Sven Eric Panitz

Lehrbriefe Programmierung in Java

Bäume

Bäume

Sven Eric Panitz

28. April 2023

Inhaltsverzeichnis

1 Bäume als verallgemeinerte Listen

In diesem Lehrbrief soll jetzt das Prinzip der Listen verallgemeinert werden zu Bäumen. Listen sind nämlich nur ein Spezialfall einer Baumstruktur.

Bäume sind ein gängiges Konzept, um hierarchische Strukturen zu modellieren. Sie sind bekannt aus jeder Art von Baumdiagramme, wie Stammbäumen oder Firmenhierarchien. In der Informatik sind Bäume allgegenwärtig. Fast alle komplexen Daten stellen auf die eine oder andere Art einen Baum dar. Beispiele für Bäume sind mannigfach:

- **Dateisystem**: Ein gängiges Dateisystem, wie es aus Unix, MacOS und Windows bekannt ist, stellt eine Baumstruktur dar. Es gibt einen ausgezeichneten Wurzelordner, von dem aus zu jeder Datei einen Pfad existiert.
- **Klassenhierarchie**: Die Klassen in Java stellen mit ihrer Ableitungsrelation eine Baumstruktur dar. Die Klasse `Object` ist die Wurzel dieses Baumes, von der aus alle anderen Klassen über einen Pfad entlang der Ableitungsrelation erreicht werden können.

- **XML**: Die logische Struktur eines XML-Dokuments ist ein Baum. Die Kinder eines Elements sind jeweils die Elemente, die durch das Element eigeschlossen sind.
- **Parserergebnisse**: Ein Parser, der gemäß einer Grammatik prüft, ob ein bestimmter Satz zu einer Sprache gehört, erzeugt im Erfolgsfall eine Baumstruktur.
- **Listen**: Auch Listen sind Bäume, allerdings eine besondere Art, in denen jeder Knoten nur maximal ein Kind hat.
- **Berechnungsbäume**: Zur statischen Analyse von Programmen, stellt man Bäume auf, in denen die Alternativen eines bedingten Ausdrucks Verzweigungen im Baum darstellen.
- **Tableaukalkül**: Der Tableaukalkül ist ein Verfahren zum Beweis logischer Formeln. Die dabei verwendeten Tableaux sind Bäume.
- **Spielbäume**: Alle möglichen Spielverläufe eines Spiels können als Baum dargestellt werden. Die Kanten entsprechen dabei einem Spielzug.
- **Prozesse**: Auch die Prozesse eines Betriebssystems stellen eine Baumstruktur dar. Die Kinder eines Prozesses sind genau die Prozesse, die von ihm gestartet wurden.

Wie man sieht, lohnt es sich, sich intensiv mit Bäumen vertraut zu machen, und man kann davon ausgehen, was immer in der Zukunft neues in der Informatik entwickelt werden wird, Bäume werden darin in irgendeiner Weise eine Rolle spielen.

Ein Baum besteht aus einer Menge von Knoten, die durch gerichtete Kanten verbunden sind. Die Kanten sind eine Relation auf den Knoten des Baumes. Die Kanten verbinden jeweils einen Elternknoten mit einem Kinderknoten. Ein Baum hat einen eindeutigen Wurzelknoten. Dieses ist der einzige Knoten, der keinen Elternknoten hat, d.h. es gibt keine Kante, die zu diesen Knoten führt. Knoten, die keinen Kinderknoten haben, d.h. von denen keine Kante ausgeht, heißen Blätter.

Die Kinder eines Knotens sind geordnet, d.h. sie stehen in einer definierten Reihenfolge. Eine Folge von Kanten, in der der Endknoten einer Vorgängerkante der Ausgangsknoten der nächsten Kanten ist, heißt Pfad.

In einem Baum darf es keine Zyklus geben, das heißt, es darf keinen Pfad geben, auf dem ein Knoten zweimal liegt.

Knoten können in Bäumen markiert sein, z.B. einen Namen haben. Mitunter können auch Kanten eine Markierung tragen.

Bäume, in denen jeder Knoten maximal zwei Kinderknoten hat, nennt man Binärbäume. Bäume, in denen jeder Knoten maximal einen Kinderknoten hat, heißen Listen.

1.1 Formale Definition

Auch Bäume werden wir induktiv definieren:

Definition 1 *Sei eine Menge E der Elemente gegeben. Die Menge der Bäume über E im Zeichen $Tree_E$ sei dann die kleinste Menge, für die gilt:*

- *Der leere Baum ist ein Baum:* $\text{Empty} \in \text{Tree}_E$.
- *Sei* $e \in E$ *und seien* $t_1, \ldots, t_n \in \text{Tree}_E$.
  Dann ist auch $\text{Branch}(e, t_1, \ldots, t_n) \in \text{Tree}_E$.

Der einzige Unterschied zu unserer formalem Definition der Listen ist, dass im zweiten Punkt ein Baum aus n bereits existierenden Bäumen und einem neuen Element gebaut werden und nicht wie bei Listen aus einer existierenden Liste und einem Element. Wenn in der Definition der Bäume das n immer nur 1 ist, dann ist die Definition identisch mit der Definition von Listen.

2 Implementierung

Die Definition eines Baumes war sehr ähnlich der Definition der Listen. Entsprechend ist auch die Implementierung sehr ähnlich zu unserer Listenimplementierung. Doch zunächst die Imports, die wir in diesem Lehrbrief und den Augfaben benötigen.

Tree.java

```
package name.panitz.util;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.BiFunction;
import java.util.function.Consumer;
import java.util.*;
import java.io.*;
import java.util.stream.Collectors;
```

Wir schreiben eine Klasse, die generisch über die Elemente, die an den Baumknoten stehen, ist:

Tree.java

```
public class Tree<E>{
```

2.1 Felder der Baumklasse

Ebenso wie bei Listen, benötigen wir ein Feld, für das an dem Baumknoten stehende Element. Dieses wurde bei Listen meist als `head` bezeichnet.

Tree.java

```
  public E element = null;
```

Eine Listenzelle hatte einen Verweis auf die Restliste. Bei Bäumen gibt es nicht nur einen Unterbaum, sondern mehrere Unterbäume, die wiederum in einer Liste gespeichert werden. Hierzu

nutzen wir die Standardliste. Also anstatt eines Feldes `LL<A> tail` wie bei Listen, benötigen wir für Bäume ein Feld, das eine Liste von Teilbäumen enthält.

Tree.java

```
public List<Tree<E>> childNodes = new ArrayList<>();
```

In einem boolschen Flag sei vermerkt, ob es sich um einen leeren Baum, der noch gar keinen Knoten enthält, handelt oder nicht. Dieses entspricht dann der `Nil`-Liste.

Tree.java

```
public boolean isEmptyTree;
```

Wir verketten unsere Bäume auch rückwärts, von den Kindern zu ihren Elternknoten.

Tree.java

```
public Tree<E> parent = null;
```

2.2 Konstruktoren

Laut Spezifikation gibt es zwei Konstruktoren. Den ersten zum Erzeugen eines leeren Baumes:

Tree.java

```
public Tree(){
    isEmptyTree = true;
  }
```

Der zweite erzeugt eine neue Baumwurzel mit einer bestimmten Markierung.

Tree.java

```
  @SafeVarargs public Tree(E e,Tree<E>... ts){
    element = e;
    isEmptyTree = false;
    for (Tree<E> t:ts){
      if (!t.isEmptyTree) {
        childNodes.add(t);
        t.parent = this;
      }
    }
  }
```

Hier wird das Konstrukt einer variablen Parameteranzahl verwendet. Die drei Punkte an dem letzten Parameter zeigen an, dass hier 0 bis n Parameter eines Typs erwartet werden.

Intern wird für diese Parameter ein Array erzeugt, über den wir in diesem Fall mit einer for-each Schleife iterieren.

In diesem Konstruktor werden die Kinder ihrem Elternknoten zugefügt. Hierzu ist für die Kinder das Feld, das auf den Elternknoten verweist, zu setzen. Es kann jeder Knoten nur einmal in genau einem Baum existieren.

2.3 Beispielbaum

Als ein wenig abstraktes Beispiel seien die Nachfahren des englische Königs George VI als ein Stammbaum angegeben:

Tree.java

```
public static Tree<String> windsor =
  new Tree<>("George"
    ,new Tree<>("Elizabeth"
      ,new Tree<>("Charles"
        ,new Tree<>("William"
                   ,new Tree<>("George")
                   ,new Tree<>("Charlotte"),new Tree<>("Louise"))
        ,new Tree<>("Henry",new Tree<>("Archie"))
        )
      ,new Tree<>("Andrew",new Tree<>("Beatrice"),new Tree<>("Eugenie"))
      ,new Tree<>("Edward",new Tree<>("Louise"),new Tree<>("James"))
      ,new Tree<>("Anne"
        ,new Tree<>("Peter",new Tree<>("Savannah"),new Tree<>("Isla"))
        ,new Tree<>("Zara",new Tree<>("Mia"),new Tree<>("Lena"))
        )
      )
    ,new Tree<>("Magaret"
      ,new Tree<>("David",new Tree<>("Charles"),new Tree<>("Margarita"))
      ,new Tree<>("Sarah",new Tree<>("Samuel"),new Tree<>("Arthur"))
      )
    );
```

Diesen Baum werden wir als Testdaten verwenden.

2.4 Beispielalgorithmen

Zum Arbeiten mit Bäumen geben wir ein paar Beispielmethoden, bevor weitere Methoden in der Aufgabe zu implementieren sind.

2.4.1 Anzahl der Knoten

Eine erste und einfache Frage ist die Frage nach der Anzahl der Knoten in einem Baum. Bei einer Liste entspricht das der Länge der Liste.

Anders als bei Listen gibt es nicht einen rekursiven Aufruf für den Tail der Liste, sondern für jedes Kind ist der rekursive Aufruf zu machen und die Teilergebnisse dann zu verknüpfen. Zur Ermittlung der Größe eines Baumes sind die Teilergebnisse zu addieren. Schließlich, wie auch bei der Länge einer Liste, ist das Ergebnis noch um eins zu erhöhen, nämlich für die Wurzel selbst.

Tree.java

```
public int groesse() {
  var result = 0;
  for (var child:childNodes){
    result += child.groesse();
  }
  return result+1;
}
```

Wir können uns aber auch daran erinnern, dass wir über die Kinderknoten einen Stream erzeugen können, für den eine einfache Faltung aufgerufen wird.

Tree.java

```
public int size() {
  return isEmptyTree ? 0
    :childNodes.parallelStream()
                .reduce(0, (r,c)->c.size()+r,(x,y)->x+y)+1;
}
```

Mit beiden Methoden können die Knoten eines Baumes durchgezählt werden:

Shell

```
jshell> windsor.groesse()
$2 ==> 29

jshell> windsor.size()
$3 ==> 29
```

2.4.2 Stringdarstellung

Eine der ersten Methoden, die üblicher Weise für Klassen geschrieben wird, ist die Darstellung des Objektes als ein Text in Form eines Stringobjektes. Hierzu wird gängiger Weise die Methode `toString` aus der Klasse Objekt überschrieben. Für die Baumklassen wäre folgende Umsetzung denkbar, die wir allerdings auf Deutsch als `alsString` bezeichnet haben.

Tree.java

```
public String alsString(){
  if (isEmptyTree) return "Empty()";
  String result = "Branch("+element+")[";
  for (Tree<E> child:childNodes){
    result = result+child.alsString();
  }
  return result+"]";
}
```

Diese Umsetzung ist rekursiv, indem die Methode für jeden Kindknoten rekursiv aufgerufen wird. Die Teilergebnisse für die Kindbäume werden dabei über die Stringkonkatenation mit dem eingebauten Operator + aneinander gehängt.

Dieses ist nicht ganz unproblematisch. Stringobjekte sind in Java unveränderbar (unmutable). Ein einmal erzeugtes Stringobjekt wird nie verändert. Der plus-Operator erzeugt aus zwei Stringobjekten ein neues Stringobjekt. Dieses geschieht dadurch, dass ein neues Objekt erzeugt wird und alle Zeichen der beiden Ausgangsobjekte in dieses neue Objekt kopiert werden. Wenn für die Berechnung eines Stringergebnisses viele Teilsstring mit dem plus-Operator aneinander gehängt werden, dann werden sehr viele Objekte erzeugt und insbesondere sehr viele Zeichen immer wieder im Speicher von einem Speicherbereich auf einen anderen kopiert.. Dieses werden wir im zweiten Teil der Vorlesung noch direkter sehen, wenn in der Programmiersprache C, die Operationen des plus-Operators explizit selbst programmiert werden müssen.<p />

Es empfiehlt sich also, bei der Konstruktion großer Texte, nicht gleich alle Teilstrings mit dem plus-Operator aneinander zu hängen und so permament Teilstrings zu kopieren.

Strings puffern mit `StringBuffer` Das Java Standard-API stellt eine Klasse zur Verfügung, die es ermöglicht eine Folge von Strings zu sammeln, die dann im Endeffekt zu einem großen String konkateniert werden: die Klasse `StringBuffer`. Diese hat intern eine Liste der einzelnen aneinander zu hängenden Strings. Erst wenn alle Strings eingesammelt wurden, dann werden alle Strings zu einem großen String zusammen kopiert. Somit werden die Zeichen der einzelnen Strings nicht mehrfach im Speicher kopiert und das Anlegen temporärer Stringobjekte vermieden.

Die Methode `toString()` kann mit Hilfe der Klasse `StringBuffer` effizienter umgesetzt werden. Es wird ein `StringBuffer`-Objekt angelegt. Mit der Methode `append` werden die einzelnen Strings dann diesem Objekt angefügt. Ein terminaler Aufruf der Methode `toString` auf dem `StringBuffer`-Objekt führt dann dazu, dass der Ergebnisstring erzeugt wird:

Zunächst nehmen wir aber den einfachen Fall des leeren Baumes aus:

Tree.java

```
@Override public String toString(){
  if (isEmptyTree) return "Empty()";
```

Andernfalls wird ein `StringBuffer`-Objekt mit dem Anfang des Ergebnisstrings erzeugt:

Tree.java

```
StringBuffer result = new StringBuffer("Branch(");
```

Diesem wird zunächst das Element in Stringdarstellung zugefügt:

Tree.java

```
result.append(element.toString());
result.append(")[");
```

Schließlich werden noch die Kinder durchiteriert, um deren String-Darstellung dem `StringBuffer`-Objekt zuzufügen:

Tree.java

```
for (Tree<E> child:childNodes){
  result.append(child.toString());
}
result.append("]");
```

Nun sind alle Teilstrings aufgesammelt und das Ergebnisobjekt kann erzeugt werden:

Tree.java

```
  return result.toString();
}
```

Verwendung eines Writer-Objekts Wer sich die letzte Lösung der Methode `toString()` etwas genauer angeschaut hat, der wird bemerkt haben, dass trotzdem viele temporäre Teilstrings erzeugt werden. Bei jedem rekursiven Aufruf, wird wieder ein neues `StringBuffer`-Objekt erzeugt, welches bei der Rückgabe einen neuen String erzeugt. Es werden also auch in dieser Umsetzung immer wieder dieselben Teilstrings im Speicher kopiert. Dieses Problem lässt sich nur umgehen, wenn man das `StringBuffer`-Objekt als Parameter an die Methode übergibt.

Eine andere Möglichkeit, die noch flexibler ist, ermöglicht es Strings nicht explizit zu erzeugen, sondern in eine Datensenke zu schreiben. Hierzu dient die aus dem letzten Semester bekannte Klasse `Writer`. Von dieser gibt es unterschiedliche Implementierungen, die jeweils die Zeichen in unterschiedliche Datensenken schreiben. Häufig wird diese natürlich eine Datei sein, es kann aber auch über ein Netzwerk an einen Server geschrieben werden, oder aber auch wieder nur einfach in einen String.

Um in ein `Writer`-Objekt zu schreiben, muss dieses wie am Ende des letzten Abschnitts erörtert, als Parameter übergeben werde. Zusätzlich ist zu beachten, dass die IO-Operationen zu Ausnahmen führen können:

Tree.java

```
public void writeAsString(String indent,Writer out) throws IOException{
```

Im Falle des leeren Baumes, wird dessen Darstellung geschrieben und die Methode verlassen:

Tree.java

```
  if (isEmptyTree) {
    out.write("[]");
    return;
  }
```

Wenn es sich um einen nichtleeren Baum handelt, können jetzt die einzelnen Teile nacheinander in den `Writer` geschrieben werden. Ähnlich wie in der vorherigen Lösung die einzelnen Teile dem `StringBuffer` angehängt wurden.

Tree.java

```
  out.write(element.toString());
```

Der Unterschied zu der vorherigen Lösung ist, dass jetzt die rekursiven Aufrufe den `Writer` als Parameter übergeben.

Tree.java

```
  if (!childNodes.isEmpty()){
    var newIndent = indent+"  ";
    for (Tree<E> child:childNodes){
      out.write(newIndent);
      child.writeAsString(newIndent,out);
    }
  }
}
```

Um diese Implementierung jetzt auf gleiche Weise nutzen zu können wie die Standardmethode `toString`, kann ein `StringWriter`-Objekt erzeugt werden, in den der String geschrieben wird.

Tree.java

```
public String asString(){
  try{
    var out = new StringWriter();
    writeAsString("\n",out);
    return out.toString();
  }catch (java.io.IOException e){
   return "";
  }
}
```

Mit dieser Implementierung ist unnötiges Kopieren von Teilstrings komplett vermieden worden. zusätzlich ist sie flexibler, weil die Methode `writeAsString` nicht nur zum Erzeugen eines Strings genutzt werden kann, sondern auch dazu, die Stringdarstellung in unterschiedliche Datensenken zu schreiben.

Erzeugen von LaTeX-Code Eine besondere Form von textueller Darstellung lässt sich über das Satzprogramm LaTeXerzielen. LaTeX-Quelltext ist ein textuelles Format, das mit Formatierungssequenzen es ermöglicht ein Druckbild zu erzeugen.

LaTeXhat eine Reihe Bibliotheken, um Bäume graphisch darzustellen. Wir verwenden das LaTeX-Paket `tikz-qtree` zur Darstellung von Bäumen.

Zunächst wird ein Ergebnispuffer erzeugt und in diesem der Quelltext für Beginn und Ende einer Baumumgebung in LaTeXerzeugt.

Tree.java

```
  public String toLaTeX() {
    var result = new StringBuffer();
    result.append("""
\\begin{tikzpicture}
\\tikzset{grow'=right,level distance=80pt}
\\tikzset{edge from parent/.append style={very thick}}");
\\Tree
""");
    toLaTeXAux(result);

    result.append("\n\\end{tikzpicture}");
    return result.toString();
  }
```

Dazwischen wird LaTeX-Quelltext für die einzelnen Knoten erzeugt. Ein Knoten ist in eckigen klammern eingeschlossen und die Elementmarkierung wird durch einen Punkt markiert.

Tree.java

```
  private void toLaTeXAux(StringBuffer result) {
    result.append("[.\\fcolorbox{black}{green!50!black}{\\bfseries ");
    result.append(element+"}");
    childNodes.forEach(child -> {
      result.append("\n");
      child.toLaTeXAux(result);

    });
    result.append(" ]");
  }
```

Damit lässt sich LaTeX-Quelltext generieren:

Shell

```
jshell> windsor.toLaTeX()
$3 ==> "\\begin{tikzpicture}\n\\tikzset{grow'=right,level distance=80p...
```

Das visuelle Ergebnis lässt sich in Abbildung 1 bewundern.

2.4.3 Navigation in verschiedene Richtungen

Ein Knoten kann in unserer Modellierung nicht nur die Kindknoten erfragen, sondern auch nach oben Richtung des Elternknoten schauen. Nur bei der Wurzel ist der Elternknoten nicht gesetzt.

Tree.java

```
public Tree<E> getRoot() {
  return parent==null?this:parent.getRoot();
}
```

So kann man zum Beispiel auch über en Großelternknoten alle Onkel und Tanten erfragen. Dabei ist der Elternknoten allerdings auszuschließen, denn Onkel und Tanten sind alle Kinder der Großeltern, die nicht die eigenen Eltern sind.

Tree.java

```
List<Tree<E>> auntsAndUncles(){
  var opa = parent.parent;
  return opa.childNodes.parallelStream()
    .filter(c->c!=parent).collect(Collectors.toList());
}
```

Die Kinder der Onkel und Tanten werden als Kusinen und Vettern bezeichnet.

Tree.java

```
List<E> cousins(){
  List<E> result = new ArrayList<>();
  if (parent==null||parent.parent==null) return result;
  auntsAndUncles()
  .stream()
  .forEach(cs -> cs.childNodes.stream()
                   .forEach(c->result.add(c.element)));
  return result;
}
```

3 Aufgaben

Aufgabe 1 Schreiben Sie nun weitere Funktionen für die Baumimplementierung.

a) Schreiben Sie eine Funktion, die das Argument auf alle Elemente des Baumes anwender.

Tree.java

```java
public void forEach(Consumer<? super E> con) {
  //TODO
}
```

Ein Beispielaufruf:

Shell

```
jshell> import static name.panitz.util.Tree.*
jshell> int[] bi = {0}
bi ==> int[1] { 0 }

jshell> windsor.forEach(x->bi[0]++)

jshell> bi
bi ==> int[1] { 29 }
```

b) Schreiben Sie eine Funktion, die testet, ob der Baum ein Element enthält, das das Prädikat erfüllt.

Tree.java

```java
public boolean contains(Predicate<? super E> pred) {
  //TODO
  return false;
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.contains(e->e.equals("Archie"))
$6 ==> true

jshell> windsor.contains(e->e.equals("Wilhelm"))
$7 ==> false
```

c) Schreiben Sie eine Funktion, die alle Elemente an Blättern des Baumes in einer Liste sammelt.

Tree.java

```
public List<E> fringe() {
  var result = new ArrayList<E>();
  fringe(result);
  return result;
}
public void fringe(List<E> result) {
  //TODO
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.fringe()
$8 ==> [George, Charlotte, Louise, Archie, Beatrice, Eugenie, Louise, James, Savannah, Isla, Mia, Lena, C
```

d) Schreiben Sie eine Funktion, die alle Elemente von eigenen Vorfahren, sortiert vom Element des Elternknoten hin zur Wurzel in eine Liste einfügt.

Tree.java

```
public List<E> ancestors(){
  var result = new ArrayList<E>();
  ancestors(result);
  return result;
}
public void ancestors(List<E> result){
  //TODO
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.childNodes.get(0).
   ...> childNodes.get(0).
   ...> childNodes.get(0).
   ...> childNodes.get(0).
   ...> ancestors()
$2 ==> [William, Charles, Elizabeth, George]
```

e) Schreiben Sie eine Funktion, die eine Liste mit allen Elementen meiner Geschwister füllt. Ich selbst gehöre nicht in diese Liste.

Tree.java

```
public List<E> siblings(){
  var result = new ArrayList<E>();
  siblings(result);
  return result;
}
public void siblings(List<E> result){
  //TODO
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.childNodes.get(0).childNodes.get(0).siblings()
$11 ==> [Andrew, Edward, Anne]
```

f) Schreiben Sie eine Funktion, die einen Pfad von der Wurzel zu einem Element erzeugt.

Wenn es einen Pfad von mir zu einem Nachkommen mit dem Element `elem` gibt, ist der Pfad in der Liste, ansonsten ist die Liste leer.

Tree.java

```
public List<E> pathTo(E elem) {
  var result = new ArrayList<E>();
  pathTo(elem,result);
  return result;
}

public void pathTo(E elem, List<E> result) {
  //TODO
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.pathTo("Savannah")
$12 ==> [George, Elizabeth, Anne, Peter, Savannah]
```

g) Erzeuge Sie einen neuen Baum, indem alle Elemente mit der Funktion `f` transformiert werden, die Baumstruktur aber übernommen wird.

Tree.java

```
public <R> Tree<R> map(Function<? super E, ? extends R> f) {
  //TODO
  return  null;
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.map(x->x.length()).fringe()
$2 ==> [6, 9, 6, 6, 8, 7, 6, 5, 8, 4, 3, 4, 6, 9, 6, 6]
```

h) Schreiben Sie eine Funktion, die ausgehend vom this-Knoten all Nachkommen einer bestimmten Generation abfragt. Die erste Generation entspricht den Kindern, die zweite allen Enkeln, die dritte allen Urenkeln. Die nullte Generation ist die einelementige Liste aus dem Element des Knotens selbst.

Tree.java

```
public List<E>  getLevel(int level) {
  var result = new ArrayList<E>();
  getLevel(level,result);
  return result;
}
public void getLevel(int level,List<E> result) {
}
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.getLevel(2)
$3 ==> [Charles, Andrew, Edward, Anne, David, Sarah]
```

i) Schreiben Sie eine Funktion, die alle Elemente in einer Liste sammelt, die von der Wurzel aus in derselben Generation sind wie der this-Knoten.

Tree.java

```
  public List<E> myGeneration() {
    return null;
  }
```

Ein Beispielaufruf:

Shell

```
jshell> windsor.childNodes.get(0).childNodes.get(0).myGeneration()
$5 ==> [Charles, Andrew, Edward, Anne, David, Sarah]
```

Tree.java

```
}
```

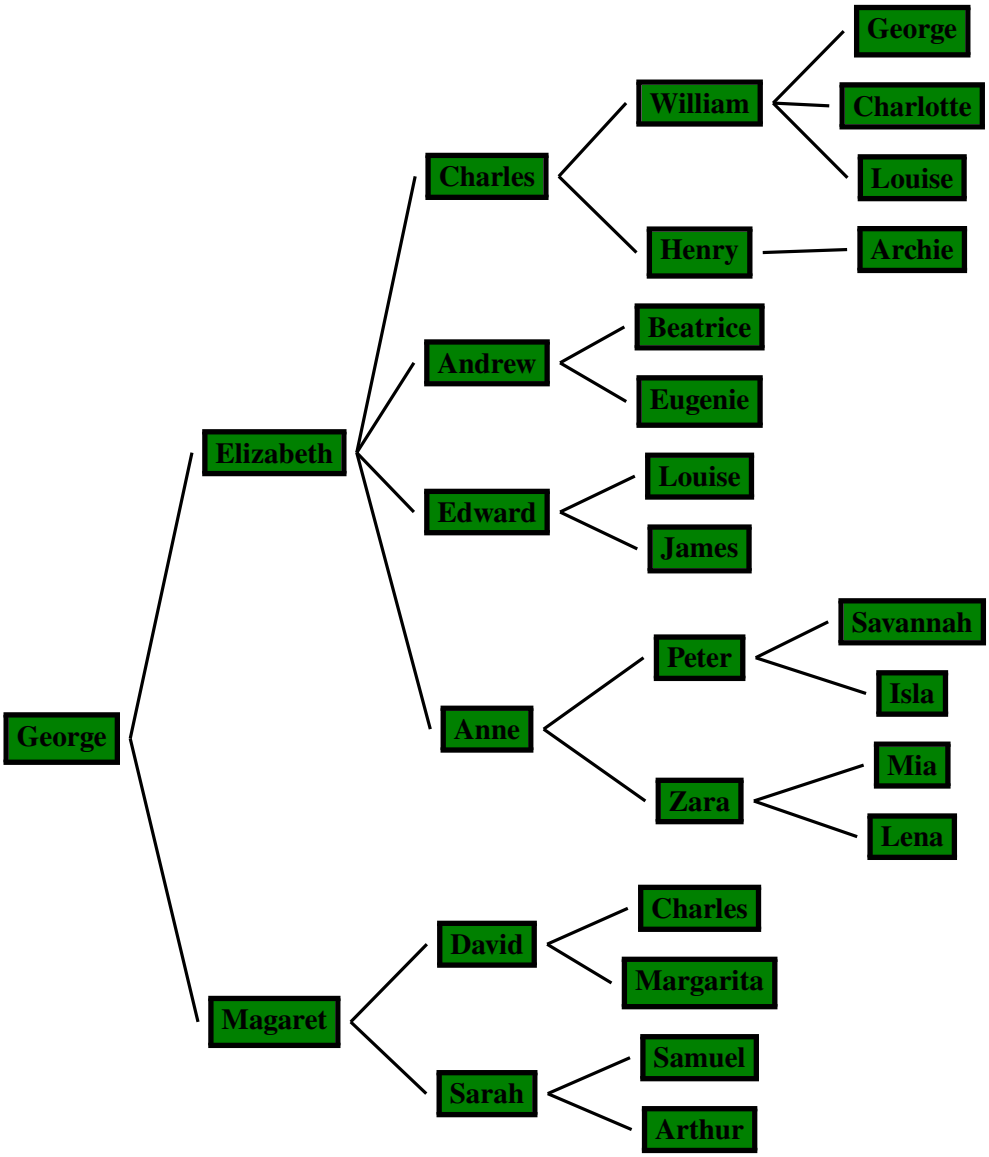

Abbildung 1: Abbildung zeigt den Stammbaum des englischen Königshauses mit Hilfe des LaTeX-Pakets `tikz-qtree`.

Baumimplemengtierung
LaTeX Darstellung
Rekursive Methoden