SWO 3x Übung zu Softwareentwicklung mit klassischen Sprachen u. Bibliotheken 3 **WS 2014/15, Übung 06**

Abgabetermin: Sa in der KW 03

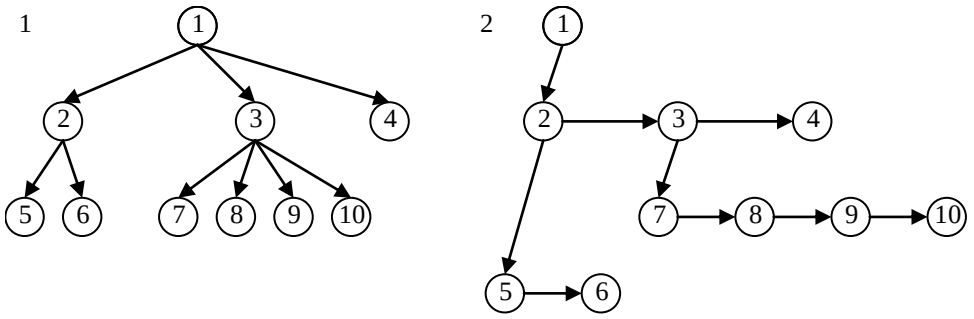
☐ **Gr. 1,** DI Franz Gruber-Leitner **Name** Roman Lumetsberger **Aufwand in h** 8

☐ **Gr. 2,** Dr. Erik Pitzer

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

1. Allgemeine Bäume (4 + 2 + 4 + 2 + 4 Punkte)

Im ersten Semester haben wir uns intensiv mit *Binärbäumen* beschäftigt, bei denen jeder Knoten maximal zwei Kinder hat. Eine Spezialform davon sind *binäre Suchbäume*, bei denen die Knoten "geordnet" sind (linkes Kind < Knoten <= rechtes Kind). Im dritten Semester haben wir uns bisher mit *allgemeinen Graphen* und deren Repräsentation (Adjazenzmatrix und Adjazenzliste) sowie Algorithmen darauf beschäftigt. *Allgemeine Bäume* fehlen noch in der Sammlung.
Entwickeln Sie einen abstrakten Datentyp zur Verwaltung von *allgemeinen Bäumen*, siehe z.B. Abbildung 1. Eine einfache Repräsentation solcher Bäume besteht darin, diese auf den Spezialfall der Binärbäume zurückzuführen, indem jeder Knoten einen Zeiger auf das erste Kind (in der Komponente `firstChild`) und einen Zeiger auf den Anfang der „Liste“ seiner Geschwister (in der Komponente `nextSibling`) hat. Jeder Knoten kommt hier mit zwei Zeigern aus, unabhängig davon, wie viele Kinder er hat. Man nennt diese Darstellung *kanonische Form*, siehe Abbildung 2.



Solch ein Datentyp lässt sich elegant mit den Mitteln der objektorientierten Programmierung realisieren: Ein möglicher Ansatz besteht darin, die Knoten des Baums und den Baum selbst durch zwei Klassen zu modellieren. Jeder Baum hat einen Zeiger auf den Wurzelknoten. Hier ist der Ansatz zur möglichen Klassendeklarationen für die abstrakte Klasse `Node` und die Klasse `Tree`:

```
class Node {
  private:
    Node *firstChild, *nextSibling;
    ...
  public:
    explicit Node(Node *firstChild = nullptr, Node *nextSibling = nullptr);
    virtual ~Node();
    virtual Node* getFirstChild() const;
    virtual Node* getNextSibling() const;
    virtual void setFirstChild(Node *n);
    virtual void setNextSibling(Node *n);
    virtual void print(std::ostream &os) const = 0;
    ...
};
```

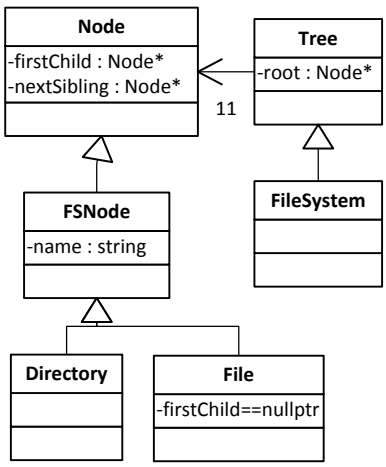
```
class Tree {
  protected:
    Node *root;
    ...
  public:
    Tree();
    virtual ~Tree();
    virtual Node* getRoot() const;
    virtual void insertChild(Node *parent, Node *child);
    virtual void deleteSubtree(Node *node);
    virtual int getSize() const;
    virtual void Clear();
    virtual void DeleteElements();
    virtual void print(std::ostream &os) const;
    ...
};
```

1. Erstellen Sie für beide Klassen je eine h- und eine cpp-Datei und implementieren Sie alle oben deklarierten Methoden in den beiden Klassen.
2. Erstellen Sie einen ersten konkreten Knotentyp `IntNode` der von `Node` abgeleitet ist und eine Datenkomponente `value` besitzt.
3. Fügen Sie in die Klasse `Tree` jene Methoden zum Einfügen neuer Knoten ein, um damit den Baum aus obigem Beispiel aufbauen zu können und stellen Sie eine Methode `getSize()` zur Verfügung, welche die Anzahl der Knoten im Baum liefert.
4. Ausgabeoperatoren `operator<<()` für beide Klassen dürfen natürlich auch nicht fehlen, damit Sie besser testen können.
5. Implementieren Sie einen geeigneten Zuweisungsoperator und Kopierkonstruktor für `Tree` und erläutern Sie anhand von Darstellungen deren Funktion.

Testen Sie Ihre Klassen ausführlich, überprüfen Sie, ob alle Objekte freigegeben werden.

2. Hierarchisches Dateisystem (8 Punkte)

Auf Basis der oben erstellten "Infrastruktur" für allgemeine Bäume, bietet es sich an, ein hierarchisches Dateisystem (*file system*) zu implementieren.



Erstellen Sie einen weiteren konkreten Knotentyp `FSNode` der eine neue Datenkomponente `name` vom Datentyp `string` hat, und von `FSNode` abgeleitet, zwei weitere Klassen: eine Klasse `Directory` und eine Klasse `File`. Die wesentliche Eigenschaft einer Datei ist, dass sie keine weiteren Dateien oder Verzeichnisse enthalten kann es muss also in einem `File`-Objekten immer `firstChild == nullptr` gelten.

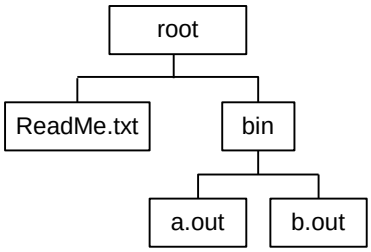
Zum Schluss leiten Sie noch eine Klasse `FileSystem` von `Tree` ab, die Funktionen zur Modifikation bietet.

Damit man mit Ihrem Dateisystem arbeiten kann, stellen Sie in der Klasse *FileSystem* folgende Methoden zur Verfügung, die, entsprechend der gleichnamigen Shell-Befehle, Methoden zum Erstellen und Löschen von Dateien und Verzeichnissen zur Verfügung stellen:

```
void touch(const string &path, const string &filename); // create new file
void mkdir(const string &path, const string &dirname); // create new directory
void rm(const string &path, const string &filename); // remove file
void rmdir(const string &path, const string &dirname); // remove directory
void ls() const; // list file system contents
```

Ihre Klassen sollten wie folgt verwendet werden können, um ein „Dateisystem“ gemäß der Abbildung rechts erstellen und verwalten zu können:

```
FileSystem *fs = new FileSystem();
fs->mkdir("", "root");
fs->touch("root", "ReadMe.txt");
fs->mkdir("root", "bin");
fs->touch("root/bin", "a.exe");
fs->touch("root/bin", "b.exe");
fs->ls();
fs->rm("root/bin", "a.exe");
fs->rmdir("", "root"); // -> ERROR: dir not empty
fs->ls();
fs->rm("root/bin", "b.exe");
fs->rmdir("", "root/bin");
fs->rm("root", "ReadMe.txt");
fs->rmdir("", "root");
fs->ls();
delete fs;
```



Hinweis:
Eine einfache Möglichkeit um Pfade mit Hilfe eines Separators zu trennen stellt die Funktion `strtok()` aus der C-Standardbibliothek dar, deren Funktion Sie in den Manual Pages nachlesen können.

1 Aufgabe 1 - Objektmengen (sets)

1.1 Anmerkungen

Diese Aufgabe wird mit der *minilib* umgesetzt und dadurch werden alle Methoden mit einem großen Anfangsbuchstaben benannt, da dies in der minilib Konvention ist.

1.2 Lösungsidee

Ein *Set* ist im grunde eine Liste, die keine mehrfachen Elemente enthalten darf. Darum wird die Klasse *Set* als Ableitung der Klasse *List*, die wir in der Übung erstellt haben, umgesetzt.

Es wird also eine doppelt verkettete Liste verwendet, die als Element die Klasse *Node* hat. Diese Klasse enthält dann das eigentliche Datenobjekt.
Es braucht dann nur die Methode Add überschrieben werden und um die Prüfung, ob ein Objekt bereits in der Liste vorhanden ist, erweitert werden. Diese Prüfung wird mit Hilfe der minilib Methode **IsEqualTo** realisiert.

1.2.1 Vereinigung - Union

Die Vereinigung zweier Mengen enthält alle Elemente der ersten und alle Elemente der zweiten Menge. Wichtig ist, dass jedes gleiche Element nur einmal in der Vereinigungsmenge vorkommt.

1.2.2 Schnittmenge - Intersect

Die Schnittmenge zweier Mengen enthält alle Elemente, die sowohl in der ersten, als auch in der zweiten Menge vorhanden sind.

1.2.3 Differenz - Difference

Die Differenz zweier Menge enthält jene Elemente der ersten, aber nicht in der zweiten Menge vorkommen.

1.3 Entwurfsentscheidungen

Die Lösung dieser Aufgabe erfordert grundsätzlich keine Änderung der minilib, da beim Einfügen eines bereits im *Set* enthaltenen Objektes eine Fehlermeldung auf *cerr* ausgegeben wird. Zugegeben, das ist nicht die Beste Lösung, doch in der kurzen Zeit von nur 1 Woche für die gesamte Übung, habe ich diese Variante gewählt. Weiters kann der Verwender der Klasse mit *Contains* prüfen, ob das Element bereits vorhanden ist.
Andere Möglichkeiten Folgende Lösungen wären auch möglich gewesen.

- Änderung des Rückgabeparameters der Methode *Collection::Add* von *void* auf *bool*, um den Aufrufer mitzuteilen, ob das Objekt eingefügt wurde oder nicht.
- Änderung des Rückgabeparameters der Methode *Collection::Add* von *void* auf *Object **. Der Rückgabewert würde dann das bereits enthaltene Objekt liefern falls es bereits vorhanden war, oder eben das Einzufügende. Damit könnte der Aufrufer unterscheiden, ob es bereits vorhanden war oder nicht.

1.3.1 Node

Diese Klasse implementiert einen Knoten im Baum. Dabei hat ein solcher Node genau einen Zeiger auf einen Kindknoten und einen Zeiger auf den nächsten Geschwisterknoten. Mit diesen beiden Zeigern können allgemeine Bäume umgesetzt werden.

- Die Zugriffe auf die Datenkomponenten werden mit Zugriffsmethoden gewährleistet.
- Der *Destruktor* löscht sowohl den Kind- als auch den Geschwisterknoten und gibt den Speicher frei.
- Die Methode *Clone* kopiert sowohl den Geschwister- als auch den Kindknoten. Diese Methode wird verwendet um eine Kopie des Baumes anzulegen.

1.3.2 Tree

Die Klasse *Tree* implementiert die geforderten Methoden um einen allgemeinen Baum mit Hilfe der Klasse *Node* aufzubauen.

- *InsertChild* fügt einen Kindknoten unter den gegebenen Knoten ein. Sollte der *Parent* gleich nullptr sein, dann wird dieser Knoten als Root-Knoten verwendet. Dabei ist zu beachten, dass der *Parent* auch wirklich Teil des Baumes ist.
- *DeleteSubtree* löscht einen Teilbaum und gibt auch seinen Speicher frei.
- *Clear* setzt nur den Root-Knoten auf nullptr und **gibt keinen Speicher frei**.
- *GetSize* berechnet die akuelle Anzahl an Knoten im Baum.

Kopierkonstruktor
Der Kopierkonstruktor kopiert den gesamten Baum, indem er die Clone Methode des Root-Elements verwendet und dadurch auch alle Kinder und Geschwister kopiert werden.
Zuweisungsoperator
Auch dieser Operator kopiert den gesamten Baum, indem er alle Nodes kopiert. Hier ist zu beachten, dass der alte Baum freigegeben werden muss. Dabei werden alle Nodes freigegeben und können danach nicht mehr verwendet werden.

2 Aufgabe 2 - Hierarchisches Dateisystem

2.1 Anmerkungen

Auch diese Aufgabe wird mit der *minilib* umgesetzt und dadurch werden alle Methoden mit einem großen Anfangsbuchstaben benannt, da dies in der minilib Konvention ist.

2.2 Lösungsidee

2.2.1 Node Klassen

Die benötigten Klassen werden, wie in der Angabe vorgegeben, abgeleitet und die Methoden entsprechend implementiert. Die Einschränkung der Klasse *File*, keine Kinder zu haben, kann durch das Überschreiben der Methode SetFirstChild sichergestellt werden.

2.2.2 Filesystem

Diese Klasse implementiert die geforderten Methoden. Dabei wird diese Klasse von *Tree* abgeleitet.
Zu den in der Basisklasse implementierten Methoden wird eine weitere benötigt, die es erlaubt einen Node im Tree anhand eines gegebenen Pfades zu finden. Diese muss also ausgehend vom Root alle Geschwister durchsuchen und bei jeden Seperator eine Ebene tiefer weitersuchen.
Hier wird die Funktion *strtok* der C-Standardbibliothek verwendet.

- **Touch**: Fügt einen neuen Knoten der Klasse *File* in den Baum ein.
- **Mkdir**: Fügt einen neuen Knoten der Klasse *Directory* in den Baum ein.
- **Rm**: Sucht und löscht den Knoten. Dabei muss beachtet werden, dass der gefundene Knoten vom Typ *File* ist.
- **Rmdir**: Verwendet *DeleteSubtree*, um das Verzeichnis zu löschen. Dabei muss geprüft werden, dass es keine Kindknoten gibt(also das Verzeichnis leer ist).
- **Ls**: Verwendet *Print*, um das Dateisystem auszugeben.

2.3 Sourcecode

Node.h

```
#ifndef NODE_H
#define NODE_H

#include <ostream>
#include <MLObject.h>


/* Node class used as base class for tree nodes
 *
 */
class Node : public ML::Object
{
  protected:
    Node *firstChild, *nextSibling;
  public:
    explicit Node (Node* firstChild = nullptr, Node* nextSibling = nullptr);
    // Deletes the firstChild and nextSibling and all containing sub elements
    virtual ~Node();

    virtual Node* GetFirstChild() const { return firstChild;}
    virtual Node* GetNextSibling() const { return nextSibling;}

    virtual void SetFirstChild(Node* node) { firstChild = node;}
    virtual void SetNextSibling(Node* node) { nextSibling = node;}

    //Prints the Node
    virtual void Print(std::ostream &os) const;

    // used for << operator in ML::Object
    virtual std::string AsString() const = 0;

    /* deeply clones the node */
    virtual Node* Clone() const = 0;

};

#endif // NODE_H
```

2.4 Testfälle