Tafelübung 09.5
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Organisatorisches

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

ICPC - Programmierwettbewerb, auch an der FAU

Think – Create – Solve

ICPC - Was ist das?

- International Collegiate Programming Contest – veranstaltet von der ACM
- dabei sollen Teams aus drei Studenten innerhalb von fünf Stunden neun bis elf knifflige und originelle Programmieraufgaben lösen
- Problem: nur ein Computer steht zur Verfügung, aber kein Internet ☹
- dreistufiger Wettbewerb mit *Local Contest* in Erlangen, *Regional Contest* irgendwo in Nordwesteuropa und *World Finals*

Local Contest
(Erlangen)

Regional Contest
(Delft)

World Finals
(Orlando)

ICPC an der FAU

- am Samstag, 28. Januar 2017 findet wieder ein FAU Local Contest statt
- von 10 bis 15 Uhr im Informatikhochhaus
- teilnehmen darf jede/r Student/in der FAU, Fachrichtung egal!
- es wird jeweils zu dritt programmiert (Einzelanmeldung möglich)

- Practice Session für alle Neulinge, bei der (einfache) typische Probleme gezeigt und erklärt werden
- Ort und Zeit werden noch bekannt gegeben
- mehr Infos/Anmeldung: `http://icpc.cs.fau.de`

Was bringt mir das Ganze?

- Spaß und Pizzabrötchen ☺
- Programmiererfahrung und Vertiefung gelernter Algorithmen
- jeder Teilnehmer erhält eine Urkunde
- im Winter sind die Probleme etwas einfacher, perfekt für Neueinsteiger
- beim Local Contest im Sommer werden die Teams bestimmt, die zum NWERC fahren dürfen, um unsere Uni zu vertreten
- die ganz Guten dürfen zu den World Finals (z.B. 2017 Rapid City (South Dakota/USA)) fahren, um unsere Uni zu vertreten

FAU-Teams können mithalten!

- FAU erfolgreichste deutsche Uni der letzten Jahre
  - 2010-2012 deutschlandweiten *Subregional* gewonnen (2013+2014: 3. Platz)
  - beim Regional Contest 2007-2014 insgesamt 9 Medaillen gewonnen (davon 6 Gold)
  - FAU durch 2. Platz beim NWERC 2014 als einziges deutsches Team zu den World Finals 2015 qualifiziert.
- 2003 (Beverly Hills), 2010 (Harbin), 2011 (Orlando), 2015 (Marrakesh) war ein Team der FAU zu den World Finals qualifiziert
- bisher größter Erfolg bei den World Finals:
  2011: Team deFAUlt, 7. Platz von 105 Teams
  (bzw. ca. 10 000 Teams in den Vorausscheiden)

Beispielproblem: „All in All“

You have devised a new encryption technique which encodes a message by inserting between its characters randomly generated strings in a clever way. To validate your method, however, it is necessary to write a program that checks if the message is really encoded in the final string.

Input Specification:
The input contains several test cases. Each is specified by two strings *s*, *t* of alphanumeric ASCII characters.

Output Specification:
For each test case output, if *s* is a subsequence of *t*, i.e. if you can remove characters from *t* such that the concatenation of the remaining characters is *s*.

Sample Input:	**Sample Output**:
`sequence subsequence`	`Yes`
`person compression`	`No`
`VERDI vivaVittorioEmanueleReDiItalia`	`Yes`
`caseDoesMatter CaseDoesMatter`	`No`

Wo kann ich trainieren?

- FAU Online Judge: `https://icpc.cs.fau.de/oj`
- einmaliges Freischalten mittels EST-Account notwendig
- mit Problemen z.B. vom Winter 2012
- Hilfestellung über das Online-Judge-Frontend bzw. IRC-Channel `#hallowelt` im IRCnet (z.B. `irc.uni-erlangen.de`)

- Codeforces: `http://codeforces.com`
- SPOJ: `http://spoj.pl`
- UVa: `http://uva.onlinejudge.org`

Generics

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Generics: Motivation

Szenario

In unserer Anwendung benötigen wir...

- eine Liste zur Speicherung von Integer-Werten
- eine Liste zur Speicherung von Strings
- eine Liste zur Speicherung von `WasAuchImmer`-Objekten

Schlechte Lösung (1)

drei Listen-Implementierungen $\rightsquigarrow$ viel duplizierter Code ↯

Schlechte Lösung (2)

Listen-Elemente vom Typ `Object` $\rightsquigarrow$ keine Typsicherheit ↯

Generics: Grundlagen

- Generics $\equiv$ *generische Typen*
  - Abstraktion von den zu Grunde liegenden Typen
  - Erstellung eines Datentyps unter Angabe eines Typparameters
    - Typparameter als „Platzhalter“ für konkreten Typ
    - sog. Typparametrisierung
  - Verwendung eines generischen Typs $\rightsquigarrow$ Angabe der Typparameter

  $\rightsquigarrow$ stellt Typsicherheit schon zur Übersetzungszeit sicher

Im Listen-Beispiel

Eine Listen-Implementierung mit Typparameter für Typen der zu speichernden Objekte.

Generics in Java

Typparametrisierbare Listen-Implementierung

```
public class List<T> {
    // ...
    public void add(T item) { /* ... */ }
    public T get(int idx) { /* ... */ }
}
```

Anmerkungen

- `List` heißt generischer Typ
- `T` heißt Typparameter
- `T` kann innerhalb `List` als „ganz normaler Typ“ verwendet werden
  - allerdings: `T` kann *nicht* zur Objekt-Instanziierung verwendet werden

Verwendung der Listen-Implementierung

```
List<Integer> integers = new List<Integer>();
```

Einschränkung des Typparameters

- mögliche Typen für Typparameter können eingeschränkt werden (*Bounds*)
  - nur Klassen, die von bestimmter Klasse erben
  - nur Klassen, die bestimmtes Interface implementieren
- Nutzen? Nutzen!
  - stellt „Mindestanforderung“ an Typen dar
  - konkrete Typen haben „mindestens“ die entsprechende Schnittstelle

  $\rightsquigarrow$ erlaubt den Aufruf dieser Methoden, ohne konkreten Typ zu kennen
- in Java: Einschränkung des Typparameters mittels `extends`
  - das gilt auch für Interfaces (kein `implements`)...

Beispiele

```
class Foo<T extends Exception> { /* ... */ }
class Bar<T extends Comparable<T>> { /* ... */ }
```

Statische generische Methoden (1)

Folgender Code kompiliert nicht

```
public class Foo<T> {
  public Foo(T param) { /* ... */ }
  public static Foo<T> getFoo(T param) {
    if (param == null) { /* Fehlerbehandlung */ }
    return new Foo<T>(param);
  }
}
```

Fehlermeldung beim Übersetzen

```
cannot make a static reference to the non-static type T
```

Was ist da schiefgegangen?

- bei der Instanziierung von Foo wird ein konkreter Typ für T eingesetzt
- aber: Klassenmethoden sind nicht an ein Objekt gebunden
  - $\rightsquigarrow$ keine Instanziierung hat stattgefunden
  - $\rightsquigarrow$ für T existiert kein konkreter Typ

Statische generische Methoden (2)

Folgender Code kompiliert nicht

```
public static Foo<T> getFoo(T param) { /* ... */ }
```

Lösung

- bei der Methodendeklaration den Typparameter mit angeben
- Compiler kann dann beim Methodenaufruf einen passenden Typ einsetzen
  - dabei werden Typparameter aus den Argumenttypen inferiert

... im Beispiel:

```
public static <T> Foo<T> getFoo(T param) { /* ... */ }
//================================================
Foo<Integer> intFoo = Foo.getFoo(4711);
```

Typlöschung

- Generics wurden erst nachträglich zu Java hinzugefügt
  - dabei wurde Augenmerk auf (Bytecode-)Kompatibilität gelegt

⇝ Java-Laufzeitsystem kennt keine Generics im Typsystem

- deswegen notwendig: Typlöschung (*Type Erasure*)
  - Typparameter werden durch `Object` ersetzt („ausradiert“)
    - bzw. bei *Bounds* durch spezifischere Typen
  - explizite Typkonvertierungen werden in den Code eingefügt

Also...

Bei der Übersetzung wird der Code generiert, den wir auch selbst hätten schreiben können („Schlechte Lösung (2)“). Die Typlöschung passiert aber erst *nach* der Typprüfung im Compiler, deswegen kann Typsicherheit zur Übersetzungszeit garantiert werden ☺

Beispiel für Typlöschung

Code vor Typlöschung

```
class Foo<T> {
  T value;

  public Foo(T value) {
    this.value = value;
  }

  public T get() {
    return this.value
  }
}

// ====================================

Foo<Integer> f = new Foo<Integer>(13);
Integer v = f.get();
```

Code nach Typlöschung

```
class Foo {
  Object value;

  public Foo(Object value) {
    this.value = value;
  }

  public Object get() {
    return this.value
  }
}

// ====================================

Foo f = new Foo(13);
Integer v = (Integer) f.get();
```

Konsequenzen der Typlöschung (1)

Beispiel 1

Auf Grund der Typlöschung ist Überladen mit Typparametern nicht möglich. Folgender Code würde zu undefiniertem Verhalten führen und ist deshalb „verboten“:

```
class Foo<T> {
  public void bar(T x) {
    /* ... */
  }

  public void bar(Object x) {
    /* ... */
  }
}
```

Konsequenzen der Typlöschung (2)

Beispiel 2

Zur Laufzeit stehen keine Typinformationen über Typparameter zur Verfügung. Folgender Code führt deshalb zu einem Fehler zur Übersetzungszeit:

```
class Foo<T> {
  /* ... */
}

// ========================

if (f instanceof Foo<Integer>) {
  /* ... */
} else if (f instanceof Foo<String>) {
  /* ... */
} else {
  /* ... */
}
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo<T> { /* ... */ }

//====================================

Foo<Integer>[] foos = new Foo<Integer>[1];
```

- Arrays mit generischen Typen sind in Java problematisch...
  - Verwendung würde zu unerwartetem Verhalten führen (s.u.) ↯
  - Grund dafür ist wieder die Typlöschung
- mögliche Lösungen sind eher *Hacks* als saubere Lösungen...
  - Compiler-Warnungen ignorieren/unterdrücken
  - „Hilfsklassen“ ohne Typparameter erstellen und verwenden

Unerwartetes Verhalten (I)

Beispiel ohne Generics

```
String[] strings = new String[1];
Integer[] ints = new Integer[1];

ints[0] = 42; // Autoboxing von int nach Integer

Object[] objects = strings;
objects[0] = ints[0]; // -> ArrayStoreException

String s = strings[0];
```

Erläuterung

Ein `String[]` ist auch ein `Object[]`. Der Code führt allerdings zu einer `ArrayStoreException`, sobald man versucht, ein „falsches“ Element abzuspeichern.

Unerwartetes Verhalten (II)

Beispiel mit Generics (**Annahme**: Code würde so übersetzt werden können)

```
ArrayList<String>[] strings = new ArrayList<String>[1];
ArrayList<Integer>[] ints = new ArrayList<Integer>[1];

ints[0] = new ArrayList<Integer>();
ints[0].add(42); // Autoboxing von int nach Integer

Object[] objects = strings;
objects[0] = ints[0];

String s = strings[0].get(0); // -> ClassCastException
```

Erläuterung

Der Code würde in einer `ClassCastException` resultieren, sobald man lesend auf ein „falsches“ Element zugreift. Eigentlich erwartet man aber schon beim Abspeichern eine `ArrayStoreException`.

Mögliche Lösungen

Lösung 1: Compiler-Warnungen unterdrücken

```
@SuppressWarnings("unchecked")
Foo<Integer>[] foos = new Foo[1];
```

Lösung 2: „Hilfsklasse“ erzeugen und verwenden

```
class Foo_Integer extends Foo<Integer> {
  /* intentionally left blank */
};

Foo_Integer[] foos = new Foo_Integer[1];
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo<T> {
  public void bar() {
    T[] array = new T[1];
  }
}
```

- Grund für den Fehler ist erneut die Typlöschung
  - T ist zur Laufzeit nicht bekannt
  - Java benötigt allerdings Basistyp zum Erzeugen eines Arrays ↯

Mögliche Lösungen

Lösung 1: Compiler-Warnungen beim Zugriff/Cast unterdrücken

```
Object[] array = new Object[1];
@SuppressWarnings("unchecked")
T elem = (T) array[0];
```

Lösung 2: Compiler-Warnungen bei der Erzeugung unterdrücken (**besser!**)

```
@SuppressWarnings("unchecked")
T[] array = (T[]) new Object[1];
```

Wildcards – Motivation

Folgender Code compiliert nicht

```
class Bar<T> { /* ... */ }

//===================================

Bar<Integer> bInt = new Bar<Integer>();
Bar<Object> bObj = bInt; // -> incompatible types
```

- Grund ist erneut – oh Wunder ☺ – die Typlöschung
  - wäre es erlaubt: Gefährdung der Typsicherheit ↯

Problem? Problem!

Szenario

Eine Funktion soll einen Parameter eines generischen Typs haben. Der Funktion ist es aber „egal“, welcher Typparameter tatsächlich verwendet wurde; insbesondere soll der Parameter auf beliebige Typparameter passen.

Folgender Code compiliert nicht

```
class Bar<T> { /* ... */ }

class Foo {
  void func1() {
   Bar<Integer> bInt = new Bar<Integer>();
   func2(bInt);
   // func2([...]) [...] cannot be applied to func2([...])
  }
  void func2(Bar<Object> p) { /* ... */ }
}
```

Wildcards

- das *Wildcard*-Symbol ? dient als Platzhalter für beliebige Typen
  - damit ist `Foo<?>` Basistyp „aller `Foos`“
  - es gehen damit aber auch Typinformationen verloren...

  ⇝ ganz grob: Lesen erlaubt, Schreiben verboten

- eine *Wildcard* kann mit Hilfe von *Bounds* eingeschränkt werden
  - `? extends Foo`
    - beliebiger Typ, aber „mindestens `Foo`“
  - `? super Foo`
    - beliebiger Typ, aber nur Oberklassen von `Foo` (inklusive `Foo`)

Lösung für unser Problem

Verwendung einer *Wildcard*

```
class Bar<T> { /* ... */ }

class Foo {
  void func1() {
    Bar<Integer> bInt = new Bar<Integer>();
    func2(bInt);
  }
  void func2(Bar<?> p) { /* ... */ }
}
```

Achtung...

Compiliert

```
List<?> list1 = new LinkedList<String>();
```

Compiliert nicht

```
List<List<?>> list2 = new LinkedList<List<String>>();
// list2 ist deklariert als Liste, deren Elemente Listen jedes
//   beliebigen Typs sind
// erzeugt wird jedoch eine Liste, deren Elemente Listen von
//   Strings sind
```

Compiliert

```
List<? extends List<?>> list3 = new LinkedList<List<String>>();
```

Bäume

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Graphen

- Graph $\equiv$ dynamische Datenstruktur

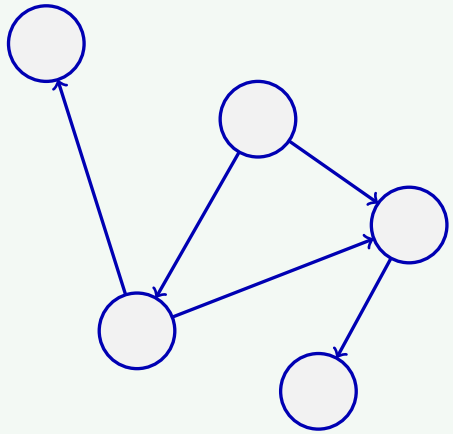
$\rightsquigarrow$ erlaubt die Speicherung beliebig vieler Elemente

- ein Graph besteht aus:
  - Knoten $\rightsquigarrow$ Repräsentation der Elemente
  - Kanten $\rightsquigarrow$ Repräsentation der Beziehungen zwischen Elementen
    - Kanten können gerichtet oder ungerichtet sein

- jeder Knoten hat einen...
  - Eingangsgrad: Anzahl der eingehenden Kanten
  - Ausgangsgrad: Anzahl der ausgehenden Kanten

Graphen: Beispiel

Beispiel für einen (gerichteten) Graphen

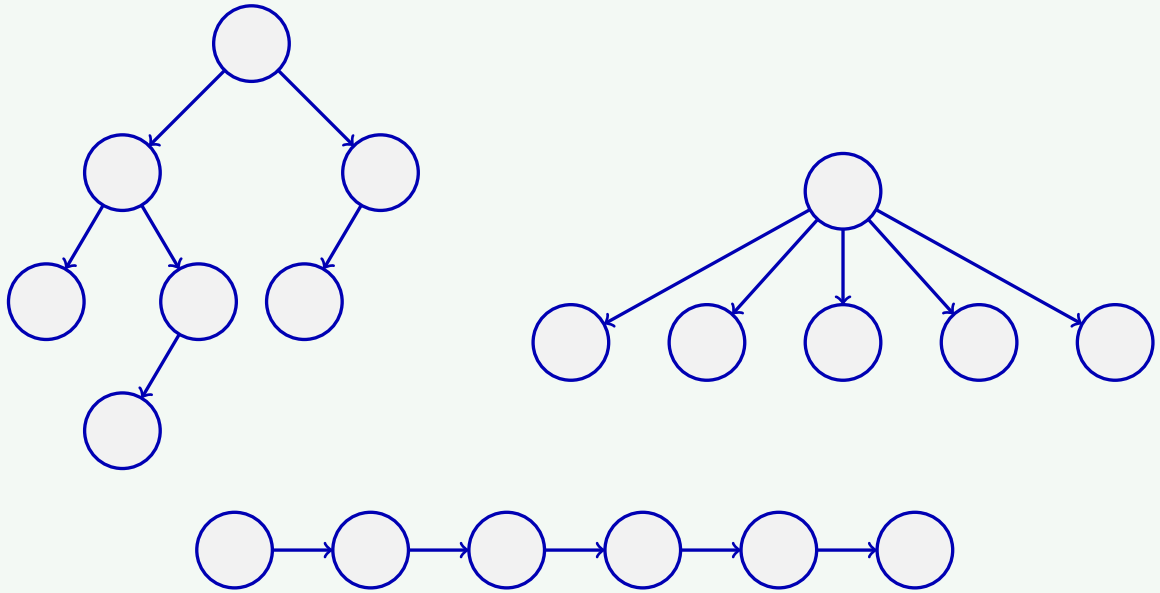


Bäume

- Baum $\equiv$ spezieller Graph
  - gerichteter Baum:
    - zusammenhängender, gerichteter und kreisfreier Graph mit genau einer Wurzel (Knoten mit Eingangsgrad 0) und einer Monohierarchie
  - ungerichteter Baum:
    - zusammenhängender, ungerichteter und kreisfreier Graph
- auf Grund ihrer Eigenschaften:
  - Darstellung von hierarchischen Beziehungen zwischen Elementen

Bäume: Beispiele

Beispiele für Bäume

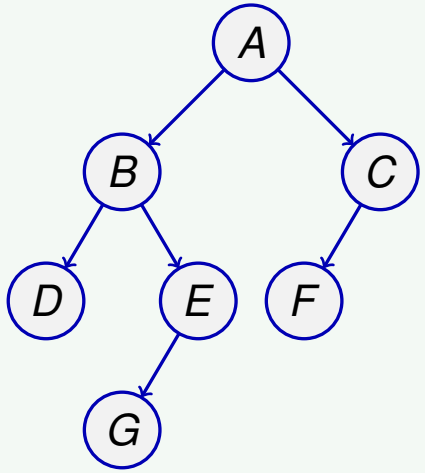


Bäume: Arten von Beziehungen (I)

- für gerichtete Bäume gilt: ein Knoten X ist...
  - Vater, Mutter, oder direkter Vorgänger eines Knoten Y,
    - falls es eine Kante von X nach Y gibt
  - Kind oder direkter Nachfolger eines Knoten Y,
    - falls es eine Kante von Y nach X gibt
  - Geschwisterknoten („*sibling*“) eines Knoten Y,
    - falls X und Y denselben direkten Vorgänger haben
  - indirekter Vorgänger eines Knoten Y,
    - falls es einen Pfad von X nach Y gibt
  - indirekter Nachfolger eines Knoten Y,
    - falls es einen Pfad von Y nach X gibt
  - ein Blatt, falls X keine Kinder hat
  - ein innerer Knoten, falls X mindestens ein Kind hat

Bäume: Arten von Beziehungen (II)

Beispiel



Beziehungen zwischen den Knoten

- *A* ist direkter Vorgänger von *B* und *C*
- *G* ist direkter Nachfolger von *E*
- *B* und *C* sind Geschwisterknoten
- *A* ist indirekter Vorgänger von allen anderen Knoten
- *G* ist indirekter Nachfolger von *A*, *B* und *E*
- *D*, *F* und *G* sind Blätter
- *A*, *B*, *C* und *E* sind innere Knoten

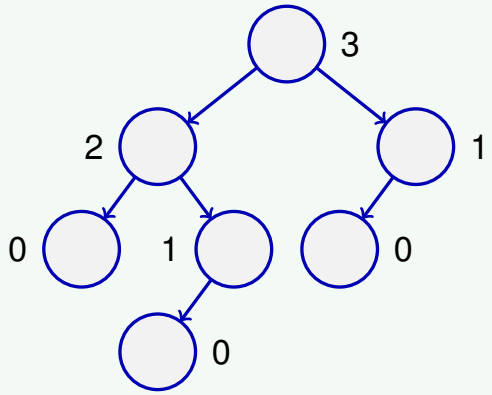
Höhe eines Knotens/Baumes

- Höhe eines Knotens:
  - Anzahl der Kanten des *längsten* Pfades zu einem Blatt

⇝ Blätter haben immer eine Höhe von 0

- Höhe eines Baumes:
  - Höhe seiner Wurzel

Beispiel

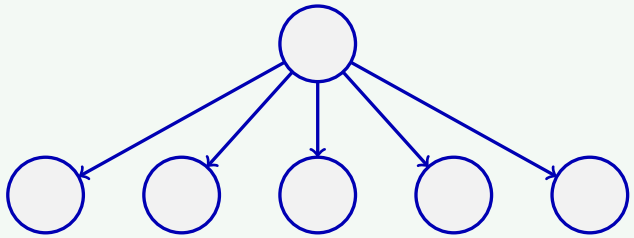


Verzweigungsgrad

- Verzweigungsgrad eines Baumes:
  - maximal erlaubte Anzahl an Nachfolgern für jeden Knoten

$\rightsquigarrow$ maximal erlaubter Ausgangsgrad

Beispiel für einen Baum mit Verzweigungsgrad **5**



Baumtraversierung

- Traversierung eines Baumes:
  - alle Knoten des Baumes ausgehend von der Wurzel „besuchen“
- man unterscheidet verschiedene Besuchsreihenfolgen
  - Tiefensuche (DFS, *depth-first search*):
    - Kinder vor „Geschwistern“ besuchen

    $\rightsquigarrow$ noch zu besuchende Knoten auf einen Stack ablegen
  - Breitensuche (BFS, *breadth-first search*):
    - „Geschwister“ vor Kindern besuchen

    $\rightsquigarrow$ noch zu besuchende Knoten in eine Queue ablegen
- Tiefen- und Breitensuche auch auf beliebigen Graphen möglich
  - dann aber notwendig: Markierung der bereits besuchten Knoten

Tiefensuche

Tiefensuche mit explizitem Stack

```
funktion dfs(G: Graph) {
    S := leerer Stack;
    fuege Wurzel von G in S ein;
    solange S nicht leer ist fuehre aus {
        a := entferne oberstes Element von S;
        bearbeite Knoten a;
        fuer alle Nachfolger n von a fuehre aus {
            lege n oben auf S;
        }
    }
}
```

Tiefensuche

Tiefensuche unter Verwendung des Aufruf-Stacks

```
funktion dfs(G: Graph) {
    dfs_helper(Wurzel von G);
}

funktion dfs_helper(k: Knoten) {
    bearbeite k;
    fuer alle Nachfolger n von k fuehre aus {
        dfs_helper(n);
    }
}
```

Breitensuche

Breitensuche mit Queue

```
funktion bfs(G: Graph) {
    Q := leere Queue;
    fuege Wurzel von G in Q ein;
    solange Q nicht leer ist fuehre aus {
        a := entferne vorderstes Element aus Q;
        bearbeite Knoten a;
        fuer alle Nachfolger n von a fuehre aus {
            fuege n hinten in Q ein;
        }
    }
}
```

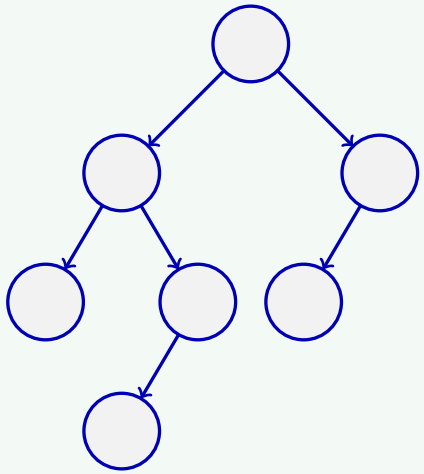
Binärbäume

- Binärbaum $\equiv$ Baum mit Verzweigungsgrad 2

$\rightsquigarrow$ jeder Knoten hat maximal zwei Nachfolger

$\rightsquigarrow$ Nachfolger werden als linkes und rechtes Kind bezeichnet

Beispiel für einen Binärbaum



Traversierung von Binärbäumen

- bei der Traversierung von Binärbäumen mittels Tiefensuche:
  - drei Möglichkeiten, wann Knoten „bearbeitet“ wird:
    - pre-order:
      - bevor das erste Kind besucht wird
    - in-order:
      - bevor das zweite Kind besucht wird
    - post-order:
      - nachdem das zweite Kind besucht wurde

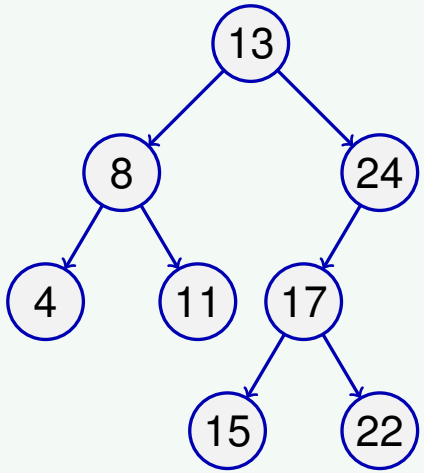
Traversierung von Binärbäumen

Tiefensuche auf Binärbäumen

```
funktion dfs_helper(k: Knoten) {
    bearbeite k; // pre-order

    falls k linkes Kind hat fuehre aus {
        dfs_helper(linkes Kind von k);
    }

    bearbeite k; // in-order

    falls k rechtes Kind hat fuehre aus {
        dfs_helper(rechtes Kind von k);
    }

    bearbeite k; // post-order
}
```

Beispiel für die Traversierungsarten

Beispiel für die Traversierungsarten



pre-order:	13	8	4	11	24	17	15	22
in-order:	4	8	11	13	15	17	22	24
post-order:	4	11	8	15	22	17	24	13

Binäre Suchbäume

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

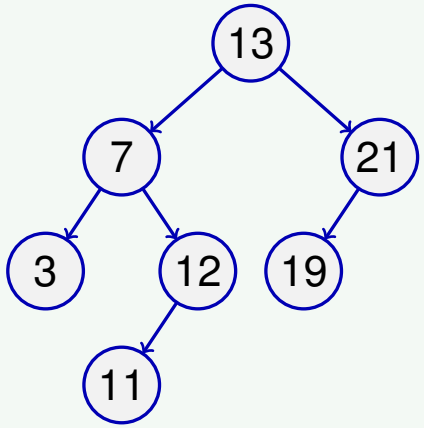
TECHNISCHE FAKULTÄT

Binäre Suchbäume

Binärer Suchbaum

Ein Binärer Suchbaum ist ein Binärbaum, der die Suchbaumeigenschaft erfüllt: Für jeden Knoten mit Wert X im Baum gilt, dass (bzgl. einer beliebigen, aber festen Ordnungsrelation) der linke Teilbaum nur kleinere Werte als X und der rechte Teilbaum nur größere Werte als X beinhaltet.

Beispiel für einen Binären Suchbaum (bzgl. natürlicher Ordnung)

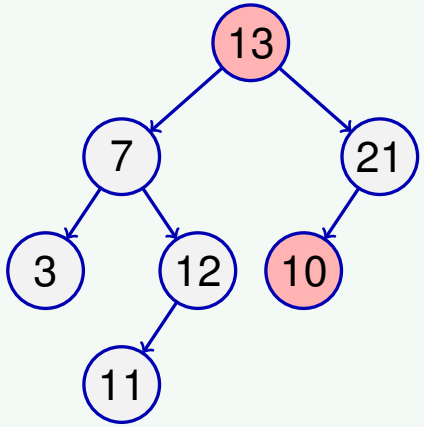


Binäre Suchbäume

Achtung...

Es reicht nicht aus, nur die direkten Kinder jedes Knotens auf die Suchbaumeigenschaft hin zu untersuchen – dies stellt eine notwendige Bedingung, aber keine hinreichende Bedingung für einen Suchbaum dar.

Beispiel: Kein Suchbaum (bzgl. natürlicher Ordnungsrelation)

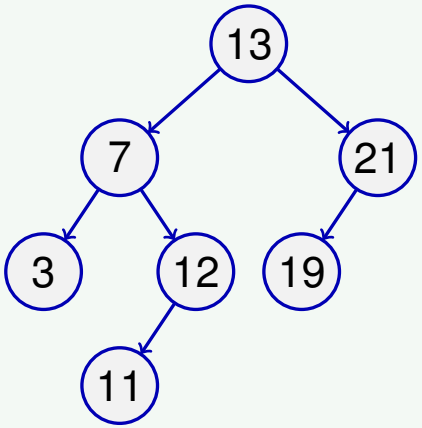


Suchbäume und *In-Order*-Traversierung

In-Order-Traversierung von Binären Suchbäumen

Wird ein binärer Suchbaum mittels Tiefensuche traversiert und werden die Werte der Knoten dabei *in-order* ausgegeben, so erhält man als Ergebnis die aufsteigend sortierte Reihung der Baumelemente.

Beispiel



in-order:

3	7	11	12	13	19	21

Binäre Suchbäume: Operationen (I)

Suche nach einem Element

- fange bei der Wurzel an
- solange gesuchter Wert nicht gefunden:
  - falls gesuchter Wert kleiner als aktueller Wert: steige nach links ab
  - falls gesuchter Wert größer als aktueller Wert: steige nach rechts ab
  - falls kein entsprechendes Kind vorhanden: Wert nicht enthalten

Einfügen eines Wertes

- Suche nach Einfügeposition mittels Suchalgorithmus (siehe oben)

$\rightsquigarrow$ Einhängen des neuen Knotens an dieser Position

Binäre Suchbäume: Operationen (II)

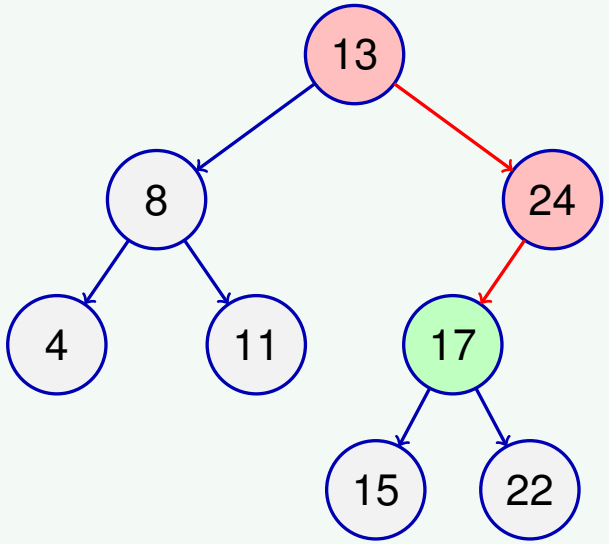
Löschen eines Wertes

- Suche nach Knoten mittels Such-Algorithmus
- falls zu löschender Knoten Blatt-Knoten ist: aus Baum entfernen
- falls zu löschender Knoten kein Blatt-Knoten ist: Knoten ersetzen durch...
  - größten Knoten im linken Teilbaum, *oder*
  - kleinsten Knoten im rechten Teilbaum;

  dadurch ggf. weitere Ersetzungen in den Teilbäumen notwendig!

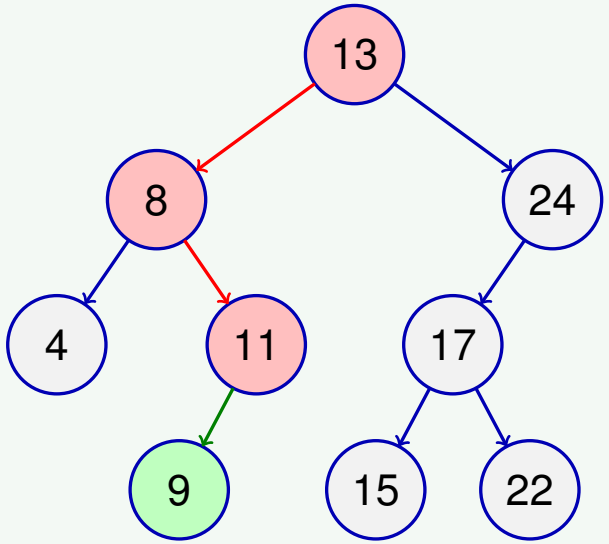
Beispiel: Suche

Beispiel: Suche nach 17



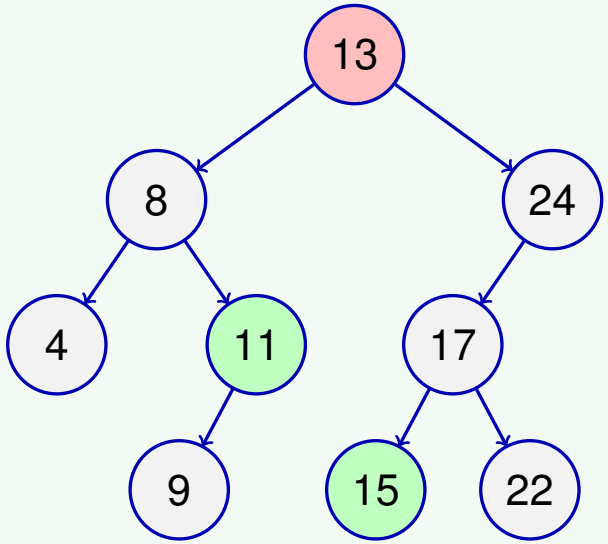
Beispiel: Einfügen

Beispiel: Einfügen von 9



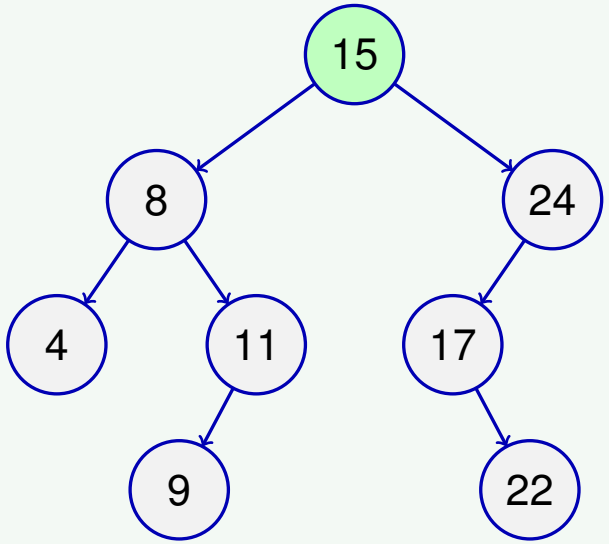
Beispiel: Löschen

Beispiel: Löschen von 13



Beispiel: Löschen

Beispiel: Löschen von 13



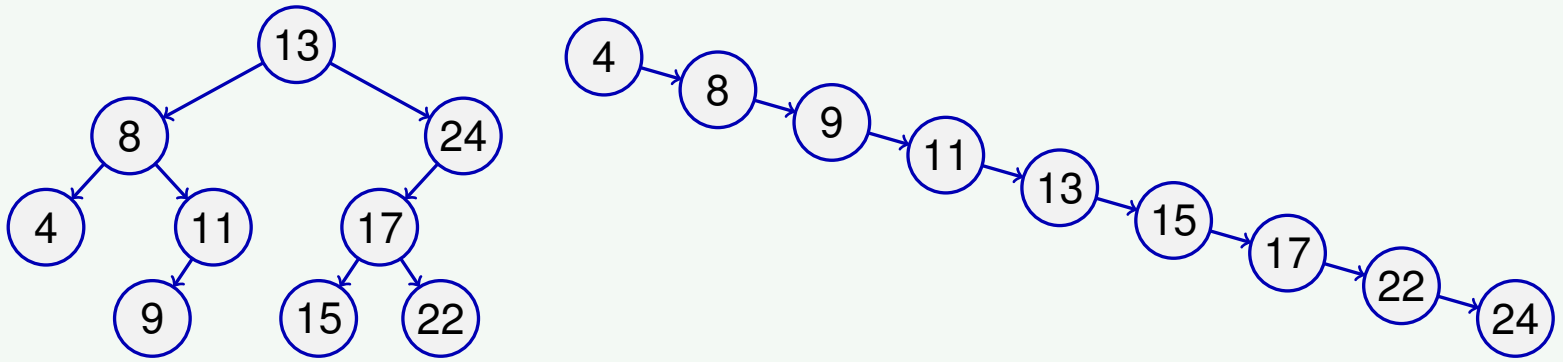
Beispiel an der Tafel

Beispiel an der Tafel

- Fügen Sie folgende Werte in einen anfangs leeren binären Suchbaum ein:
  - 14, 7, 4, 5, 24, 20, 13, 20, 25, 12
- Löschen Sie anschließend die folgenden Werte wieder aus dem Suchbaum:
  - 25, 13, 14

Entartete Suchbäume

Beispiel: Zwei Suchbäume mit denselben Werten...



Beobachtung

- ein Suchbaum ist nicht zwangsläufig balanciert
- Ausprägung eines Suchbaums hängt von Einfügereihenfolge ab

Problem? Problem!

Je schlechter ein Suchbaum balanciert ist, desto aufwändiger sind i.A. die typischen Suchbaum-Operationen, d.h. das Suchen, Einfügen und Löschen von Knoten.

Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT