PDA - Notizen

1. Einführung

Algorithmus ist eine endliche Folge von Anweisungen, die in einer Maschine repräsentiert werden können. Algorithmus Eigenschaften: Endliche Repräsentation/Menge/Ergebnis, Allgemein, Terminiert

Compiler: Prüft Lexikalik (gültige Zeichen), Syntax (gültiger Aufbau), Semantik (Interpretation)

Interpreter: Programm-Compilierung „auf Zuruf“

- Während der Ausführung wird der Code compiliert
- Interpretiert Programme sind meist langsamer als compilierte Programme
- Dafür flexibel einsetzbar (JavaScript in diversen Browser unterschiedlicher Hardware)

EBNF (Extended-Backus-Naur-Form):

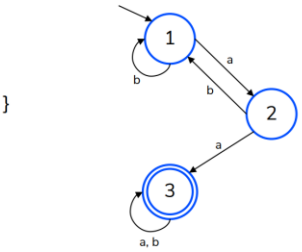
- Expression = Expression, “+“, Expression;
- Expression = “ (“, Expression, “)“;
- LettersOrDigits = {Letter | Digit};
- Terminalsymbole / Konstanten: +, -, (,)
- Nicht Terminale: Müssen beschrieben werden
- {x} – x kann belieb oft wiederholt werden (null bis unendlich)
- [x] – x ist optional, d.h. es kann null oder einmal vorkommen
- (X | Y), Z – gruppiert Ausdrücke
- 1.Klammern (), 2. Sequenzen (,), 3. Alternativen (|)

Endliche Automaten:

Beispiel: Endliche Automaten

- Zustandsmenge: { 1, 2, 3 }
- Startzustand: 1
- Endzustandsmenge: { 3 }
- Eingabealphabet: { „a“, „b“ }
- Zustandsübergänge

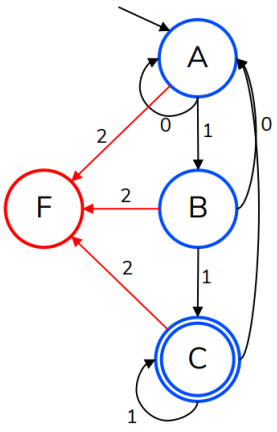
Zustand	Gelesenes Zeichen	Nächster Zustand
1	a	2
1	b	1
2	a	3
2	b	1
3	a	3
3	b	3



Repräsentation eines Endlichen Automaten als gerichteter Graph: Knoten (Kreise) repräsentieren Zustände, Kanten (Pfeile) repräsentieren Übergänge in Abhängigkeit von Eingabezeichen.

▪ Existiert zu jedem Zustand und zu jeder Eingabe eines endlichen Automaten maximal ein Folgezustand, dann heißt der Automat **deterministisch**, sonst heißt er **nichtdeterministisch**.

- Zustandsmenge: { A, B, C, F }
- Startzustand: A
- Endzustandsmenge: { C }
- Eingabealphabet: { „0“, „1“, „2“ }
- **Fehlerzustand: F**
  - spezieller Endzustand
  - signalisiert *keine* Akzeptanz des Eingabewortes (deshalb nicht Teil des Endzustandsmenge)
  - kann üblicherweise nicht wieder verlassen werden
  - gesonderte Fehlerbehandlung



2. Java

- Java Eigenschaften: plattformunabhängig durch JVM, objekt-orientiert, einfach, robust, sicher, schnell, verteilt, dynamisch, weit verbreitet, kostenlos
- != ungleich
- && und
- || oder
- Kommentare setzen zur Nachvollziehbarkeit
- Leerzeichen setzen vor und nach Klammern und um Operatoren, Variablen beginnen klein, CamelCase und Klassen beginnen groß

3. Elementare Datentypen

- Elementare Datentypen sind unteilbar (atomar) wie integer oder char
- Abstrakte Datentypen: Klassen von Objekten
- Referenztypen Wert ist eine Speicheradresse, unter der ein tatsächlicher Wert nachgeschlagen wird
- Binärrechnung erfolgt durch Teilen durch 2 und wenn Rest existiert, dann 1 und wenn nicht dann 0
- String ist **kein** elementarer Datentyp

5. Arrays 1

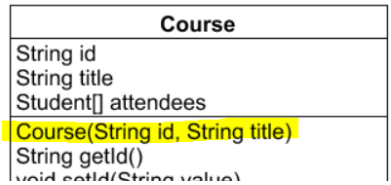
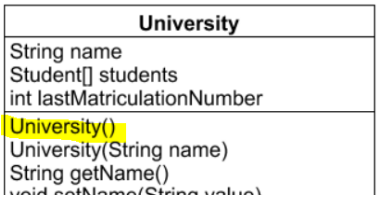
- Reihe von Werten des gleichen Typs
- Feste Größe, die ist jedoch beliebig
- int[] numbers = new int[6];
- Nullpointer: numbers = null;
- numbers = new int[] { 3, 5, 2, 7, 0, 9 };
- **while:** int i = 0; while (i < 5) { System.out.println(i); i++; }
- **for:** for (int i = 0; i < 5; i++) { System.out.println(i); }
- **for each:** for (int value : numbers) { System.out.println(value); }
- **break**; um Ausführung an einer Stelle abzubrechen
    - gilt nur für den Codeblock wo es drin steht
- Sequentielle Suche um einen Wert im Array zu finden
    - ```
      for (int i = 0; i < keys.length; i++) {
          if (keys[i] == key) {
              // key found at position i
              break;
          } else if (keys[i] > key) {
              // key not found
              break;
          }
      }
      ```

6. Aufwand von Algorithmen

7. Arrays 2

8. Methoden

10. Rekursion

11. Quicksort und Mergesort
      ```

13. Klassen und Objekte I

14. Klassen und Objekte II

- Hier ist kein Default Konstruktor möglich im Aufruf





- Default Konstruktor muss explizit angelegt werden
- Getter bei Arrays bietet keinen Schutz vor Veränderung des Arrays
    - -> Lösung dafür wäre eine Kopie des Arrays auszugeben
    - Alternativ eine Anzahl angeben

this.xxx greift auf das Attribut der Klasse zu und nicht auf den Parameter, der mitgegeben wird

- Enthält eine Referenz auf die aktuelle Instanz des Objektes

16. Testen und Dokumentation

- Dokumentation zur Nachvollziehbarkeit sehr wichtig (um auch eigenen Code zu checken)
- JavaDoc Kommentare mit /** einzuleiten (um Dokumente generieren zu lassen)
    - Oberhalb der Klasse / Methode
- Es gibt keine vollständigen Tests hinsichtlich 100% Korrektheit (dient nur dem Aufdecken von Fehlern)
- Defensives Programmieren: Jeder Eingabe misstrauen und überprüfen/testen
- Testgetriebene Softwareentwicklung: Tests zuerst definieren, danach Code schreiben
    - Testfälle, die fehlschlagen sollen
    - Testfälle, die erfolgreich durchlaufen werden sollen
- Grenzwerte/Randfälle austesten (Min.-/Max.-Wert / negative Zahlen usw.)
- JUnit-Framework zur Unterstützung
- Programmverifikation bedeutet Korrektheit bezüglich der formalen Spezifikationen
    - Precondition und Postcondition bei jeder Anweisung
    - Bei Schleifen Invarianten verwenden
    - Aufwändig und kompliziert -> für die Klausur reicht testen
- Debugging zur Fehlersuche
    - An jeder Stelle println ausführen möglich, aber aufwändig
    - Debugger ausführen (Käfer Icon)
        - Breakpoints definieren, dort dann aktuelle Werte einsehbar
        - Step into und Step over (Methoden) für nächste Zeile / nach Methode
- RuntimeException
    - ArrayIndexOutOfBound
    - Nullpointer

17. Listen

- Flexibler als ein Array
  - Einfügen von Elementen ist billig
  - Löschen von Elementen ist billig
  - Holen von Elementen ist teuer
  - Benötigt mehr Speicher als ein Array (für Referenzen)

LinkedStringList
int size() boolean isEmpty() void clear() String get(int index) String remove(int index) void prepend(String value) void add(String value) boolean insert(String value, int index) boolean set(String value, int index) boolean contains(String value)

.iste von Strings mit Operationen zum
- Einfügen (**prepend**, **add**, **insert**),
- Löschen (**clear**, **remove**),
- Ersetzen (**set**) und
- Abfragen (**size**, **isEmpty**, **get**, **contains**)

-
- Es ist nicht möglich an eine gewisse Indexposition direkt zu kommen
  - Es wird immer nur vom Startelement zum nächsten gesprungen
  -
```
LinkedStringListElement current;
current = start;
current = current.getNext();
current = current.getNext();
return current.getValue(); // "C"
```
- Elemente erstellen (hinten dran)
  -
```
LinkedStringListElement current;
current = start;
LinkedStringListElement elem2 = new LinkedStringListElement();
elem2.setValue("B");
elem2.setNext(current.getNext());
current.setNext(elem2);
```
  - Elem1 auf Elem3
  - Elem2 wird danach erst dazwischengesetzt
- Einfügen (vorne dran)
  -
```
public void prepend(String value) {
    LinkedStringListElement elem = new LinkedStringListElement();
    elem.setValue(value);
    elem.setNext(start);
    start = elem;
}
```
- Einfügen am Ende
  -
```
public void add(String value) {
    LinkedStringListElement elem = new LinkedStringListElement();
    elem.setValue(value);
    if (start == null) { // list is empty
        start = elem;
    } else {
        LinkedStringListElement current = start;
        while (current.getNext() != null) { // find last element
            current = current.getNext();
        }
        current.setNext(elem);
    }
}
```

elem3 : LinkedStrinc
next = **null**
value = **"C"**

- Einfügen an Indexposition x
  - ```
    public boolean insert(String value, int index) {
        if (index < 0) { // negative index not allowed
            return false;
        }
        if (index == 0) { // insert at the beginning of the list
            prepend(value);
            return true;
        }
        if (start == null) { // can only insert at index 0 in empty list
            return false;
        }
    ```
- Löschen (bzw. überspringen)
  - ```
    LinkedStringListElement current;
    current = start;
    current.setNext(current.getNext().getNext());
    ```
  - Zeiger wird einfach auf den Nachfolger gesetzt
- Remove
  - ```
    public String remove(int index) {
        if (start == null) { // list is empty
            return null;
        }
        if (index == 0) { // remove from the beginning of non-empty list
            String result = start.getValue();
            start = start.getNext();
            return result;
        }
    ```
- Nach Wert in der Liste suchen (contains)
  - ```
    public boolean contains(String value) {
        LinkedStringListElement current = start;
        while (current != null) {
            if (current.getValue().equals(value)) {
                return true; // found
            }
            current = current.getNext();
        }
        return false; // not found
    }
    ```
- Iteration
  - ```
    LinkedStringList list = new LinkedStringList();
    list.add("A");
    list.add("B");
    list.add("C");
    list.reset(); // reset iteration
    while (list.hasNext()) {
        System.out.println(list.getNext());
    }
    ```
    ```

- Interne Struktur:

```
public void reset() {
    iterCurrent = start;
}

public String getNext() {
    String result = iterCurrent.getValue();
    iterCurrent = iterCurrent.getNext();
    return result;
}

public boolean hasNext() {
    return iterCurrent != null;
}
```

- **Autoboxing:** Der Compiler ersetzt transparent Instanzen der einfachen Datentypen (int, char, double, …) durch ObjektInstanzen der analogen Klassen (Integer, Character, Double, …)
- LinkedList in Java

```
LinkedList<String> stringValues = new LinkedList<String>();
stringValues.add("abc");

LinkedList<Integer> intValues = new LinkedList<Integer>();
intValues.add(1);
```

- Doppelt verkettete Listen (mit previous)
  - Elemente anhängen ist billig
  - Elemente holen ist billiger als bei einfach verketteten Listen (kann sowohl von hinten als auch von vorne durchlaufen)
  - Benötigt mehr Speicher als einfach verkettete Listen (für Referenzen)
  - Bi-direktionales Iterieren ist einfach und billig
- ArrayList
  - Tatsächliche Länge >= offizielle Länge (die sichtbar ist an der Oberfläche)
  - Sollte das offizielle > tatsächlich, dann muss neues array mit größerer tatsächlicher Länge erstellt werden und vom alten rüberkopiert werden
  - Vergrößerung soll möglichst selten nötig sein
  - Ansatz: verdopple bei jeder Vergrößerung die Länge des tatsächlichen Arrays

```
public class ArrayStringList {

    private static final int INITIAL_CAPACITY = 12;
    private String[] values = new String[INITIAL_CAPACITY];
    private int size = 0; // official size

    public int size() {
        return size;
    }

    public String get(int index) {
        if (index < 0 || index >= size) { // invalid index
            return null;
        }
        return values[index];
    }
```

```java
private void checkEnlargement() {
    if (size == values.length) {
        String[] tmp = new String[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}

public void add(String value) {
    checkEnlargement();
    values[size] = value;
    size++;
}
```

- Oft ist die Verwendung von ArrayList effizienter als die von LinkedList
- **Datentyp Stack**
    - Einfügen (push) → oben auf den Stack legen
    - Löschen (pop) → oben vom Stack nehmen
    - Abfragen (isEmpty, peek)

```java
public class StringStack {

    private LinkedStringList values = new LinkedStringList();

    public boolean isEmpty() {
        return values.size() == 0;
    }

    public String peek() {
        return values.get(0);
    }

    public String pop() {
        return values.remove(0);
    }

    public void push(String value) {
        values.insert(value, 0);
    }
}
```

    - Einfach zu implementieren per Delegation ▪
        - Verwendung einer Liste zur internen Repräsentation der Daten
        - Stack-Methoden greifen auf die Methoden der Liste zurück

18. Bäume

Queue

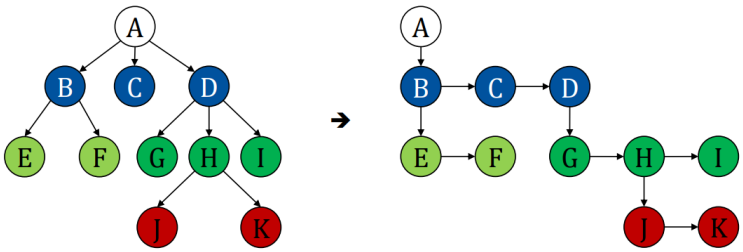
Queue
void enqueue(String value) String peek() String dequeue() boolean isEmpty()

Queue (dt. Warteschlange)
Datenstruktur mit Operationen zum

- Einfügen (**enqueue**),
- Löschen (**dequeue**) und
- Abfragen (**isEmpty**, **peek**)

```
public class StringQueue {

    private LinkedStringList values = new LinkedStringList();

    public boolean isEmpty() {
        return values.size() == 0;
    }

    public String peek() {
        return values.get(0);
    }

    public String dequeue() {
        return values.remove(0);
    }

    public void enqueue(String value) {
        values.add(value);
    }
```

- **Bäume** in Java
  - Nachteil: Unvollständige Bäume mit großem k haben viele leere Teilbäume, d.h. viele null-Referenzen
  - Lösung: Umwandlung in einen Binärbaum (k = 2)



  - Knotenklasse

```
public class Node {

    private String content;
    private Node left;
    private Node right;

    public Node(String content) {
        this.content = content;
        left = null;
        right = null;
    }

    // getter and setter ...

}
```

- Binärbaumklasse

```
public class BinTree {

    private Node root;

    public BinTree() { }

    public BinTree(String content) {
        this.root = new Node(content);
    }
```

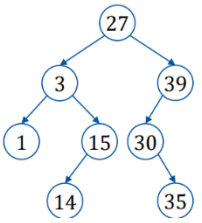
```
  public BinTree(BinTree left, String content, BinTree right) {
      root = new Node(content);
      if (left != null) {
          root.setLeft(left.root);
      }
      if (right != null) {
          root.setRight(right.root);
      }
  }

  private BinTree(Node root) {
      this.root = root;
  }

  public boolean isEmpty() {
      return root == null;
  }
```

```
public String getValue() {
    if (isEmpty()) {
        return null; // error
    }
    return root.getContent();
}

public BinTree getLeft() {
    if (isEmpty()) {
        return null; // error
    }
    return new BinTree(root.getLeft());
}

public BinTree getRight() {
    if (this.isEmpty()) {
        return null; // error
    }
    return new BinTree(root.getRight());
}
```

- Es gibt verschiedene Reihenfolgen der Traversierung
  - inorder: linker Teilbaum, Daten, rechter Teilbaum
  - preorder: Daten, linker Teilbaum, rechter Teilbaum
  - postorder: linker Teilbaum, rechter Teilbaum, Daten
- Tiefensuche
  - Geht von oben nach unten tief zu jedem Blatt bis es das gewünschte Ergebnis gefunden hat (vertikal)
  - Backtracking um zu erkennen, welche Stränge schon abgearbeitet wurden
  - Entweder lösbar über Rekursion oder Iterativ mit Stack
- Breitensuche
  - Horizontale Suche von der Oberfläche (links nach rechts) bis in die Tiefe

        - Iterativer Ansatz mit einer Queue
    - Einfügen von Knoten entweder über Tiefe zuerst oder über Breite zuerst
    - Löschen abhängig davon was gelöscht wird
        - Blatt einfach weg
        - Ast mit nur einem Strang geht auch einfach weg
        - Ast mit mehreren Unterästen is schwierig (Blatt wird zum Oberast)

19. AVL-Bäume



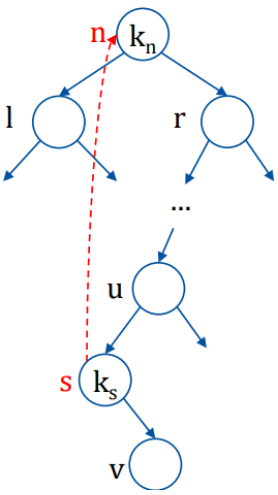
- Ein Binärbaum ist ein Suchbaum, wenn für alle Knoten n gilt:
    - jeder Schlüssel im linken Teilbaum von n ist kleiner als der Schlüssel in n
    - der Schlüssel in n ist kleiner als alle Schlüssel im rechten Teilbaum von n

```
public KeyValue search(int key) {
    if (!isEmpty()) {
        if (key == getValue().getKey()) {
            return getValue();
        } else if (key < getValue().getKey() && getLeft() != null) {
            // search left sub-tree
            return getLeft().search(key);
        } else if (key > getValue().getKey() && getRight() != null) {
            // search right sub-tree
            return getRight().search(key);
        }
    }
    return null; // key not found
}
```

- Suche nach k

```
public void insert(KeyValue content) {
    if (isEmpty()) {
        root = new Node(content);
    } else {
        if (content.getKey() == getValue().getKey()) {
            root.setContent(content);
        } else if (content.getKey() < getValue().getKey()) {
            if (getLeft() == null) {
                setLeft(new SearchTree(content));
            } else {
                getLeft().insert(content);
            }
        } else if (content.getKey() > getValue().getKey()) {
            if (getRight() == null) {
                setRight(new SearchTree(content));
            } else {
                getRight().insert(content);
            }
        }
    }
}
```

- Einfügen
- Löschen:
    - Wird die Wurzel gelöscht, dann wird auf der rechten Seite geschaut, was der nächstgrößere Wert ist (die kleinste auf der rechten Seite). Dieser hat dementsprechend keinen linken Unterteilbaum.
    - Nach Löschen muss ggfls. Rebalanciert werden
- !!!Rotation in AVL-Bäumen üben, wichtig für Klausur
    - L-Rotation
    - R-Rotation
    - LR-Rotation
    - RL-Rotation



20. Suche in Strings

- Methode von Boyer Moore
  - bewege das Muster von links nach rechts über den Text
  - vergleiche die Zeichen von Text und Muster von rechts nach links
  - bei Ungleichheit: bewege das Muster so weit wie nötig, damit die Ungleichheit nicht wieder auftritt

Muster	a	b	c	b	a	f	b	b	d	a
Index	0	1	2	3	4	5	6	7	8	9

$m = 10$

Zeichen	a	b	c	d	f	Sonstige
$delta_1$	5 (= m – 5)	2 (= m – 8)	7 (= m – 3)	1 (= m – 9)	4 (= m – 6)	10 (nicht im Muster)

  -
  - Deltatabelle kann nur bei Fall 2 angewendet werden
    - Fall 2 ist, dass rechts von dem ungültigen Zeichen das Zeichen selbst nicht mehr vorkommt
    - Fall 2 ist bei diesem Beispiel gegeben

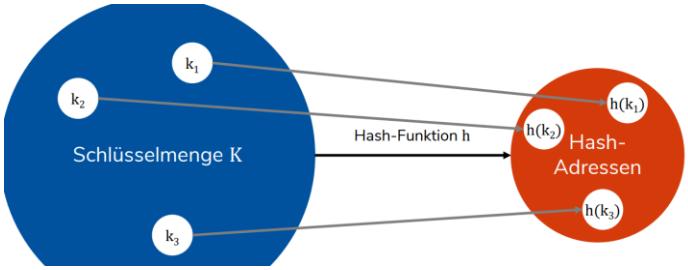
	0	1	1	1	0	1	0	0
i += 1	1	1	1	1	0	0	1	
i += 1		1	1	1	1	0	0	1

  - Weil das letzte ignoriert wird ist es nicht Fall 1
    -

i += 3				1	1	1	1	0	0	1

  - Worst-case: O(m · n) (genauso schlecht wie die naiven Methoden zuvor)
    - Kommt aber nie vor bzw. nur sehr sehr selten, daher effektiver als die naiven
  - Beispiel
    - ▪ Text = a…a (n-mal)
    - Muster = ba…a (m-1-mal)
    - jedes Mal gibt es eine Ungleichheit beim m-ten Vergleich
      - j = 1, delta1 (a) = 1
      - m – j > delta1 (a) = 1
      - Muster wird um eins gegen den Text verschoben
      - ca. n Verschiebungen mal m Vergleiche pro Verschiebung

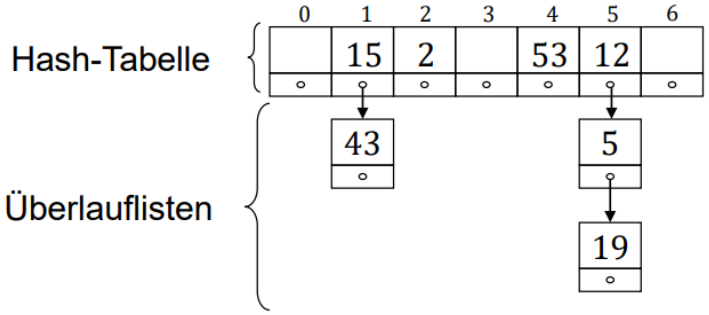
21. Hashing

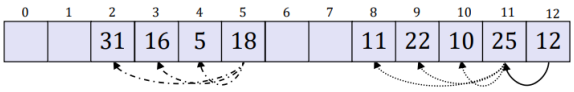
- Schnelle Suche durch direkter Aufruf
- 

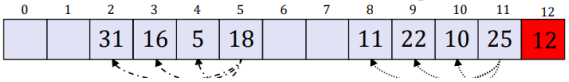
- Ziel ist möglichst bijektiv abzubilden, damit so wenig Speicher wie möglich, aber so viel wie nötig verwendet wird (16000 Studenten ⇔100000 Einträge der Tabelle)
- Ziel: Finde eine Hash-Funktion h, die im Durchschnitt wenige Kollisionen erzeugt
- Von Mensch ist eher die Division mit Rest Methode

- Maschinell ist eher die Multiplikative Methode

Kollision ist, wenn zwei oder mehrere Werte auf der gleichen Stelle zeigen

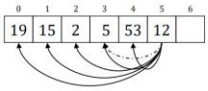
- Überlauflisten
  - Eigene Liste die für Kollisionsfälle erstellt wird (an jeder Stelle der Hashing Tabelle)
  - Schlüsselgröße mod Größe der Tabelle ergibt die Position, wo der Schlüssel hinzugefügt wird
  - 

- Sondieren (Open Hashing)
  - Ist eher geeignet, wenn man weiß, dass nicht viel gelöscht wird
  - Es wird eine freie Stelle gesucht (freier Speicherplatz in der Hashing Tabelle)
  - Sei s: $\mathbb{N} \times K \rightarrow \mathbb{N}$ mit $s(j, k) \in \{ 0, \ldots, m-1 \}$ eine Nachfolgerfunktion
  - Die Sondierungsreihenfolge wird berechnet durch $(h(k) - s(j, k)) \bmod m$
    - k ist der Schlüssel, der eingefügt werden soll
    - j ist die Nummer des Einfügeversuchs für k (beginnend bei 0)
  - **Einfügen**: Zyklisch links wird geschaut, ob was frei ist
    $$(h(k) - j) \bmod m = ((k \bmod 13) - j) \bmod 13$$
    - Einfügen der Schlüssel 18, 12, 16, 22, 10, 25, 5, 31, 11
  - Beispiel:
    

  - **Löschen:** ist problematisch, da Elemente, die später (weiter links) eingefügt wurden nicht mehr erkannt bzw. gefunden werden, weil die Sondierungsreihenfolge nicht mehr gegeben ist.
    Problem: Dies kann die Sondierungsreihenfolge unterbrechen
    - Beispiel: Nach dem Löschen von 12 kann die 25 nicht mehr gefunden werden
    

    - Strategie 1: Es wird nur als gelöscht markiert, aber für das Suchen wird so getan als wäre er noch da
    - Strategie 2: Reorganisiere die Tabelle, damit die Sondierungsreihenfolge nicht unterbrochen ist
  - Sondierungsmethoden
    - Lineares Sondieren

- Lineares Sondieren: $s(j, k) = j$
- Beispiel
  - $m = 7$ und $h(k) = k \bmod m$
  - Einfügen von 12, 53, 5, 15, 2, 19



  - Problem: es entwickeln sich Sondierungsketten, d.h. zusammenhängende belegte Bereiche

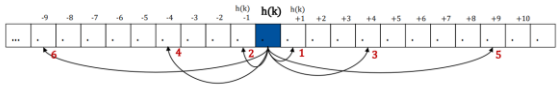
- Nachteil
  - Gefahr von langen Sondierungsketten
  - nach dem Einfügen von 15, 53 und 5, ist die Wahrscheinlichkeit, dass T[i] beim nächsten Einfügen belegt ist:

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]
$\frac{1}{7}$	$\frac{1}{7}$	$\frac{4}{7}$	belegt			$\frac{1}{7}$

  - die Wahrscheinlichkeit, dass die Kette verlängert wird, ist 5/7

- Quadratisches Sondieren

Quadratisches Sondieren: $s(j, k) = (\lceil j/2 \rceil)^2 \cdot (-1)^j$



- $(h(k) - s(j, k)) \bmod m$

Vergleich: Überlauflisten / Open Hashing

Überlauflisten

- Vorteil: Variable Größe
- Nachteil: Referenzen für Listen benötigen zusätzlichen Speicherplatz

Open Hashing

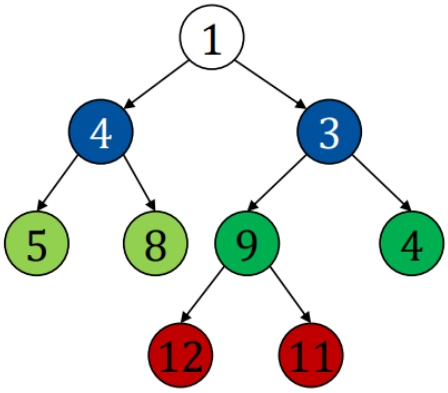
- Vorteile
  - gute Zugriffszeiten (quadratisches Sondieren)
  - benötigt keinen zusätzlichen Speicher
- Nachteile
  - feste Größe
  - nicht immer werden alle Felder belegt (quadratisches Sondieren)

-
- Dynamisches Hashing
  - Erweitere die Tabelle
    - Dynamische Hash-Tabelle
      - sinnvoll für Datensätze von variabler Größe
      - wird eingesetzt um z.B. Datenbanken in externem Speicher zu verwalten oder für Daten in virtuellen Speicher
      - wird Hash-Datei genannt (statt Hash-Tabelle)
    - Idee
      - teile Daten in Blöcke
      - Hash-Datei mit t Blöcken: Hash-Funktion h_t weist einem Schlüssel k die Nummer des Blocks zu, der den Datensatz enthält
      - wenn sich die Anzahl von Blöcken ändert
        - keine globale Re-Organisation
        - neue Hash-Funktion $h_{t'}$ wird für neue Einträge verwendet
  -

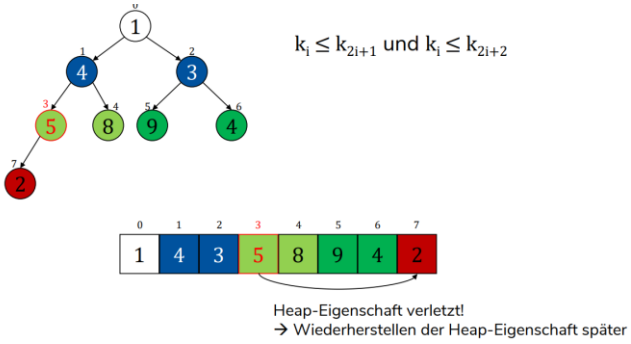
Großübung

- Binärbaum ist 2-beschränkt aber darf auch weniger haben
- Suchbaum links immer kleinerer Wert und rechts immer größer (Beachte Back to the roots)
- AVL-Baum auf Balance Faktor schauen

Heapsort



- Beispiel von einem Heap (ist kein Suchbaum):
- Heap Datenstruktur mit Operationen zum
  - Einfügen (add)
  - Löschen (removeMin)
  - Abfragen (size, isEmpty, getMin)
- Der Schlüssel eines Elternknotens ist immer **kleiner** als die Schlüssel **aller** Kindknoten
  - Elternknoten sind grundsätzlich kleiner als die Kindknoten
  - Kindknoten demnach immer größer als Elternknoten



- Baum in ein Array:
  - Implementierung: Array (Effizienz)
  - Interpretation: Binärbaum (Intuition)
- Herausforderung: Halte jederzeit die Heap Eigenschaft ein!
  - beim Löschen der Wurzel
  - beim Einfügen eines neuen Elements
- Heapify() Methode läuft von der Mitte bis zur Wurzel (Blätter erfüllen nämlich schon die Heap Eigenschaft)
  - Immer mit dem kleineren Element tauschen
- Priority Queue …
- Heapsort
  - MaxHeap -> höchster Wert in der Wurzel
  - entferne nach und nach immer wieder das größte Element und speichere die Elemente absteigend in einem Array
  - Aufwand immer O(n log n) (Sowohl bei M bzw. Vertauschen, als auch bei C)
    - Analog zu Quick- und Mergesort
    - Geht nicht schneller grundsätzlich, weil das Problem so komplex ist (bei vergleichsbasierten Sortierefahren)

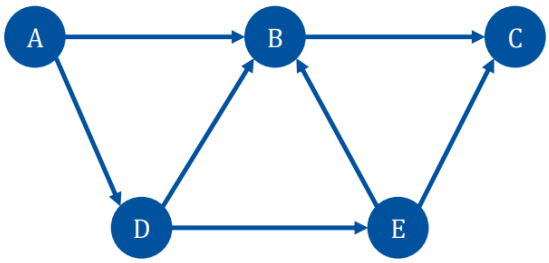
Graphen

- Eigenschaften

- Benachbart (Knoten, die nebeneinander sind)
- Gerichtet (Zwischen zwei Knoten gibt es nur eine Kante)
- Ungerichtet (Mehrere Kanten zwischen zwei Knoten z.B. oder Schleife bei einem Knoten)
- Ein Weg (Von A kann man B erreichen)
- Erreichbar (Wenn es einen Weg gibt)
- Abstand (Summe aller Kanten von A nach B)
- Geschlossen (Kreis, Zyklus)
  - Trivialer Zyklus = Schleife bei Weg mit Länge 1
- Zusammenhängend ⇔ Nicht zusammenhängend
- Knotengrad
  - Eingangsgrad
  - Ausgangsgrad
- Vollständig (Jeder Knoten ist mit jedem anderen Knoten verbunden über Kanten)
- Mehrfachkanten (Mehrere Kanten zwischen zwei Knoten)
- Baum (zyklenfrei, Eingangsgrad muss einmal 0 sein (Wurzel), gibt genau einen Weg nach unten)
- Beschränkt (2-beschränkter Baum ist Binärbaum, Anzahl an Knoten an einem Knoten)

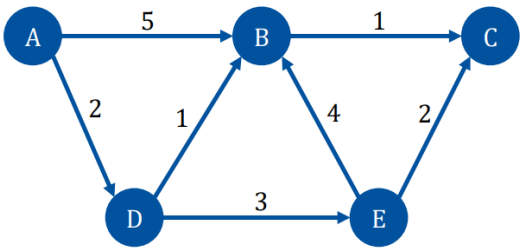
- Implementierung
  - Adjazenzliste
    - Benachbarte Knoten
  - Menge von Knotenobjekten
    - ausgehende Kanten von jedem Knoten
  - Adjazenzmatrix
    -

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	0	0
C	0	0	0	0	0
D	0	1	0	0	1
E	0	1	1	0	0



    - Erweiterung sehe so aus
    -

	A	B	C	D	E
A	0	5	0	2	0
B	0	0	1	0	0
C	0	0	0	0	0
D	0	1	0	0	3
E	0	4	2	0	0



  - Adjazenzliste
    - geringer Speicherbedarf bei vielen Knoten und wenigen Kanten
    - hohe Zugriffszeiten bei vielen Kanten
  - Adjazenzmatrix
    - schneller Zugriff auf Kanten
    - hoher Speicherbedarf bei vielen Knoten und wenigen Kanten
- Topologische Sortierung
  - Vor.: gerichteter, zyklenfreier Graph
  - Durchnummerieren von Knoten

    - Existiert keine topologische Reihenfolge, dann hat er einen Zyklus
    - Existiert eine topologische Reihenfolge, dann hat er keinen Zyklus

Graphen II

- Traversieren ist strukturiertes Durchlaufen eines Graphen
- Tripel-Algorithmus
    - Alle Knoten werden durchnummeriert
    - Kürzester Weg zwischen allen Knotenpaaren
    - Worst-Case O(Knoten^2) bzw. O(Knoten^3)
- Minimale Spannbäume (MST)
    - Alle Knoten enthält und erreicht der Spannbaum
    - So wenig Strecke wie möglich
    - Jeder Knoten kann die Wurzel sein
    - Algorithmus von Prim (ist ein Greedy-Verfahren)
        - Beliebiger Knoten wird gewählt
        - Wähle eine Kante, die minimales Gewicht hat und einen neuen Knoten erreicht bzw. mit dem beliebigen Knoten verbindet
        - Es wird also keine Kante jemals entfernt
        - Bei gleichem Gewicht ist es egal welche man wählt
        - Aufwand
            - bei Adjazentmatrix: O (Knoten^2)
            - bei Adjazenzliste: O (Ecken * log (Kanten))
- Allgemeine Lösungsverfahren
    - Backtracking
        - Gehe den Weg zurück, wenn eine (Teil-)Lösung nicht zum Erfolg führt
    - Greedy-Verfahren
        - Entscheidungen werden auf Basis von bis dahin gesammeltes Daten gefällt und nicht revidiert
        - Oft effizienter
        - Oft liefert es dadurch nicht die optimale Lösung