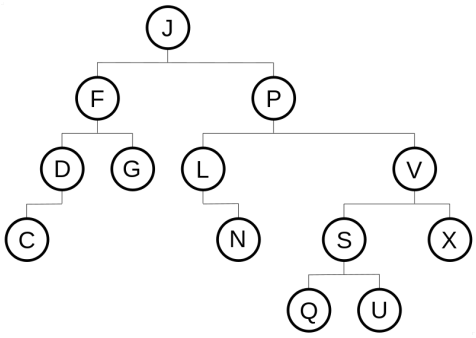
Prof. Dr.-Ing. Hartmut Schirmacher
Fachbereich Informatik und Medien

C/C++ für Java-Programmierer
Aufgabe 2.2: Die Map und der Baum



Aufgabe 2 besteht aus mehreren aufeinander aufbauenden Teilaufgaben und kann/soll über mehrere Wochen bearbeitet werden (sie *Abgabe*). Dieses Blatt enthält die Teilaufgabe 2.2.

Vorbereitung und Überblick

In Aufgabe 2.1 implementieren Sie einen Typ `RationalNumber`, in 2.2 eine assoziative Map, die es erlaubt, Objekte vom Typ `RationalNumber` als Suchschlüssel zu verwenden. In 2.3 schließlich verallgemeinern Sie diese Map mittels Templates und fügen Iteratoren hinzu.

Lösen Sie Aufgabe 2.1 zunächst vollständig, bevor Sie 2.2 implementieren. Falls Sie größere Schwierigkeiten mit 2.1 haben, ersetzen Sie die `RationalNumber` in der Map zunächst einfach durch den Typ `int` oder `std::string`.

Aufgabe 2.2: Modul `rn::Map`

Implementieren Sie ein Modul, welches eine Klasse Map definiert. Die Map soll jeweils einem Schlüssel vom Typ `RationalNumber` eindeutig einen Wert vom Typ `int` zuordnen. Intern soll die Map mittels eines einfachen binären Suchbaums implementiert werden (nur Suchen und Einfügen, kein Balancieren, kein Löschen; siehe Hinweise auf letzter Seite!).

Verpacken Sie zunächst Ihren Typ `RationalNumber` in einen Namensraum `rn`, und verfahren Sie genauso mit der Map-Klasse sowie allen zugehörigen Helferklassen.

Definieren Sie in der Klasse Map zwei öffentliche Typdefinitionen **`key_type`** und **`mapped_type`**, die jeweils als Alias für `RationalNumber` und `int` dienen. Der Sinn dieser Definitionen ist es, dass Sie von den tatsächlich verwendeten Typen so weit wie möglich abstrahieren und die Klasse später leichter in eine Template-Klasse umwandeln können (Aufgabe 2.3). ***Verwenden Sie überall in diesem Modul diese beiden Typ-Aliase, niemals direkt die Original-Typnamen.***

```
class Map
{
public:

    typedef RationalNumber key_type;
    typedef int mapped_type;
```

Die Map soll mindestens folgende öffentliche Schnittstelle anbieten:

- die vier wichtigen Grundoperatoren
  - Konstruktor ohne Argumente
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator
- eine Methode `contains(const key_type&)`, die nach einem Schlüssel im Baum sucht und nur einen bool zurückliefert
- einen `operator[](const key_type&)`, welcher es erlaubt, den Wert zu einem Schlüssel auszulesen und auch zu setzen, z.B. so:

```
Map m;
m[RationalNumber(3,4)] = 1:
int x = m[RationalNumber(3,4)];
```

- Tipp: Wenn der `operator[]` den Schlüssel nicht findet, fügt er ihn immer ein. Auch beim reinen Lesezugriff wird so ggf. ein neues Schlüssel/Wert-Paar eingefügt und der Wert dabei üblicherweise auf den "Standardwert" (Aufruf des Default-Konstruktors der Wert-Klasse) gesetzt.

Intern soll die Map einen Zeiger auf den Wurzelknoten des binären Suchbaums verwalten, der die eigentliche Arbeit übernimmt. Definieren Sie dazu eine getrennte Klasse `KeyValueNode`, die Sie entweder verdeckt implementieren oder sichtbar, aber in einem Namensraum `rn::internal`, um zu kennzeichnen, dass dieser Baum nicht für die direkte Benutzung von außen gedacht ist. Ein Knoten eines Baumes besteht immer aus einem Schlüssel, einem Wert, und Zeigern auf den linken und rechten Unterbaum. Es gibt keinen leeren Baumknoten, jedoch kann die Map leer sein, was sie durch `m_root = 0` repräsentieren soll.

Der Baum benötigt mindestens folgende Operationen:

- Wieder `typedefs` für `key_type` und `mapped_type`
- Konstuktor `(const key_type&, const value_type&)`
- Destruktor
- `find(const key_type&)` liefert Zeiger auf Knoten im Baum zurück, oder 0
- `insert(const key_type&, const value_type&)` fügt einen neuen Knoten an der richtigen Stelle geordnet in den Baum ein. Falls der Schlüssel schon vorhanden sein sollte, wird in dem vorhandenen Knoten einfach nur der Wert durch den angegebenen ersetzt.
- `clone()`, um einen Knoten (und alle Unterknoten rekursiv) für Kopier- und Zuweisungsoperationen der Map zu replizieren. Liefert Zeiger auf geklonten Baumknoten zurück.

- Alle Vergleichsoperationen von Schlüsseln im Baum sollten mit Hilfe der bereits in 2.1 implementierten Vergleichsoperationen == und < der Klasse `RationalNumber` erfolgen.

Sehr wichtig: überlegen Sie sich geeignete Units-Tests für **alle** Operationen der Map und testen Sie diese gründlich. Unterziehen Sie Ihren Code einem gründlichen Review. Stellen Sie vor allem sicher, dass

- das Programm korrekt funktioniert und nicht abstürzt ;-)
- keine Speicherlöcher entstehen. Legen Sie in Ihrem Test die Map-Objekte nicht dynamisch an, sondern auf dem Stack, so dass die Destruktoren am Ende der Tests automatisch aufgerufen werden. Stellen Sie sicher, dass die Destruktoren all dass Löschen, was durch die Klasse dynamisch angelegt wurde.
- beim Kopieren (Zuweisen) einer Map die zugewiesene Map vollständig repliziert wird, und keine Knoten gemeinsam von beiden Map-Objekten verwendet werden. Wenn Sie nach einer Zuweisungsoperation einen neuen Schlüssel in einen Baum einfügen, sollte dieser nicht z.B. in dem anderen Baum vorhanden sein.
- auch beim Zuweisen / Überschreiben einer Map mit einer anderen keine Speicherlöcher entstehen

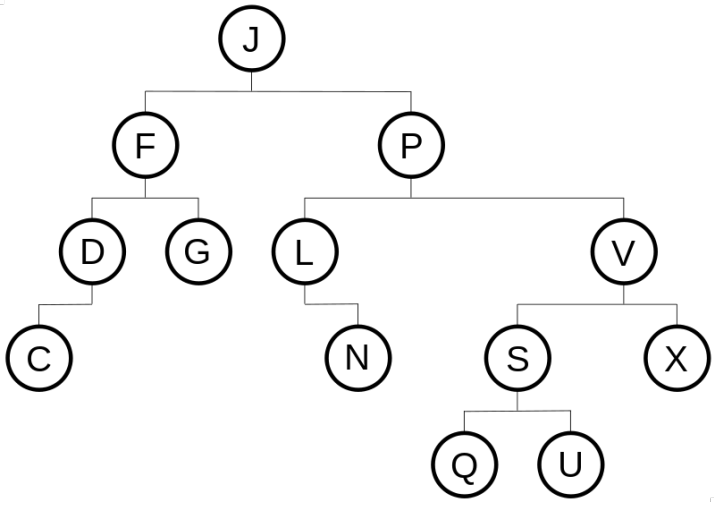
Abgabe und Demonstration

Die Abgabe der gesamten Aufgabe 2 soll bis zu dem in Moodle festgelegten Termin erfolgen. Verspätete Abgaben werden wie in den Handouts beschrieben mit einem Abschlag von 2/3-Note je angefangener Woche Verspätung belegt. Geben Sie bitte pro Gruppe jeweils nur eine einzige .zip-Datei mit den Quellen Ihrer Lösung ab. Bitte keine Kompilate in der Abgabe!

Demonstrieren und erläutern Sie dem Übungsleiter Ihre Lösung in der Übungen nach dem Abgabedatum. Die Qualität Ihrer Demonstration ist, neben dem abgegebenen Code, ausschlaggebend für die Bewertung! Es wird erwartet, dass alle Mitglieder einer Gruppe anwesend sind und Fragen beantworten können.

Tipp: Lesen Sie auch die Hinweise auf der folgenden Seite!

Anhang: Unbalancierte Binäre Suchbäume



Bildquelle: wikipedia.de, Nomen4Omen

Das Prinzip eines (unbalancierten) Suchbaums ist recht einfach:

- Jeder Knoten des Baums enthält Schlüssel und Wert und je einen Zeiger auf den linken und rechten Unterbaum (sowie einen auf den Elternknoten)
- Alle Schlüssel im **linken** Unterbaum sind **kleiner** als der aktuelle Schlüssel
- Alle Schlüssel im **rechten** Unterbaum sind **größer** als der aktuelle Schlüssel
- Die Zeiger können auch 0 sein, dann gibt es den jeweiligen Unterbaum nicht

Das Suchen in dem Baum ist einfach und effektiv. Beim Wurzelknoten startend, vergleiche Suchschlüssel mit aktuellem Schlüssel:

- Wenn gleich, ist der aktuelle Knoten der gesuchte Knoten
- Wenn Suchschlüssel< aktuellem Schlüssel, suche im linken Teilbaum weiter
- Sonst suche im rechten Teilbaum weiter

Das Einfügen eines neuen Schlüssels in den Baum ist ebenfalls nicht schwierig:

- (Wenn noch kein Baum existiert, lege neuen Wurzelknoten an. Diese Operation ist keine Methode des Baums, sondern des Objekts welches den Baum kapselt.)
- Sonst vergleiche neuen Schlüssel mit Schlüssel in aktuellem Knoten:
    - neuer = aktueller Schlüssel? Schlüssel bereits im Baum. Mit dem Ziel, ein Set bzw. eine Map zu implementieren, sollte man in diesem Fall den Wert im aktuellen Knoten mit dem neuen Wert überschreiben
    - neuer Schlüssel < aktueller Schlüssel:
        - linker Teilbaum leer: erzeuge neuen Knoten und hänge ihn als linken Teilbaum ein
        - sonst fahre im linken Teilbaum mit dem Einfügen fort
    - sonst: verfahre wie im "linken" Fall, aber mit dem rechten Teilbaum

Tipp: Sie können sich die Operationen sehr schnell vergegenwärtigen, wenn Sie einen Suchbaum mit einem Knoten auf dem Papier erzeugen und dann nach und nach einige Werte einfügen bzw. nach ihnen suchen.