Script **generated by TTT**

Title: groh: profile1 (01.07.2016)

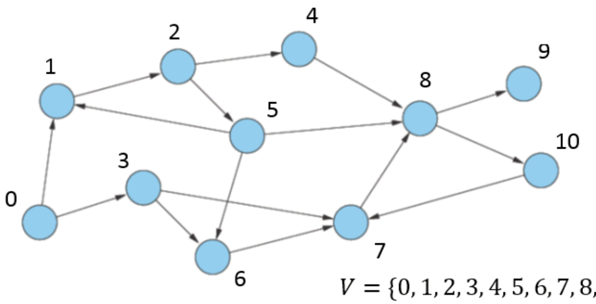
Date: Fri Jul 01 13:03:50 CEST 2016

Duration: 89:00 min

Pages: 50

Anwendung für Queue: Breadth First Traversierung eines Graphen

- Zur Vereinfachung: Integer-Codierung der Knoten:



$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

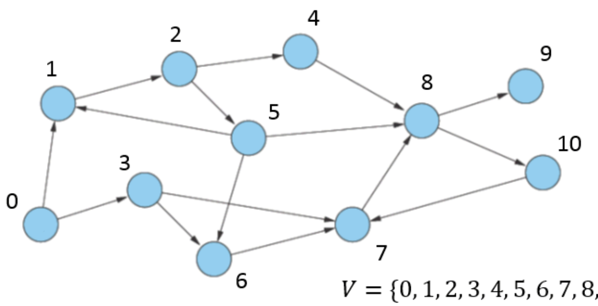
$E = \{(0,1), (1,2), (2,5), \dots, (8,10)\}$

- Annahme: Es existiere Klasse `Graph`, die entsprechende Methoden zum Speichern und Zugreifen von Elementen des Graphen bereitstellt.
- Annahme: Es existiere Klasse `Queue<Integer>`, die eine Queue von Integers (←→ Knoten) implementiert.

der nachfolgende Code ist aber zur besseren Lesbarkeit mit in statt mit Integer formuliert

Anwendung für Queue: Breadth First Traversierung eines Graphen

- Zur Vereinfachung: Integer-Codierung der Knoten:



$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$E = \{(0,1), (1,2), (2,5), \dots, (8,10)\}$

- Annahme: Es existiere Klasse `Graph`, die entsprechende Methoden zum Speichern und Zugreifen von Elementen des Graphen bereitstellt.
- Annahme: Es existiere Klasse `Queue<Integer>`, die eine Queue von Integers (←→ Knoten) implementiert.

der nachfolgende Code ist aber zur besseren Lesbarkeit mit in statt mit Integer formuliert

63

Fehler auf alter Version von Slides

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

falsch

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = g.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[nodesAdjacentTo[w]]) {
            parentOf[nodesAdjacentTo[w]] = v;
            distTo[nodesAdjacentTo[w]] = distTo[v] + 1;
            marked[nodesAdjacentTo[w]] = true;
            queue.enqueue(nodesAdjacentTo[w]);
        }
    }
}
```

richtig

Fehler auf alter Version von Slides

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

falsch

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = g.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[nodesAdjacentTo[w]]) {
            parentOf[nodesAdjacentTo[w]] = v;
            distTo[nodesAdjacentTo[w]] = distTo[v] + 1;
            marked[nodesAdjacentTo[w]] = true;
            queue.enqueue(nodesAdjacentTo[w]);
        }
    }
}
```

richtig

Fehler auf alter Version von Slides

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

falsch

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = g.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[nodesAdjacentTo[w]]) {
            parentOf[nodesAdjacentTo[w]] = v;
            distTo[nodesAdjacentTo[w]] = distTo[v] + 1;
            marked[nodesAdjacentTo[w]] = true;
            queue.enqueue(nodesAdjacentTo[w]);
        }
    }
}
```

richtig

Fehler auf alter Version von Slides

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

falsch

```java
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = g.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[nodesAdjacentTo[w]]) {
            parentOf[nodesAdjacentTo[w]] = v;
            distTo[nodesAdjacentTo[w]] = distTo[v] + 1;
            marked[nodesAdjacentTo[w]] = true;
            queue.enqueue(nodesAdjacentTo[w]);
        }
    }
}
```

richtig

Rekursion

- **Rekursion**:
  - Definiere eine Funktion durch sich selbst bzw.
  - formuliere Lösung eines Problems durch Rückgriff auf die Lösung selbst: Gebe Lösung für Basisfälle an und gib Regeln an, wie sich allg. Problem in Richtung der Basisfälle aufteilen lässt.
  - d.h. bspw. durch eine Methode, die sich selbst wieder aufruft

Bsp:

- Fakultät $$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$
- Fibonacci $$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ f(n-1) + f(n-2) & \text{if } n > 2. \end{cases}$$

Rekursion

- **Rekursion**:
  - Definiere eine Funktion durch sich selbst bzw.
  - formuliere Lösung eines Problems durch Rückgriff auf die Lösung selbst: Gebe Lösung für Basisfälle an und gib Regeln an, wie sich allg. Problem in Richtung der Basisfälle aufteilen lässt.
  - d.h. bspw. durch eine Methode, die sich selbst wieder aufruft

Bsp:

- Fakultät

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$

- Fibonacci

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ f(n-1) + f(n-2) & \text{if } n > 2. \end{cases}$$

Rekursion

- **Rekursion**:
  - Definiere eine Funktion durch sich selbst bzw.
  - formuliere Lösung eines Problems durch Rückgriff auf die Lösung selbst: Gebe Lösung für Basisfälle an und gib Regeln an, wie sich allg. Problem in Richtung der Basisfälle aufteilen lässt.
  - d.h. bspw. durch eine Methode, die sich selbst wieder aufruft

Bsp:

- Fakultät

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$

- Fibonacci

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ f(n-1) + f(n-2) & \text{if } n > 2. \end{cases}$$

92

Rekursion

- **Rekursion**:
  - Definiere eine Funktion durch sich selbst bzw.
  - formuliere Lösung eines Problems durch Rückgriff auf die Lösung selbst: Gebe Lösung für Basisfälle an und gib Regeln an, wie sich allg. Problem in Richtung der Basisfälle aufteilen lässt.
  - d.h. bspw. durch eine Methode, die sich selbst wieder aufruft

Fakultät

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \cdot n & \text{if } n > 0. \end{cases}$$

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```


Call Stack

Für jeden Methodenaufruf (auch Klassenmethoden oder nicht rekursive Methodenaufrufe) werden lokale Variablen + Parameter + Rücksprungadresse („von wem (von wo aus) wurde die Methode aufgerufen?“) auf Call-Stack gespeichert

```
public class someClass {

  int a;
  int b;

  int someMethodOne(int paramOne1, int paramOne2){
    int localOne = 5;
    int result = someMethodTwo(17) + localOne;
    return result;
  }

  int someMethodTwo(int paramTwo){
    int localTwo = 8;
    return paramTwo * localTwo;
  }
}
```

```
…
   SomeClass someObject = new SomeClass();
   int bbb = someObject.someMethodOne(12, 34);
…
```

…	…	…
2345	someObject	<2346>
2346	someObject.a	
2347	someObject.b	
…		

…	…	…
5467		int result = someMethodTwo(17) + localOne;
…		
5899		int bbb = someObject.someMethodOne(12, 34);
…		

…	…	…
6000	(Rücksprung-Adresse)	<5467>
6001	(Aufruf-Objekt)	<2346>
6002	paramTwo	17
6003	localTwo	8
6004	(Rücksprung-Adresse)	<5899>
6005	(Aufruf-Objekt)	<…>
6006	paramOne1	12
6007	paramOne2	34
6008	localOne	5
…	…	…

Call Stack

Für jeden Methodenaufruf (auch Klassenmethoden oder nicht rekursive Methodenaufrufe) werden lokale Variablen + Parameter + Rücksprungadresse („von wem (von wo aus) wurde die Methode aufgerufen?") auf Call-Stack gespeichert

```
public class someClass {

  int a;
  int b;

  int someMethodOne(int paramOne1, int paramOne2){
    int localOne = 5;
    int result = someMethodTwo(17) + localOne;
    return result;
  }

  int someMethodTwo(int paramTwo){
    int localTwo = 8;
    return paramTwo * localTwo;
  }
}
```

```
...
    SomeClass someObject = new SomeClass();
    int bbb = someObject.someMethodOne(12, 34);
...
```

...	...	...
2345	someObject	<2346>
2346	someObject.a	
2347	someObject.b	
...		

...	...	...
5467		int result = someMethodTwo(17) + localOne;
...		
5899		int bbb = someObject.someMethodOne(12, 34);
...		

...	...	...
6000	(Rücksprung-Adresse)	<5467>
6001	(Aufruf-Objekt)	<2346>
6002	paramTwo	17
6003	localTwo	8
6004	(Rücksprung-Adresse)	<5899>
6005	(Aufruf-Objekt)	<...>
6006	paramOne1	12
6007	paramOne2	34
6008	localOne	5
...	...	...

Call Stack

Für jeden Methodenaufruf (auch Klassenmethoden oder nicht rekursive Methodenaufrufe) werden lokale Variablen + Parameter + Rücksprungadresse („von wem (von wo aus) wurde die Methode aufgerufen?") auf Call-Stack gespeichert

```
public class someClass {

  int a;
  int b;

  int someMethodOne(int paramOne1, int paramOne2){
    int localOne = 5;
    int result = someMethodTwo(17) + localOne;
    return result;
  }

  int someMethodTwo(int paramTwo){
    int localTwo = 8;
    return paramTwo * localTwo;
  }
}
```

```
...
    SomeClass someObject = new SomeClass();
    int bbb = someObject.someMethodOne(12, 34);
...
```

...	...	...
2345	someObject	<2346>
2346	someObject.a	
2347	someObject.b	
...		

...	...	...
5467		int result = someMethodTwo(17) + localOne;
...		
5899		int bbb = someObject.someMethodOne(12, 34);
...		

...	...	...
6000	(Rücksprung-Adresse)	<5467>
6001	(Aufruf-Objekt)	<2346>
6002	paramTwo	17
6003	localTwo	8
6004	(Rücksprung-Adresse)	<5899>
6005	(Aufruf-Objekt)	<...>
6006	paramOne1	12
6007	paramOne2	34
6008	localOne	5
...	...	...

Call Stack

Für jeden Methodenaufruf (auch Klassenmethoden oder nicht rekursive Methodenaufrufe) werden lokale Variablen + Parameter + Rücksprungadresse („von wem (von wo aus) wurde die Methode aufgerufen?") auf Call-Stack gespeichert

```
public class someClass {

  int a;
  int b;

  int someMethodOne(int paramOne1, int paramOne2){
    int localOne = 5;
    int result = someMethodTwo(17) + localOne;
    return result;
  }

  int someMethodTwo(int paramTwo){
    int localTwo = 8;
    return paramTwo * localTwo;
  }
}
```

```
...
    SomeClass someObject = new SomeClass();
    int bbb = someObject.someMethodOne(12, 34);
...
```

...	...	...
2345	someObject	<2346>
2346	someObject.a	
2347	someObject.b	
...		

...	...	...
5467		int result = someMethodTwo(17) + localOne;
...		
5899		int bbb = someObject.someMethodOne(12, 34);
...		

...	...	...
6000	(Rücksprung-Adresse)	<5467>
6001	(Aufruf-Objekt)	<2346>
6002	paramTwo	17
6003	localTwo	8
6004	(Rücksprung-Adresse)	<5899>
6005	(Aufruf-Objekt)	<...>
6006	paramOne1	12
6007	paramOne2	34
6008	localOne	5
...	...	...

Call Stack

Für jeden Methodenaufruf (auch Klassenmethoden oder nicht rekursive Methodenaufrufe) werden lokale Variablen + Parameter + Rücksprungadresse („von wem (von wo aus) wurde die Methode aufgerufen?") auf Call-Stack gespeichert


```
public class someClass {

  int a;
  int b;

  int someMethodOne(int
    int localOne = 5;
    int result = someMethodTwo(17) + localOne;
    return result;
  }

  int someMethodTwo(int paramTwo){
    int localTwo = 8;
    return paramTwo * l
  }
}
```

```
...
    SomeClass someObjec
    int bbb = someObjec
...
```

Speicherbereich für Objekte: „Heap"

Einfügeende des Call Stacks →

„Call Stack"

...	...	...
2345	someObject	<2346>
2346	someObject.a	
2347	someObject.b	
...		

...	...	...
5467		int result = someMethodTwo(17) + localOne;
...		
5899		int bbb = someObject.someMethodOne(12, 34);
...		

...	...	...
6000	(Rücksprung-Adresse)	<5467>
6001	(Aufruf-Objekt)	<2346>
6002	paramTwo	17
6003	localTwo	8
6004	(Rücksprung-Adresse)	<5899>
6005	(Aufruf-Objekt)	<...>
6006	paramOne1	12
6007	paramOne2	34
6008	localOne	5
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

aktiver Stack-Frame

lassen wir aus Übersichtsgründen weg

	(Rücksprung-Adresse)	...
	(Aufruf-Objekt)	...
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

aktiver Stack-Frame

lassen wir aus Übersichtsgründen weg

	(Rücksprung-Adresse)	...
	(Aufruf-Objekt)	...
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	2
	temp	0
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	0
	temp	0
	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	0
	temp	0
	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	0
	temp	0
	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	2
	temp	1
	n	3
	temp	0
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	n	3
	temp	2
	ccc	...
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
...
 int ccc = factorial(3);
...
```

	ccc	6
	...	...
...	...	...

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
…
 int ccc = factorial(3);
…
```

	ccc	6
	…	…
…	…	…

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
…
 int ccc = factorial(3);
…
```

	n	0
	temp	0
	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	…
	…	…
…	…	…

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
…
 int ccc = factorial(3);
…
```

	n	0
	temp	0
	n	1
	temp	0
	n	2
	temp	0
	n	3
	temp	0
	ccc	…
	…	…
…	…	…

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
…
 int ccc = factorial(3);
…
```

	n	1
	temp	1
	n	2
	temp	0
	n	3
	temp	0
	ccc	…
	…	…
…	…	…

Rekursive Methodenaufrufe – Call Stack

```
long factorial(int n) {
    long temp;
    if (n == 0) {
        return 1;
    } else {
        temp = factorial(n-1);
        return n * temp;
    }
}
```

```
…
 int ccc = factorial(3);
…
```

	n	1
	temp	1
	n	2
	temp	0
	n	3
	temp	0
	ccc	…
	…	…
…	…	…

Java Klasse String

- Klasse `String` verwaltet konstante Zeichenketten in Java
  (`StringBuffer`: Klasse für variable Zeichenketten)

```
String someString1 = "didumdidei";
String someString2 = new String("tirilitirilo");
boolean isEqual = someString1.equals(someString2);
int length2 = someString2.length();
char[] charArray2 = someString2.toCharArray();
StringBuffer someStringBuffer = new StringBuffer("arghh");
someStringBuffer.append("hhh");
```


Java Klasse String

- Klasse `String` verwaltet konstante Zeichenketten in Java
  (`StringBuffer`: Klasse für variable Zeichenketten)

```
String someString1 = "didumdidei";
String someString2 = new String("tirilitirilo");
boolean isEqual = someString1.equals(someString2);
int length2 = someString2.length();
char[] charArray2 = someString2.toCharArray();
StringBuffer someStringBuffer = new StringBuffer("arghh");
someStringBuffer.append("hhh");
```

117

Hashing

- allgemein: Annahme: jedes Element e einer Datenstruktur habe einen Schlüssel key(e) (analog zu Schlüssel bei Datenbanken)

  Bsp:
  Elemente: String-Objekte, → als key die Zeichenkette selbst oder Integer-Codierung davon benutzen
  Elemente: Fahrräder → als key Rahmennummer
  Elemente: Professoren → als key Personalnummer

- K: Menge aller Schlüssel eines Elementtyps
  Hash-funktion h: $K \rightarrow [0, m-1]$

- Wörterbuch (HashMap, assoziatives Array, Hashtabelle) S: speichert Elemente e ∈ E (oder Referenzen auf Elemente) unter ihrem Key key(e) in einem Array mit m Elementen.

- Operationen:
  - S.insert(Element e): Fügt e in S ein.
  - S.remove(Key k): Löscht e mit key(e)=k
  - S.find(Key k): Gibt e mit key(e)=k zurück (falls enthalten; sonst gib ⊥ zurück)

Hashing

- allgemein: Annahme: jedes Element e einer Datenstruktur habe einen Schlüssel key(e) (analog zu Schlüssel bei Datenbanken)

  Bsp:
  Elemente: String-Objekte, → als key die Zeichenkette selbst oder Integer-Codierung davon benutzen
  Elemente: Fahrräder → als key Rahmennummer
  Elemente: Professoren → als key Personalnummer

- K: Menge aller Schlüssel eines Elementtyps
  Hash-funktion $h\colon K \rightarrow [0, m-1]$

- Wörterbuch (HashMap, assoziatives Array, Hashtabelle) S: speichert Elemente e ∈ E (oder Referenzen auf Elemente) unter ihrem Key key(e) in einem Array mit m Elementen.

- Operationen:
  - S.insert(Element e): Fügt e in S ein.
  - S.remove(Key k): Löscht e mit key(e)=k
  - S.find(Key k): Gibt e mit key(e)=k zurück (falls enthalten; sonst gib ⊥ zurück)

Hashing

- allgemein: Annahme: jedes Element e einer Datenstruktur habe einen Schlüssel key(e) (analog zu Schlüssel bei Datenbanken)

  Bsp:
  Elemente: String-Objekte, → als key die Zeichenkette selbst oder Integer-Codierung davon benutzen
  Elemente: Fahrräder → als key Rahmennummer
  Elemente: Professoren → als key Personalnummer

- K: Menge aller Schlüssel eines Elementtyps
  Hash-funktion $h\colon K \rightarrow [0, m-1]$

- Wörterbuch (HashMap, assoziatives Array, Hashtabelle) S: speichert Elemente e ∈ E (oder Referenzen auf Elemente) unter ihrem Key key(e) in einem Array mit m Elementen.

- Operationen:
  - S.insert(Element e): Fügt e in S ein.
  - S.remove(Key k): Löscht e mit key(e)=k
  - S.find(Key k): Gibt e mit key(e)=k zurück (falls enthalten; sonst gib ⊥ zurück)

121

Hashing

- allgemein: Annahme: jedes Element e einer Datenstruktur habe einen Schlüssel key(e) (analog zu Schlüssel bei Datenbanken)

  Bsp:
  Elemente: String-Objekte, → als key die Zeichenkette selbst oder Integer-Codierung davon benutzen
  Elemente: Fahrräder → als key Rahmennummer
  Elemente: Professoren → als key Personalnummer

- K: Menge aller Schlüssel eines Elementtyps
  Hash-funktion $h\colon K \rightarrow [0, m-1]$

- Wörterbuch (HashMap, assoziatives Array, Hashtabelle) S: speichert Elemente e ∈ E (oder Referenzen auf Elemente) unter ihrem Key key(e) in einem Array mit m Elementen.

- Operationen:
  - S.insert(Element e): Fügt e in S ein.
  - S.remove(Key k): Löscht e mit key(e)=k
  - S.find(Key k): Gibt e mit key(e)=k zurück (falls enthalten; sonst gib ⊥ zurück)

121

Hashing

- Anforderungen an Hashfunktion:
  - platzsparend (bspw. u.a. im Idealfall surjektiv)
  - gute Streuung / Verteilung über Tabelle
  - effizient berechenbar
  - ...

- Idealfall: h in O(1) berechenbar und jedes Element e alleine unter Index h(key(e)) gespeichert → find, insert, remove in O(1) realisierbar

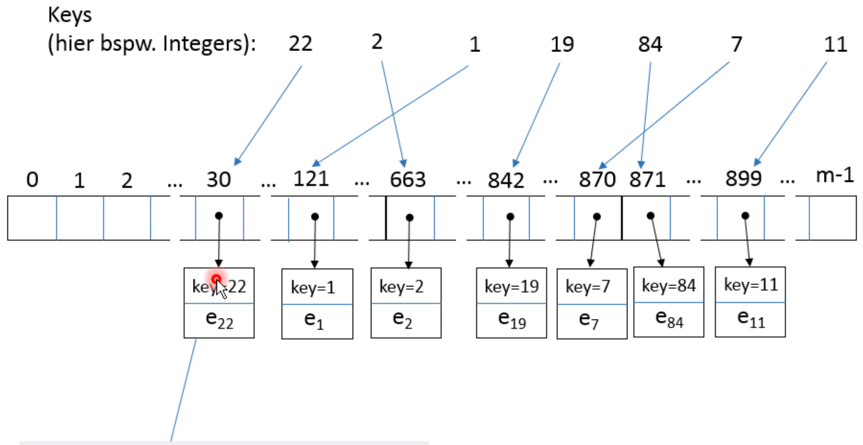
```
void insert(ElementType e){
    hashTable[h(key(e))] = e; (1)
}

void remove(Key k){
    hashTable[h(k)] = null;
}
```

```
Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

(1) Genauer müsste hier natürlich stehen: `hashTable[h(key(e))] = new Item(key(e),e);` dann sollte (um zweimaliges Berechnen zu vermeiden) key(e) natürlich zwischengespecihert werden.

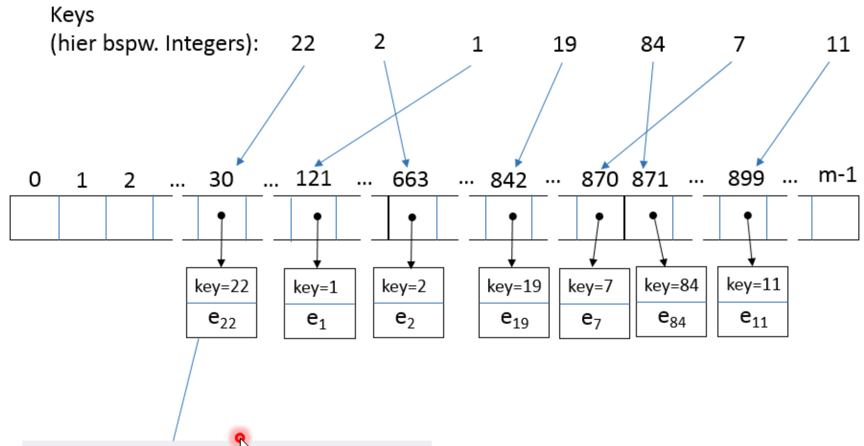
Hashing




```
class Item<ElementType,Key>{
    ElementType e;
    Key k;
}
```

je nach Zusammenhang verkürzend zur Verbesserung der Lesbarkeit: vereinfachend Items (e,k) mit Elementen e oder ihren keys k identifizieren

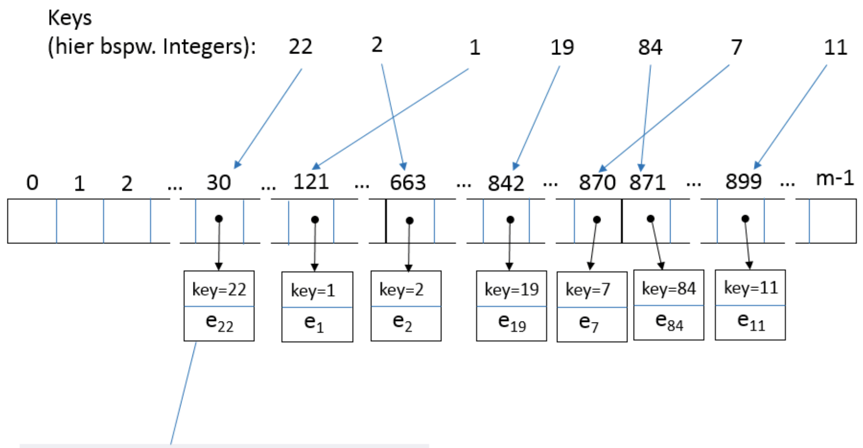
Hashing




```
class Item<ElementType,Key>{
    ElementType e;
    Key k;
}
```

je nach Zusammenhang verkürzend zur Verbesserung der Lesbarkeit: vereinfachend Items (e,k) mit Elementen e oder ihren keys k identifizieren

Hashing




```
class Item<ElementType,Key>{
    ElementType e;
    Key k;
}
```

je nach Zusammenhang verkürzend zur Verbesserung der Lesbarkeit: vereinfachend Items (e,k) mit Elementen e oder ihren keys k identifizieren

Hashing

- Anforderungen an Hashfunktion:
  - platzsparend (bspw. u.a. im Idealfall surjektiv)
  - gute Streuung / Verteilung über Tabelle
  - effizient berechenbar
  - ...
- Idealfall: h in O(1) berechenbar und jedes Element e alleine unter Index h(key(e)) gespeichert → find, insert, remove in O(1) realisierbar

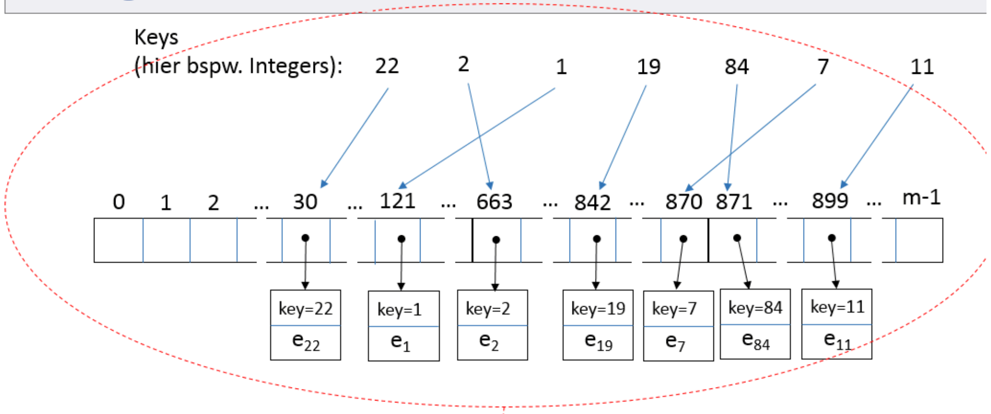
```
void insert(ElementType e){
    hashTable[h(key(e))] = e; (1)
}

void remove(Key k){
    hashTable[h(k)] = null;
}
```

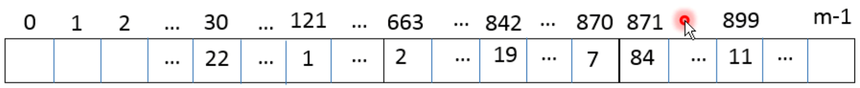
```
Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

(1) Genauer müsste hier natürlich stehen: `hashTable[h(key(e))] = new Item(key(e),e);` dann sollte (um zweimaliges Berechnen zu vermeiden) key(e) natürlich zwischengespeichert werden.

Hashing



Verkürzende Darstellung:



Hashing

- Anforderungen an Hashfunktion:
  - platzsparend (bspw. u.a. im Idealfall surjektiv)
  - gute Streuung / Verteilung über Tabelle
  - effizient berechenbar
  - ...

- Idealfall: h in O(1) berechenbar und jedes Element e alleine unter Index h(key(e)) gespeichert → find, insert, remove in O(1) realisierbar

```
void insert(ElementType e){
    hashTable[h(key(e))] = e; (1)
}

void remove(Key k){
    hashTable[h(k)] = null;
}
```

```
Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

(1) Genauer müsste hier natürlich stehen: `hashTable[h(key(e))] = new Item(key(e),e);` dann sollte (um zweimaliges Berechnen zu vermeiden) key(e) natürlich zwischengespeichert werden.

Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)

- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim i}_{\text{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} (1-\frac{i}{m})$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)

- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim i}_{\text{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} (1-\frac{i}{m})$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)

- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim } \mathrm{i}_{\mathrm{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} \left(1-\frac{i}{m}\right)$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)

- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim } \mathrm{i}_{\mathrm{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} \left(1-\frac{i}{m}\right)$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

125

Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)

- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim } \mathrm{i}_{\mathrm{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1} \left(1-\frac{i}{m}\right)$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

125