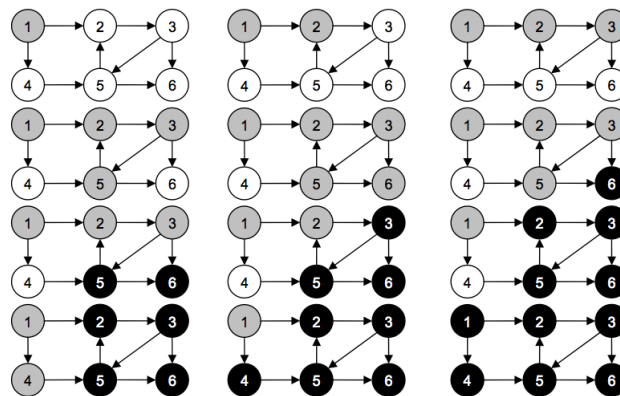
```
private void topologicalSort(HashSet<Knoten> knoten) {
    private Stack<Knoten> knotenMitIndegNull = new Stack<Knoten>();
    Iterator<Knoten> it = knoten.it();
4    while (it.hasNext()) {
        Knoten current = it.next();
        if (current.getIndeg() == 0) knotenMitIndegNull.add(current);
    }
    for(int i = 0; i < knoten.size(); i++) {
9        if(knotenMitIndegNull.size() == 0) break; // keine Sortierung
        Knoten u = knotenMitIndegNull.get(0);
        System.out.print(u.getName() + "->");
        int anzInzidenzeKnoten = u.getInzidenteKnoten().size();
        for(int k = 0; k < anzInzidenzeKnoten; k++) {
14            Knoten v = u.getInzidenteKnoten().get(0);
            if(v.getIndeg() == 0) knotenMitIndegNull.add(v);
            u.removeInzidenterKnoten(v);
        } knotenMitIndegNull.remove(u);
    }
19 }
```

4.5.2 Tiefensuche (DFS)

Die DFS-Strategie ist eine Verallgemeinerung des Preorder-Durchlaufprinzips bei Binärbäumen. Das Prinzip ist das folgende:

- Kanten werden ausgehend von dem zuletzt entdeckten Knoten v , der mit noch unerforschten Kanten inzident ist, erforscht.
- Erreicht man von v aus einen noch nicht erforschten Knoten w , so verfährt man mit w genauso wie mit v .
- Wenn alle mit w inzidenten Kanten erforscht sind, erfolgt ein Backtracking zu v .

Eine Methode ist es drei Arten von Knoten zu haben. Am Anfang ist jeder Knoten weiss. Das bedeutet, dass er noch nicht entdeckt worden ist. Sobald ein Knoten entdeckt worden ist, wird er grau. Wenn ein Knoten komplett abgearbeitet ist, wird er schwarz gefärbt. Dies ist der Fall, wenn er das Backtracking zum Vorgängerknoten einleitet. Des Weiteren wird für jeden Knoten noch abgespeichert, wer sein Vorgänger ist. Der Vorgänger wird in einem Array α gespeichert. Durch dieses Verfahren werden alle Zusammenhangskomponenten gefunden. Jeder Knoten und jede Kante werden einmal besucht. Am Ende enthält α die Informationen eines Spannbaums für den Graphen, falls dieser zusammenhängend ist.



Listing 12: Tiefensuche

```
public class Tiefensuche {
    private static Stack<Knoten> knotenMitIndegNull = new Stack<Knoten>();
    private static void tiefensuche(ArrayList<Knoten> knoten) {
        for(int i = 0; i < knoten.size(); i++)
5         if(knoten.get(i).getColor()==Farbe.WEISS) visitKnoten(knoten.get(i));
    }
    private static void visitKnoten(Knoten v) {
        v.setColor(Farbe.GRAU);
        int anzInzidenzeKnoten = v.getInzidenteKnoten().size();
10        for(int i = 0; i < anzInzidenzeKnoten; i++) {
            Knoten w = v.getInzidenteKnoten().get(i);
            if(w.getColor() == Farbe.WEISS) {
                w.setVorgaenger(v);
                visitKnoten(w);
15        }
    }
    v.setColor(Farbe.SCHWARZ); System.out.print(" -> " + v.getName());
}
}
```

4.5.3 Breitensuche (BFS)

Der Algorithmus für die Breitensuche lautet wie folgt:

1. Für einen Knoten werden die noch nicht besuchten Nachbarknoten gesucht.
2. Jeder Nachbarknoten wird ans Ende einer Warteschlange eingefügt.
3. Solange die Warteschlange nicht leer ist, wähle den nächsten Knoten und beginne wieder bei 1.

Listing 13: Breitensuche

```
1 public class Breitensuche {
    private Queue<Knoten> nachbarknoten = new LinkedList<Knoten>();
    private void breitensuche(ArrayList<Knoten> knoten) {
        Knoten start = knoten.get(0);
        start.setFound(true);
6        nachbarknoten.add(start);

        // Solange Warteschlange nicht leer
        while(nachbarknoten.size() > 0) {
            Knoten u = nachbarknoten.poll();
11
            for(int i = 0; i < u.getInzidenteKnoten().size(); i++) {
                Knoten v = u.getInzidenteKnoten().get(i);
                if(!v.isFound()) {
                    System.out.print(" -> " + v.getName());
16                    v.setFound(true);
                    nachbarknoten.add(v);
                }
            }
        }
21    }
}
```

4.5.4 Kürzeste Pfade mit Dijkstra

Der Algorithmus von Dijkstra arbeitet mit einer *Wellenfront-Strategie*. Der Algorithmus von Dijkstra setzt nicht negative Kantengewichte voraus. Für alle Knoten hinter der Front ist der endgültige Distanzwert bereits ermittelt worden. Wir nennen diese Knoten permanent und speichern sie ab. Für diese Knoten v gilt also: $dist(s, v) = d[v]$. Die Knoten vor der Front sind noch nicht bearbeitet worden. Für alle Knoten auf der Front ist die Berechnung in Gange. Diese Knoten werden in einer Prioritätswarteschlange PQ gespeichert. Die Prioritäten entsprechen den momentanen Abständen zum Startknoten.

Anfangszustand: Am Anfang ist die Menge der permanenten Knoten leer. Die Menge der Knoten auf der Front enthält nur den Startknoten s .

- Nimm einen Knoten u aus PQ, welcher den kleinsten Abstand zu s hat.
- Für alle Nachbarknoten v von u überprüfe die Distanz. Wenn nötig wird diese Distanz aktualisiert. Ist der Nachbarknoten v schon in PQ, wird seine Priorität dort aktualisiert. Ist er noch nicht in dieser Menge, wird er mit dem aktuellen Wert der Distanz in diese Menge eingefügt.
- Nimm den Knoten u in die Menge der permanenten Knoten auf.

Listing 14: Dijkstra

```

public class Dijkstra {
    private static List<Knoten> PERM;
3   public static final int INFINITE_DISTANCE = Integer.MAX_VALUE;
    private static Knoten startKnoten;
    private static PriorityQueue<Knoten> PQ;
    private final static Comparator<Knoten> shortestDistanceComparator = ;
    private static void startDijkstra(ArrayList<Knoten> knoten) {
8       PERM = new ArrayList<Knoten>();
        PQ = new PriorityQueue<Knoten>(8, new Comparator<Knoten>() {
            public int compare(Knoten left, Knoten right) {
                int result = left.getDistance() - right.getDistance(); return
                result;
            }
        });
13      startKnoten = knoten.get(0);
        startKnoten.setShortestNeighboar(null);
        startKnoten.setDistance(0);
        PQ.add(startKnoten);
        while(!PQ.isEmpty()) {
18          Knoten u = PQ.remove();
            for(int i = 0; i < u.getInzidenteKnoten().size(); i++) {
                Knoten v = u.getInzidenteKnoten().get(i).getKnoten();
                if(v.getDistance() == INFINITE_DISTANCE) PQ.add(v);
                if(test(u,v)) { // update prio
23                    PQ.remove(v);
                    PQ.add(v);
                }
            }
            PERM.add(u);
28      }
    }
    private static boolean test(Knoten u, Knoten v) {
        int weight = u.getWeight(v);
        if(v.getDistance() > (u.getDistance() + weight)) {
33            v.setDistance(u.getDistance() + weight);
            v.setShortestNeighboar(u);
            return true;
        } else return false;
    }
38    public static class Knoten {
        int distance = INFINITE_DISTANCE;;
        Knoten shortestNeighboar = null;
        String name;
        List<Kante> inzidenteKnoten = new ArrayList<Kante>();
43    public Knoten(String name) { name = name; }
        public int getWeight(Knoten v) {
            for(Kante k : inzidenteKnoten) if(k.getKnoten() == v) return k.
                getWeight();
            return 0;
        }
48    public int compareTo(Knoten k) { return this.distance - k.distance; }
    }
}

```

4.5.5 Kürzeste Pfade mit Bellman und Ford

Der Algorithmus von Bellman und Ford, der hier erklärt wird, kann auch mit negativen Gewichten umgehen (wenn der Graph gerichtet ist). Wenn negative Gewichte zugelassen sind, tritt ein grundsätzliches Problem auf: es kann ein Kreis negativer Länge von s aus erreichbar sein und damit $dist(s, v) = -\infty$ für alle Knoten v gelten, die von s aus erreichbar sind. Denn dieser Kreis kann unendlich viele Male durchlaufen werden und vermindert den Wert des Pfades bei jedem Durchlaufen.

- Es gibt n Phasen, wobei in der ersten Phase initialisiert wird.
- In jeder weiteren Phase wird jede Kante mit $test(u, v)$ überprüft und $d[v]$ aktualisiert.
- Am Ende enthält das Array $d[v]$ für alle Knoten v die Distanz zwischen s und v .

Es ist zentral, dass eine Kantenreihenfolge festgelegt wird, in der alle Kanten des Graphen besucht werden. Obwohl der Algorithmus für alle möglichen Kantenreihenfolgen funktioniert, hat die Wahl Einfluss darauf, wie früh die kürzesten Distanzen entdeckt werden.

Listing 15: BellmanFord

```
public class BellmanFord {
    public static final int INFINITE_DISTANCE = Integer.MAX_VALUE;
    private static Knoten startKnoten;
4    private static List<DoppelKnoten> reihenfolge = new ArrayList<DoppelKnoten>
        >();
    private static void startBellmanFord(ArrayList<Knoten> knoten) {
        startKnoten = knoten.get(0);
        startKnoten.setDistance(0);
        for(int i = 0; i < knoten.size(); i++) {
9            for(int k = 0; k < reihenfolge.size(); k++) {
                test(reihenfolge.get(k).getKnoten1(), reihenfolge.get(k).getKnoten2
                    ());
            }
        }
    }
14    private static boolean test(Knoten u, Knoten v) {
        if(u.getDistance() != INFINITE_DISTANCE) {
            int weight = u.getWeight(v);
            if(v.getDistance() > (u.getDistance() + weight)) {
                v.setDistance(u.getDistance() + weight);
19                v.setShortestNeighboar(u);
                return true;
            } else return false;
        } else return false;
    }
24 }
```

4.6 Spannbäume

Spannbäume sind Bäume die aus Kanten eines Graphen bestehen. Ein Baum mit n Knoten hat genau $n-1$ Kanten. Für einen Spannbaum werden also $n-1$ Kanten aus der Menge der Kanten ausgesucht, so dass ein zusammenhängender Graph entsteht. In diesem Graphen gibt es zwischen zwei Knoten genau einen Weg.

Im Code von BFS und DFS ist das Berechnen eines Spannbaumes schon *eingebaut*. Speichert jeder Knoten den adjazenten Knoten, von dem er entdeckt wurde, dann entspricht dies dem Speichern des Vaters im Spannbaum.



4.6.1 Definitionen

Spannbaum

Ein Spannbaum ST (Spanning Tree) eines Graphen $G = (V, E)$ besteht aus $n - 1$ Kanten der Menge E . Diese Kanten bilden einen zusammenhängenden Graphen, der alle Knoten aus V enthält.

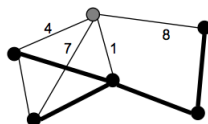
Minimaler Spannbaum

Ein minimaler Spannbaum MST (Minimum Spanning Tree) ist ein Spanbaum, bei dem die Summe der Kantengewichte minimal ist.

4.6.2 Algorithmus von Prim

Der Algorithmus von Prim, auch Prim-Dijkstra-Algorithmus genannt, funktioniert analog zum Dijkstra-Algorithmus. Der einzige Unterschied liegt darin, dass der Abstand zum Spannbaum und nicht der Abstand zum Startknoten als Priorität verwendet wird.

Der Abstand eines Knoten zum Spannbaum ist gleich dem kleinsten Gewicht einer Kante, die den Knoten mit einem Knoten des Spannbaums verbindet.



Im Beispiel hat der graue Knoten vier inzidente Kanten, die ihn mit Knoten aus dem bisherigen Spannbaum verbinden. Die Kante mit dem geringsten Gewicht hat das Gewicht $w(e) = 1$. Somit hat der graue Knoten den Abstand 1 zum Spannbaum.

Abstand von v zum minimalen Spannbaum MST:

$$\text{dist}(v, \text{MST}) = \min\{w(e) \mid e = \{v, u\} \in E \text{ und } v, u \in V\}$$

Ist der Knoten v zu keinem Knoten des Spannbaums adjazent, ist die Distanz unendlich.

Start Als Ausgangspunkt wird ein beliebiger Knoten genommen. Dieser ist zu Beginn der einzige Knoten im Spannbaum. Der Spannbaum enthält noch keine Kante.

Wachsen Es wird der Knoten gesucht, der den kleinsten Abstand vom Spannbaum hat. Dieser Knoten darf aber nicht schon im Spannbaum enthalten sein.

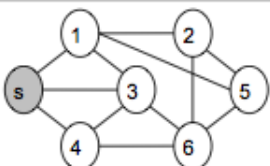
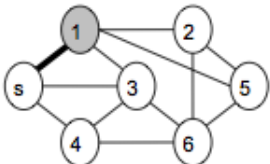
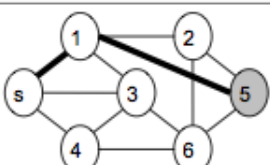
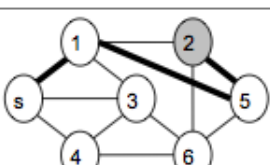
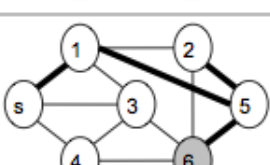
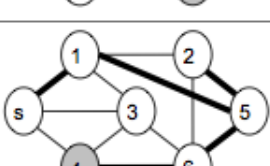
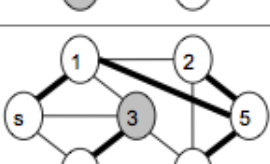
Der Knoten wird dann über die minimale Kante mit dem Spannbaum verbunden und in die Menge der Knoten des Spannbaums aufgenommen.

Ende Wenn alle Knoten im Spannbaum enthalten sind, dann ist die Berechnung zu Ende.

4.6.3 Beispiel des Prim Algorithmus

Gegeben ist ein Graph mit nebenstehender Adjazenzmatrix. Die Werte in den Feldern geben die Kantengewichte an. In diesem Beispiel wird ein Spannbaum ausgehend vom Knoten s berechnet. Das Bild zeigt den momentanen minimalen Spannbaum. Der nächste Knoten, die Queue und die Menge der Knoten werden in der mittleren Spalte angegeben. Rechts werden die Abstände der Knoten angegeben.

	s	1	2	3	4	5	6
s	0	2	0	5	7	0	0
1	2	0	8	6	0	3	0
2	0	8	0	0	0	2	4
3	5	6	0	0	1	0	5
4	7	0	0	1	0	0	1
5	0	3	2	0	0	0	2
6	0	0	4	5	1	2	0

	Queue: {s} MST: {}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr></table>	s	1	2	3	4	5	6	0	∞	∞	∞	∞	∞	∞
s	1	2	3	4	5	6										
0	∞	∞	∞	∞	∞	∞										
	Knoten: s Queue: {1,3,4} MST: {}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>2</td><td>∞</td><td>5</td><td>7</td><td>∞</td><td>∞</td></tr></table>	s	1	2	3	4	5	6	0	2	∞	5	7	∞	∞
s	1	2	3	4	5	6										
0	2	∞	5	7	∞	∞										
	Knoten: 1 Queue: {5, 3, 4, 2} MST: {s}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>8</td><td>5</td><td>7</td><td>3</td><td>∞</td></tr></table>	s	1	2	3	4	5	6	0	0	8	5	7	3	∞
s	1	2	3	4	5	6										
0	0	8	5	7	3	∞										
	Knoten: 5 Queue: {2, 6, 3, 4} MST: {s, 1}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>2</td><td>5</td><td>7</td><td>0</td><td>2</td></tr></table>	s	1	2	3	4	5	6	0	0	2	5	7	0	2
s	1	2	3	4	5	6										
0	0	2	5	7	0	2										
	Knoten: 2 Queue: {6, 3, 4} MST: {s, 1, 5}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>0</td><td>5</td><td>7</td><td>0</td><td>2</td></tr></table>	s	1	2	3	4	5	6	0	0	0	5	7	0	2
s	1	2	3	4	5	6										
0	0	0	5	7	0	2										
	Knoten: 6 Queue: {4, 3} MST: {s, 1, 5, 2}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>0</td><td>5</td><td>1</td><td>0</td><td>0</td></tr></table>	s	1	2	3	4	5	6	0	0	0	5	1	0	0
s	1	2	3	4	5	6										
0	0	0	5	1	0	0										
	Knoten: 4 Queue: {4} MST: {s, 1, 5, 2, 6}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	s	1	2	3	4	5	6	0	0	0	1	0	0	0
s	1	2	3	4	5	6										
0	0	0	1	0	0	0										
	Knoten: 3 Queue: {} MST: {s, 1, 5, 2, 6, 4}	<table><tr><th>s</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	s	1	2	3	4	5	6	0	0	0	0	0	0	0
s	1	2	3	4	5	6										
0	0	0	0	0	0	0										

5 Spieltheorie

5.1 Minimax-Algorithmus

Der Minimax-Algorithmus ist ein Algorithmus zur Ermittlung der optimalen Spielstrategie für endliche Zwei-Personen-Nullsummenspiele mit perfekter Information. Zu diesen Spielen gehören insbesondere Brettspiele wie Schach, Go, Reversi, Dame, Mühle und Vier gewinnt, bei denen beide Spieler stets die gesamte Historie der Partie kennen. Auch für Spiele mit Zufallseinfluss wie Backgammon lässt sich der Minimax-Algorithmus auf Grundlage von Erwartungswerten erweitern. In der Regel, aber nicht ausschließlich, wird der Minimax-Algorithmus auf Spiele mit abwechselndem Zugrecht angewandt.

5.1.1 Beschreibung

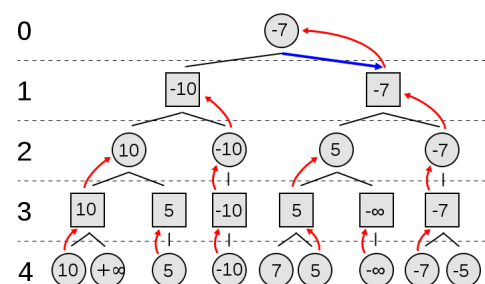
Die Kreise stellen die Züge der Spieler im Algorithmus dar (Maximierung), die Quadrate die Züge der Gegner (Minimierung). Die Werte in den Kreisen und Quadraten stellen den Wert α des Minimax-Algorithmus dar. Die roten Pfeile repräsentieren den gewählten Zug, die Nummern links die Baumtiefe und die blauen Pfeile den gewählten Zug.

Die Knoten der Ebenen 0 und 2 entsprechen Spielsituationen, in denen Spieler A am Zug ist. Hier

wird jeweils die Bewertungsfunktion der untergeordneten Knoten maximiert, d. h. der für Spieler A günstige Zug ausgewählt und dessen Wert dem Elternknoten zugewiesen.

Die Knoten der Ebenen 1 und 3 entsprechen Spielsituationen, in denen Spieler B am Zug ist. Hier wird jeweils die Bewertungsfunktion der untergeordneten Knoten minimiert, d. h. der für Spieler B günstigste Zug ausgewählt und dessen Wert dem Elternknoten zugewiesen.

Der Algorithmus beginnt unten bei den Blättern und geht dann nach oben bis zur Wurzel. In Ebene 3 wählt der Algorithmus den kleinsten Wert der Kindknoten und weist diesen dem Elternknoten zu (es wird minimiert). In Ebene 2 wird dann der jeweils größte Kindknoten dem Elternknoten zugewiesen (es wird maximiert). Dies wird abwechselnd so lange durchgeführt bis die Wurzel erreicht ist. Der Wurzel wird der Wert des größten Kindknoten zugewiesen. Dabei handelt es sich dann um den Zug der gespielt werden soll.



5.1.2 Implementierung

Listing 16: Auszug Main

```
1  gespeicherterZug = NULL;
   int gewünschteTiefe = 4;
   int bewertung = max(+1, gewünschteTiefe);
   if (gespeicherterZug == NULL)
       es gab keine weiteren Zuege mehr;
6  else
    gespeicherterZug ausführen;
```

Listing 17: Funktionen

```
int max(int spieler, int tiefe) {
    if (tiefe == 0 or keineZuegeMehr(spieler)) return bewerten();
3    int maxWert = -unendlich;
```

```

    generiereMoeglicheZuege(spieler);
    while (noch Zug da) {
        fuehreNaechstenZugAus();
        int wert = min(-spieler, tiefe-1);
8      macheZugRueckgaengig();
        if (wert > maxWert) {
            maxWert = wert;
            if (tiefe == anfangstiefe) gespeicherterZug = Zug;
        }
13    }
    return maxWert;
}

int min(int spieler, int tiefe) {
    if (tiefe == 0 or keineZuegeMehr(spieler))
18    return bewerten();
    int minWert = unendlich;
    generiereMoeglicheZuege(spieler);
    while (noch Zug da) {
        fuehreNaechstenZugAus();
23    int wert = max(-spieler, tiefe-1);
        macheZugRueckgaengig();
        if (wert < minWert) {
            minWert = wert;
        }
28    }
    return minWert;
}

```

5.2 Alpha-Beta-Suche

Die Alpha-Beta-Suche, auch Alpha-Beta-Cut oder Alpha-Beta-Pruning genannt, ist eine optimierte Variante des Minimax-Suchverfahrens, also eines Algorithmus zur Bestimmung eines optimalen Zuges bei Spielen mit zwei gegnerischen Parteien. Während der Suche werden zwei Werte Alpha und Beta aktualisiert, die angeben, welches Ergebnis die Spieler bei optimaler Spielweise erzielen können. Mit Hilfe dieser Werte kann entschieden werden, welche Teile des Suchbaumes nicht untersucht werden müssen, weil sie das Ergebnis der Problemlösung nicht beeinflussen können.

Die einfache (nicht optimierte) Alpha-Beta-Suche liefert exakt dasselbe Ergebnis wie die Minimax-Suche.

5.2.1 Beschreibung

Der Minimax-Algorithmus analysiert den vollständigen Suchbaum. Dabei werden aber auch Knoten betrachtet, die in das Ergebnis (die Wahl des Zweiges an der Wurzel) nicht einfließen. Die Alpha-Beta-Suche versucht, möglichst viele dieser Knoten zu ignorieren.

Ein anschauliches Beispiel für die Funktionsweise ist ein Zweipersonenspiel, bei dem der erste Spieler eine von mehreren Taschen auswählt und von seinem Gegenspieler den Gegenstand mit geringstem Wert aus dieser Tasche erhält.

Der Minimax-Algorithmus durchsucht für die Auswahl alle Taschen vollständig und benötigt somit viel Zeit. Die Alpha-Beta-Suche hingegen durchsucht zunächst nur die erste Tasche vollständig nach dem Gegenstand mit minimalem Wert. In allen weiteren Taschen wird nur solange gesucht, bis der Wert eines Gegenstands dieses Minimum unterschreitet. Ist dies der Fall, wird die Suche in dieser Tasche abgebrochen und die nächste Tasche untersucht. Andernfalls ist diese Tasche eine bessere Wahl für den ersten Spieler und ihr minimaler Wert dient für die weitere Suche als neue Grenze.

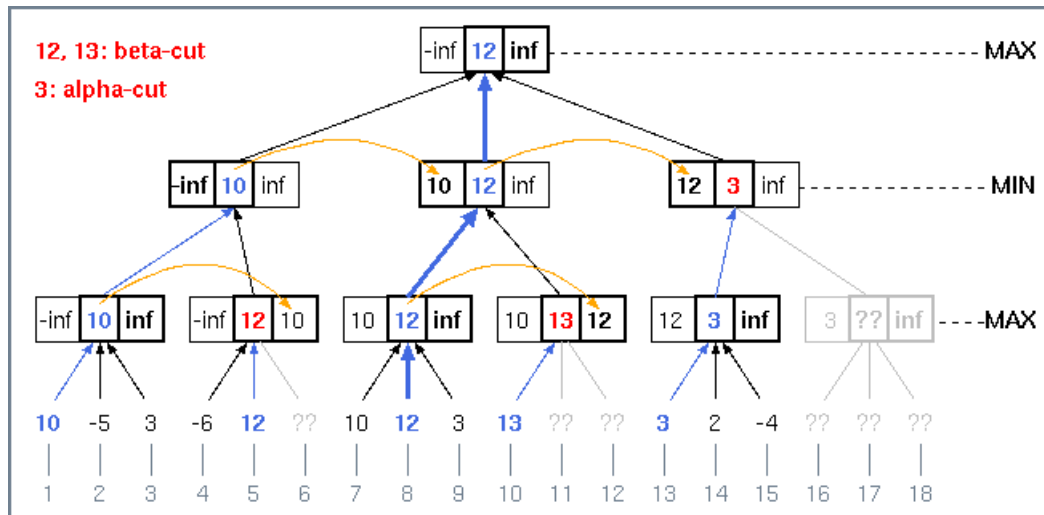
Ähnliche Situationen sind jedem Schachspieler vertraut, der gerade einen konkreten Zug darauf prüft, ob er ihm vorteilhaft erscheint. Findet er bei seiner Analyse des Zuges eine für sich selbst ungünstige Erwiderung des Gegners, dann wird er diesen Zug als *widerlegt* ansehen und verwerfen. Es wäre völlig sinnlos, noch weitere Erwiderungen des Gegners zu untersuchen um festzustellen, ob der Gegner noch effektivere Widerlegungen besitzt und wie schlecht der geplante Zug tatsächlich für den Spieler ist.

5.2.2 Der Algorithmus

Die Alpha-Beta-Suche arbeitet prinzipiell genauso wie obige informelle Beschreibung. Die Idee ist, dass zwei Werte (Alpha und Beta) weitergereicht werden, die das Worst-Case-Szenario der Spieler beschreiben. Der Alpha-Wert ist das Ergebnis, das Spieler A mindestens erreichen wird, der Beta-Wert ist das Ergebnis, das Spieler B höchstens erreichen wird (Hier ist zu beachten, dass es für Spieler B darum geht, ein möglichst niedriges Ergebnis zu erhalten, da er ja *minimierend* spielt!)

Besitzt ein maximierender Knoten (von Spieler A) einen Zug, dessen Rückgabe den Beta-Wert überschreitet, wird die Suche in diesem Knoten abgebrochen (Beta-Cutoff, denn Spieler B würde A diese Variante erst gar nicht anbieten, weil sie sein bisheriges Höchst-Zugeständnis überschreiten würde). Liefert der Zug stattdessen ein Ergebnis, das den momentanen Alpha-Wert übersteigt, wird dieser entsprechend nach oben angehoben.

Analoges gilt für die minimierenden Knoten, wobei bei Werten kleiner als Alpha abgebrochen wird (Alpha-Cutoff) und der Beta-Wert nach unten angepasst wird.



Obige Abbildung zeigt einen Beispielbaum mit 18 Blättern, von denen nur 12 ausgewertet werden. Die drei umrandeten Werte eines inneren Knotens beschreiben den Alpha-Wert, den Rückgabewert und den Beta-Wert. Der Suchalgorithmus verwendet ein sogenanntes Alpha-Beta-Fenster, dessen untere Grenze der Alpha-Wert und dessen obere Grenze der Beta-Wert darstellt. Dieses Fenster wird zu den Kindknoten weitergegeben, wobei in der Wurzel mit dem maximalen Fenster $[-\text{inf}, \text{inf}]$ begonnen wird. Die Blätter 1, 2 und 3 werden von einem maximierenden Knoten ausgewertet und der beste Wert 10 wird dem minimierenden Vaterknoten übergeben. Dieser passt den Beta-Wert an (signalisiert durch die orange Linie links unten) und übergibt das neue Fenster $[-\text{inf}, 10]$ dem nächsten maximierenden Kindknoten, der die Blätter 4, 5 und 6 besitzt. Der Rückgabewert 12 von Blatt 5 ist aber so gut, dass er den Beta-Wert 10 überschreitet. Somit muss Blatt 6 nicht mehr betrachtet werden, weil das Ergebnis 12 dieses Teilbaumes besser ist, als das des linken Teilbaumes, und deshalb vom minimierenden Spieler nie gewählt werden würde.

Ähnlich verhält es sich beim minimierenden Knoten mit dem 3-Alpha-Cutoff. Obwohl dieser Teilbaum erst teilweise ausgewertet wurde, ist klar, dass der maximierende Wurzelknoten diese Variante niemals wählen würde, weil der minimierende Knoten ein Ergebnis von höchstens 3 erzwingen könnte, während aus dem mittleren Teilbaum das Ergebnis 12 sichergestellt ist.

5.2.3 Implementierung

Listing 18: Auszug Main

```
gespeicherterZug = NULL;
int gewuenschteTiefe = 4;
int bewertung = max(+1, gewuenschteTiefe, -unendlich, +unendlich);
if (gespeicherterZug == NULL)
5   es gab keine weiteren Zuege mehr;
else
    gespeicherterZug ausführen;
```

Listing 19: Funktionen

```
int max(int spieler, int tiefe, int alpha, int beta) {
    if (tiefe == 0 or keineZuegeMehr(spieler)) return bewerten();
3   int maxWert = alpha;
    generiereMoeglicheZuege(spieler);
```

```

    while (noch Zug da) {
        fuehreNaechstenZugAus();
        int wert = min(-spieler, tiefe-1, maxWert, beta);
8      macheZugRueckgaengig();
        if (wert > maxWert) {
            maxWert = wert;
            if (maxWert >= beta) break;
            if (tiefe == anfangstiefe) gespeicherterZug = Zug;
13      }
    }
    return maxWert;
}

int min(int spieler, int tiefe, int alpha, int beta) {
18  if (tiefe == 0 or keineZuegeMehr(spieler)) return bewerten();
    int minWert = beta;
    generiereMoeglicheZuege(spieler);
    while (noch Zug da) {
        fuehreNaechstenZugAus();
23      int wert = max(-spieler, tiefe-1, alpha, minWert);
        macheZugRueckgaengig();
        if (wert < minWert) {
            minWert = wert;
            if (minWert <= alpha) break;
28      }
    }
    return minWert;
}

```
