

Übungsblatt 8 - Lösungen

Aufgabe 1 - Polymorphie

Bestimme durch Überlegen die Zeichenfolge, die durch die Methode `run` ausgegeben wird.
Überprüfe Deine Vermutung gegebenenfalls durch Übersetzen und Ausführen der Methode `run`.

```
public static void m( char p ) { System.out.print( "X" ); }
public static void m( long p ) { System.out.print( "L" ); }
public static void m( double p ) { System.out.print( "D" ); }

public static void run()
{
    m( 'a' );
    m( new Integer( 2 ) + 2 );
    m( 8.1 / 2.0 );
    m( 2 + 10 );
    m( 8L / 2.0 );
    m( '1' / new Integer( 3 ) );
}
```

Ausgabe: X L D L D L

Bestimme durch Überlegen die Zeichenfolge, die durch die Methode `run` ausgegeben wird.
Überprüfe Deine Vermutung gegebenenfalls durch Übersetzen und Ausführen der Methode `run`.

```
class Top
{
    public void m( Top p ) { System.out.print("B"); }
    public void m( Bottom p ) { System.out.print("C"); }
}

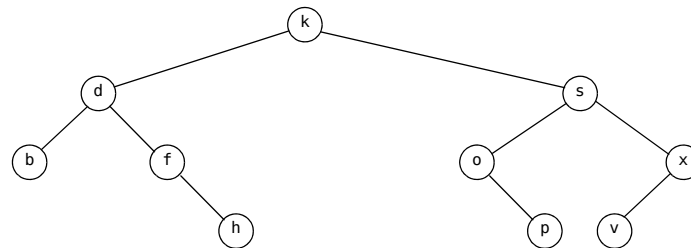
class Middle extends Top
{
    public void m( Bottom p ) { System.out.print("K"); }
}

class Bottom extends Middle
{
    public void m( Middle p ) { System.out.print("W"); }
    public void m( Bottom p ) { System.out.print("X"); }
}

class Test
{
    public static void run()
    {
        Top tm = new Middle();
        Top tb = new Bottom();
        Middle mb = new Bottom();
        tm.m( tb );
        tm.m( mb );
        tb.m( tm );
        tb.m( mb );
        mb.m( new Middle() );
        new Bottom().m( tb );
    }
}
```

Ausgabe: B B B B B B

Aufgabe 2 - Binärer Suchbaum – Konzeption



- Durchläufe durch Binäre Suchbäume
 - Gib die Reihenfolge der Zeichen bei Ausgabe durch einen *InOrder*-Durchlauf an: **b d f h k o p s v x**
 - Gib die Reihenfolge der Zeichen bei Ausgabe durch einen *PreOrder*-Durchlauf an: **k d b f h s o p x v**
 - Gib die Reihenfolge der Zeichen bei Ausgabe durch einen *PostOrder*-Durchlauf an: **b h f d p o v x s k**
- Aufbau von Binären Suchbäumen
 - Erstelle einen neuen Baum, in den Du die Werte in der Reihenfolge des oben vorgenommenen *InOrder*-Durchlaufs einträgst. Gib eine Begründung für die Struktur dieses Baums an.
Es entsteht ein einziger "langer Ast", da der nachfolgend eingefügte Wert aufgrund der Sortierung des *InOrder*-Durchlaufs immer größer als alle vorangehenden Werte ist.
 - Erstelle einen weiteren neuen Baum, in den Du die Werte in der Reihenfolge des oben vorgenommenen *PreOrder*-Durchlaufs einträgst. Gib eine Begründung für die Struktur dieses Baums an.
Es entsteht eine Kopie des Ausgangsbaums, da aufgrund der *PreOrder*-Folge Wurzelknoten immer vor allen ihren Kindern eingefügt werden.

Aufgabe 3 - Binärer Suchbaum – Vorgehen zum Löschen eines Knotens

- Überlege, wie der in Aufgabe 1 vorgegebene Baum restrukturiert werden muss, wenn das Zeichen **k** gelöscht werden soll. Formuliere einen allgemeinen Algorithmus zum Löschen eines Knotens in einem binären Suchbaum.

Der einfachste Algorithmus ist, den Inhalt des Wurzelknotens durch einen geeigneten anderen Inhalt des Baums derart zu ersetzen, so dass die Struktur weitgehend erhalten bleiben kann. Geeignet sind die Inhalte, die (1) einen Wert besitzen, der so dicht am Wert des zu löschenden Knotens liegt, dass dazwischen keine weiteren Werte liegen, die umsortiert werden müssten, und die (2) nur einen Nachfolgeknoten besitzen, der dann problemlos "aufrücken" kann. Es gibt zwei Inhalte, auf die diese Forderungen zutreffen: den größten Inhalt im linken Teilbaum (dessen Knoten keinen rechten Nachfolger hat) und den kleinsten Inhalt im rechten Teilbaum (dessen Knoten keinen linken Nachfolger hat).

- Methoden zum Löschen eines Knotens:
Implementiere eine Methode `CharacterSearchTree biggestInLeft()`, die eine Referenz auf den Knoten mit dem größten Wert im Attribut `token` im linken Teilbaum zurückgibt. Ist der linke Teilbaum leer, so soll der leere Baum zurückgegeben werden.

Die Methode setzt die Idee einfach um:

```

private CharacterSearchTree biggestInLeft()
{
    CharacterSearchTree current = leftChild;
    if ( ! current.isEmpty() )
    {
        while ( ! current.rightChild.isEmpty() )
        {
            current = current.rightChild;
        }
    }
    return current;
}

```

- Implementiere eine Methode `void delete()`, die den Inhalt – also das `HuffmanTriple`-Objekt – des Knotens aus dem Baum löscht, der die Methode `delete()` ausführt. Dazu soll der Knoten genutzt werden, den der Aufruf von `biggestInLeft` liefert: Der Inhalt dieses Knotens soll den zu löschenden Inhalt ersetzen, der Knoten selbst soll anschließend gelöscht werden.

```
public void delete()
{
    CharacterSearchTree biggest = biggestInLeft();
    if ( ! biggest.isEmpty() )
    {
        content = biggest.content;
        biggest.content = biggest.leftChild.content;
        biggest.leftChild = biggest.leftChild.leftChild;
        biggest.rightChild = biggest.leftChild.rightChild;
    } else {
        content = rightChild.content;
        leftChild = rightChild.leftChild;
        rightChild = rightChild.rightChild;
    }
}
```

Aufgabe 4 - Binärer Suchbaum – Eigenschaft bestimmen

Die Methode `boolean completePath()` soll `true` zurückgeben, falls es im Baum mindestens einen Pfad von der Wurzel zu einem Blatt gibt, auf dem alle inneren Knoten einen linken und einen rechten Nachfolgeknoten besitzen, die beide keine leeren Bäume sind. Existiert kein solcher Pfad, soll `false` zurückgegeben werden. Hat der Baum keine inneren Knoten oder ist er leer, so soll `true` zurückgegeben werden.

```
public boolean completePath()
{
    if ( isEmpty() || isLeaf() )
    {
        return true;
    } else {
        if ( leftChild.isEmpty() || rightChild.isEmpty() )
        {
            return false;
        }
        else
        {
            return leftChild.completePath() || rightChild.completePath();
        }
    }
}
```