TU Berlin
Software and Business Engineering
Prof. Dr. Ingo Weber
Dr. Tobias Heindel
Richard Hobeck

WiSe 2021/22
Aufgabe 9
Block 3
3. Januar 2022

Aufgabe 9 *Programmieren I*

Hinweise

- Die Abgabe dieser Übungsaufgaben muss bis spätestens Sonntag, den 9. Januar 2022 um 23:59 Uhr im ISIS-Kurs erfolgt sein. Es gelten die Ihnen bekannten Übungsbedingungen.
- Lösungen zu diesen Aufgaben sind als gezippter Projektordner abzugeben. Eine Anleitung zum Zippen von Projekten finden Sie auf der Seite des ISIS-Kurses. *Bitte benutzen Sie einen Dateinamen der Form* `VornameNachname.zip`.
- Bitte beachten Sie, dass Abgaben im Rahmen der Übungsleistung für die Zulassung zur Klausur relevant sind. Durch Plagiieren verwirken Sie sich die Möglichkeit zur Zulassung zur Klausur in diesem Semester.

Erstellen Sie für diese Hausaufgabe ein neues Projekt mit Namen Aufgabe9.

Aufgabe 9.1 *Bäume für Verzeichnisstrukturen* (5 Punkte)

Verzeichnisstrukturen von Dateisystemen bilden in natürlicher Weise eine Art von Bäumen (siehe Abbildung 1). Verzeichnisse sind (innere) Knoten, die Kind-Knoten haben, und zwar eine *(geordnete) Liste* von Kind-Knoten; schließlich bilden Dateien die Blätter des Baumes.

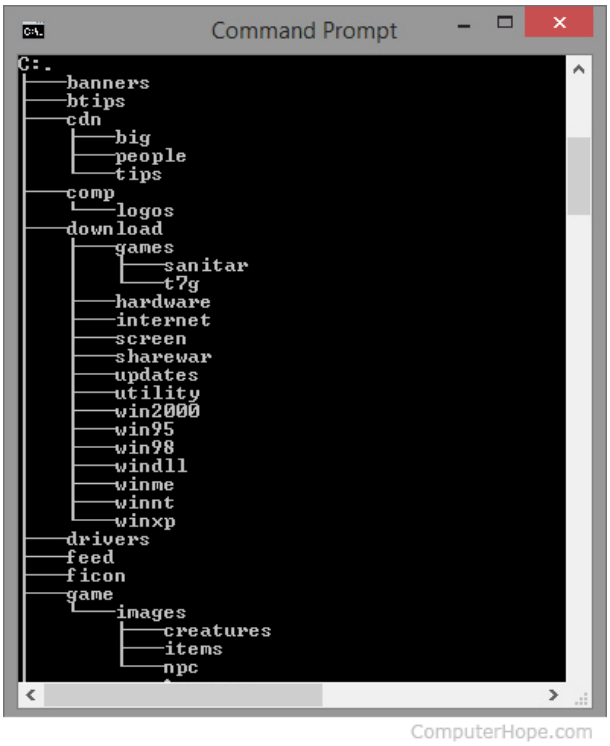

Abbildung 1: Ein Baum

- Die “Werte” im Baum sind nun die Namen der Knoten.
- Es gibt eine Variable `workingDirectory`, die das derzeitig Arbeits-Verzeichnis anzeigt.
- Die Klasse `DirectoryTree` hat eine Konstante `SEPARATOR`, die das Trennzeichen in Pfaden angibt (`/` oder `\`).
- Eine Methode `chooseSubDirectory`, mit der Sie in ein Unterverzeichnis absteigen können.
- Eine Methode `goUp`, die in das Arbeits-Verzeichnis auf dessen Eltern-Knoten setzt.
- Eine Methode `makeDirectory`, die ein Verzeichnis anlegt, im derzeitigen Verzeichnis.
- Eine Methode `createFile`, die eine Datei erzeugt, im derzeitigen Verzeichnis.

```
    public static void main(String[] args) {
        DirectoryTree x = new DirectoryTree();
        x.createFile("a");
        x.makeDirectory("dirOne");
        x.chooseSubDirectory("dirOne");
        x.createFile("b");
        x.goUp();
        x.createFile("c");
        System.out.println(x);
    }
```

```
/
|-a
|-dirOne
| |-b
|-c
```

Abbildung 2: Die `main`-Methode der `DirectoryTree` -Klasse und ihre Ausgabe

Vervollständigen Sie die Implementierung von Abbildung 3.

Die Grundidee von konkreten `Map`-Objekten, also Objekten von Klassen die von der `AbstractMap`-Klasse erben bzw. das `Map`-Interface implementieren, lässt sich kurz zusammenfassen.

`Map`-Objekte Wir haben eine endliche Menge von Schlüssel-Wert-Paaren (Englisch: *key-value pairs*)

$$\Big\{\langle k_1, v_1\rangle, \langle k_2, v_2\rangle, \ldots, \langle k_n, v_n\rangle\Big\}$$

mit der Eigenschaft, dass wenn zwei Schlüssel k_i und k_j gleich sind, dass dann auch die entsprechenden Werte v_i und v_j gleich sind. In kurz, wenn $k_i = k_j$, dann auch $v_i = v_j$ (für alle i, j). Das bedeuten dass jeder Schlüssel k_i genau einen Wert v_i zugeordnet hat.[1] Entsprechend wird die Menge von Paaren häufig auch etwas schöner wie folgt geschrieben:

$$\Big\{k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots, k_n \mapsto v_n\Big\}.$$

Wenn wir nun die API anschauen, dann finden wir folgende Zusammenhänge:

- die `keySet`-Methode liefert die Menge aller Schlüssel $\{k_1, \ldots, k_n\}$;
- die `get`-Methode liefert den einem Schlüssel k_i entsprechenden Wert v_i;
- die `put`-Mathode erlaubt es, ein neues Schlüssel-Wert-Paar $\langle k_j, v_j\rangle$ hinzuzufügen (bzw. einem bereits vorhandenem Schlüssel k_j einen neuen Wert v_j zuzuordnen).

Listen ohne Duplikate kann man nun durch `HashMap`-Objekte implementieren.[2] Die zwei wesentlichen Unterschiede sind folgende:

- Die `head`-Referenz ist nun vom Typ `T`.
- Das nachfolgende Element ist durch ein einziges `HashMap`-Objekt kodiert.

Aufgabe 9.2 *Listen ohne Wiederholung als `HashMap`* **(2 Punkte)**

Vervollständigen Sie die Klasse `NoDuplicatesList<T>` in Abbildung 4. Also, implementieren Sie die Methode, die ein neues Element an beliebiger Stelle einfügt.

Aufgabe 9.3 *Strukturelle Gleichheit von binären Bäumen* **(2 Zusatzpunkte)**

Schreiben Sie eine `equals`-Methode für `BinTree` (und `BinTreeNode`), die zwei Bäume genau dann als gleich betrachten, wenn die üblich graphische Darstellung nicht zu unterscheiden ist.[3]

[1] Mit anderen Worten, es handelt sich um (rechts-)eindeutige Relationen (siehe W).

[2] Achtung! Das ist natürlich nicht effizient.

[3] Tatsächlich könnte die sehr ineffiziente Implementierung wählen, in der wir die `toString`-Methode entsprechend implementieren. Also, Gleichheit ist dann Gleichheit bzgl. der `String`-Repräsentation.

```java
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;


public class DirectoryTree {
    Directory root = new Directory(null,SEPARATOR);

    Directory workingDirectory = root;

    static final String SEPARATOR = "/";

    public boolean chooseSubDirectory(String name){
        //
    }

    public void createFile(String name){
        //
    }

    public void makeDirectory(String name){
        //
    }

    @Override
    public String toString() {
        //
    }

    public void goUp(){
        //
    }

    public static void main(String[] args) {
        DirectoryTree x = new DirectoryTree();
        x.createFile("a");
        x.makeDirectory("dirOne");
        x.chooseSubDirectory("dirOne");
        x.createFile("b");
        x.goUp();
        x.createFile("c");
        System.out.println(x);
    }

}

class Node {
    Node parent;
    String name;
}

class File extends Node{
    StringBuffer contents = new StringBuffer();
}

class Directory extends Node{
    List<Node> contents = new LinkedList<>();
}
```

Abbildung 3: Die Struktur der Klasse `DirectoryTree`

```java
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

public class NoDuplicatesList<T> {
    T head;
    HashMap<T,T> next = new HashMap<>();

    public boolean contains(T element){
        if (head == null) {
            return false;
        } else {
            if (element.equals(head)){
                return true;
            } else {
                return (next.values()).contains(element);
            }
        }
    }

    public T get(int index){
        T tmp = head;
        for(int i = 0; i < index && tmp != null; i++){
            tmp = next.get(tmp);
        }
        return tmp;
    }

    public void insertAtHead(T x){
        if (head != null) {
            next.put(x, head);
        }
        head = x;
    }

    public void insertAtPosition(T x, int pos){
      //
    }

    public static void main(String[] args) {
        NoDuplicatesList<Integer> list = new NoDuplicatesList<>();
        for(int i = 0; i < 10; i++){
            list.insertAtHead(i);
        }
        for(int i = 0; i < 10; i++){
            System.out.println(list.get(i));
        }
    }
}
```

Abbildung 4: Liste ohne Duplikate