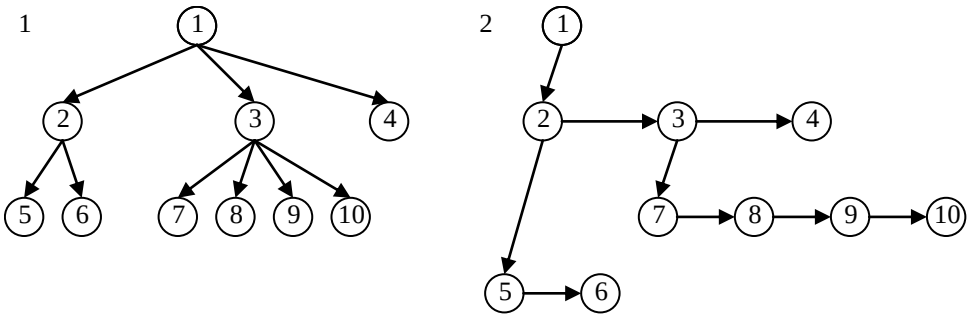
☐ **Gr. 1,** DI Franz Gruber-Leitner **Name** _____ **Aufwand in h** _____

☐ **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;
    ...
};
```

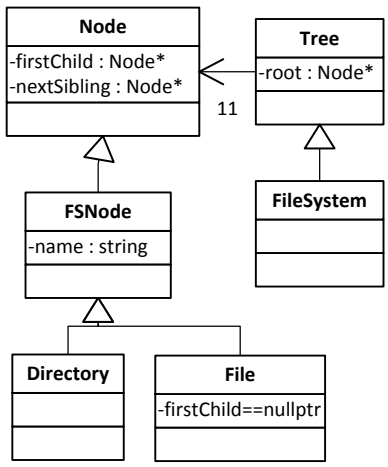
```
class Tree {
  protected:
    Node *root;
    ...
  public:
    Tree();
    virtual ~Tree();
    virtual Node* getRoot() const;
    virtual void insertChild(Node *parent, Node *child);
    virtual void deleteSubtree(Node *node);
    virtual int getSize() const;
    virtual void Clear();
    virtual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

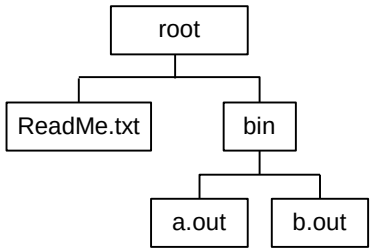
Zum Schluss leiten Sie noch eine Klasse `FileSystem` von `Tree` ab, die Funktionen zur Modifikation bietet.

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.