Rekursion

Binärbäume sind **rekursive Datenstrukturen**, denn jeder Binärbaum hat zwei Teilbäume - und die sind selber wieder Binärbäume.

Deswegen bietet es sich an, Binärbäume **rekursiv** zu durchlaufen.

Strategie

Bei der Implementierung einer rekursiven Methode für Binärbäume wird der gesamte Baum nur sehr grob betrachtet: Er besteht aus

- der Wurzel,
- dem linken Teilbaum (für den die Methode rekursiv aufgerufen wird)
- dem rechten Teilbaum (für den die Methode nochmal rekursiv aufgerufen wird).

In der **Sachlogik** muss man folgende Frage beantworten:

Wie setzt sich das Gesamtergebnis aus der Wurzel, dem Ergebnis des linken Teilbaumes und dem Ergebnis des rechten Teilbaumes zusammen?

Bestandteile einer rekursiven Methode:

- eine **Abbruchbedingung** oder mehrere Abbruchbedingungen.
  Diese hängen vom Sachzusammenhang ab - in der Regel braucht man mindestens eine Abbruchbedingung für einen leeren Binärbaum.
- **Wurzel auslesen** *(=Wurzelbehandlung)*
- **rekursive Aufrufe**: Die Methode ruft sich selber auf.
  - Meistens braucht man **zwei rekursive Aufrufe**: einen für den linken Teilbaum und einen für den rechten Teilbaum.
  - Bei Methoden, die etwas zurückgeben, muss man sich für den Rückgabewert interessieren!
- **Sachlogik**: Hier werden die Wurzel und die Ergebnisse der rekursiven Aufrufe behandelt.

Implementierungsbeispiel

```
public int summe(BinaryTree<Integer> pTree){
  // ergebnis deklarieren und geeigneten Startwert festlegen.

  _____________________________
  //Abbruchbedingung für einen leeren Baum
  if(_____________________________________){

    ______________________________;
  }
  //Wurzel auslesen
  int wurzel = pTree.getContent();
  //rekursive Aufrufe für den linken und den rechten Teilbaum

  ____________________________________________________

  ____________________________________________________
  //Sachlogik
  ergebnis = ____________________________________
  return ergebnis;
}
```

Durchlaufen eines Pfades

In manchen Situationen, vor allem in Bäumen mit Suchbaumstruktur, reicht es, wenn man einen **Pfad von der Wurzel bis zu einem Blatt durchläuft**. Das lässt sich mit einer `while`-Schleife realisieren, d.h. Rekursion ist hier nicht nötig.

Strategie

Um einen Pfad in dem Binärbaum `pTree` von der Wurzel zu einem Blatt zu durchlaufen, geht man wie folgt vor:

- Es wird eine `while`-Schleife geöffnet, die so lange läuft, bis `pTree` leer ist.
- In der `while`-Schleife wird - abhängig von der Sachlogik - nach links oder nach rechts abgebogen. Das realisiert man, indem man `pTree` durch seinen linken oder rechten Teilbaum updated.
- Nach Beendigung der `while`-Schleife ist man dann bei einem leeren Knoten unterhalb eines Blattes angekommen.

Implementierungsbeispiel

Als Beispiel wird die Methode **einfuegen** für einen mit Zahlen gefüllten Suchbaum implementiert.

```
public void einfuegen(BinaryTree<Integer> pTree, int pZahl) {
  // lokale Variable, fuer einen Baum den man "Zersaegen" kann.
  BinaryTree<Integer> b = pTree;
  // den Baum b so lange durchlaufen, bis man am "Ziel" ist.
  while(_____________________________){
    int wurzel = b.getContent();
    // UPDATE von b
    if(pZahl < wurzel){
      ____________________________________
    }
    else{
      ____________________________________
    }
  }  // end while
  // jetzt ist man beim richtigen leeren Knoten angekommen.
  // d.h. jetzt kann man einfuegen!
  ____________________________________________
}
```

Linearisierung

Linearisierung ist eine Strategie, wie man rekursive Strukturen (z.B. einen Binärbaum) komplett durchlaufen kann, ***OHNE eine rekursive Methode zu verwenden***.

Vorgehensweise

Die Vorgehensweise wird hier am Beispiel **Levelorder** aufgezeigt. In Levelorder wird der Binärbaum "schichtenweise" von oben nach unten durchlaufen, d.h. es handelt sich hier um eine **Breitensuche**.

Die Idee der Linarisierung ist die folgende:

Linearisierung:

1. eine Hilfsliste `baumListe` wird angelegt; in diese Hilfsliste kommen nur Bäume!
2. der ganze Baum wird in `baumListe` gepackt, d.h. `baumListe` hat jetzt ein Element.
3. dann wird `baumListe` mit einer Schleife von vorne bis zum Ende durchlaufen; dabei wird `baumListe` **ständig ergänzt!**
    1. bei jedem Schleifen-Durchlauf wird das aktuelle Element (=ein Baum) aus `baumListe` entnommen.
    2. die beiden Teilbäume (wenn sie nicht leer sind) werden hinten an `baumListe` angehängt.
4. Jetzt hat man in `baumListe` eine Liste aller Teilbäume von `pTree`.
    1. Diese Liste kann jetzt für die Sachlogik verwendet werden.

Sachlogik:

1. eine Ergebnisliste `ergebnisListe` wird angelegt; in `ergebnisListe` kommen die Knoten in der Levelorder-Reihenfolge.
2. `baumListe` wird mit einer Schleife durchlaufen. Bei jedem Schleifendurchlauf wird...
    1. der aktuelle Baum aus `baumListe` ausgelesen.
    2. die Wurzel des aktuellen Baumes in `ergebnisListe` eingefügt.

Implementierung der Liniearisierung

```
public List levelorder(BinaryTree<Integer> pTree){
  // 1) die Baumliste erstellen!
  // die baumliste als lokale Variable deklarieren und erzeugen
  List<_____________________________> baumListe = new List<>();
  // den urspruenglichen Baum an die Baumliste anhaengen
  baumListe.append(pTree);
  // die Baumliste mit einer Schleife durchlaufen
  // die Baumliste wird dabei immer mehr erweitert.
  for(baumListe.toFirst; baumListe.hasAccess(); baumListe.next()){
    BinaryTree<Integer> aktuell = baumListe.getContent();
    // Wenn rechter und linker Teilbaum nicht leer sind,
    // dann an die baumListe anhaengen
    if(_____________________________________________________){
      ___________________________________________________________
    }
    if(_____________________________________________________){
      ___________________________________________________________
    }
  }  // Ende der for-Schleife
  // 2) Sachlogik:
  // die baumListe durchlaufen,
  // von jedem Element (Typ: BinaryTree!)
  // die Wurzel auslesen und an ergebnisListe anhaengen
  List<Integer> ergebnisListe = new List<Integer>();
  for(baumListe.toFirst; baumListe.hasAccess(); baumListe.next()){
    ___________________________________________________________
    int aktuelleWurzel = ____________________________________
    ergebnisListe.append(aktuelleWurzel);
  }
  return ergebnisListe;
}
```