ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

7. Juni 2018
Blatt 4

Objektorientierte Programmierung mit C++ (SS 2018)

Abgabe bis zum 14. Juni 2018, 16:00 Uhr

Lernziele:

- Entwicklung einer Template-basierten Container-Klasse

Aufgabe 5: Tries

Ein Trie, abgeleitet von *re**trie**val*, jedoch ausgesprochen wie *try*, ist eine Datenstruktur, die von René de la Briandais und Edward Fredkin 1959 bzw. 1960 zuerst vorgestellt worden ist. Während normale Suchstrukturen wie sortierte Bäume oder Hash-Verfahren den gesamten Schlüssel als atomar betrachten, besteht dieser bei Tries aus einer endlichen Folge von Zeichen eines endlichen Alphabets. Unterstützt werden dann präfix-basierte Abfragen, d.h. es ist effizient möglich, sämtliche Objekte zu ermitteln, deren Schlüssel mit einem gegebenen Präfix beginnen.

Implementiert werden Tries durch Mehrwegebäume, d.h. bei jedem Knoten des Baumes gibt es für jedes Zeichen des Alphabets einen möglichen Unterbaum. Ferner kann bei jedem Knoten ein dort zu findendes Objekt eingetragen werden. Bei Blattknoten gibt es immer ein Objekt zu finden, bei Nicht-Blattknoten kann ein Objekt zu finden sein. Es können bei einem Knoten jedoch nicht mehrere Objekte eingetragen werden. Wenn ein Objekt hinzukommt, wird das alte entfernt.

Wenn ein Schlüssel/Objekt-Paar in einen Trie eingefügt wird, dann findet ein rekursiver Abstieg in den Baum entsprechend den Zeichen des Schlüssels statt. Fehlende Knoten werden dabei erzeugt und eingehängt. Schließlich, wenn alle Zeichen des Schlüssels abgearbeitet sind, wird das Objekt bei dem zugehörigen Knoten abgelegt (und ggf. ein früheres Objekt entfernt).

Wenn alle Objekte zu einem Präfix ermittelt werden sollen, findet zunächst entsprechend des Präfix ein Abstieg in den Baum statt. (Wenn wir hier auf einen Nullzeiger stoßen, gibt es keine Objekte mit dem Präfix.) Wenn der Knoten ermittelt worden ist, gehören alle Objekte ab diesem Knoten zu den gewünschten.

Konkret sind Tries als Template-Container-Klasse zu implementieren. Die Parametrisierung des Templates darf auf den Objekttyp beschränkt werden. Aus Gründen der Einfach-

heit darf der Schlüsseltyp auf *std::string* festgelegt werden, wobei Sie dann das Alphabet auf Buchstaben beschränken dürfen in einer Weise, dass sowohl Klein- als auch Großbuchstaben auf den Bereich 0 bis 25 abgebildet werden. Andere Zeichen wären dann nicht zulässig. Eine Überprüfung sollte in diesem Falle mit Hilfe von Assertions erfolgen. (Alternativ dürfen Sie auch den Schlüsseltyp als Template-Parameter vorsehen, aber dies macht die Implementierung etwas komplizierter.)

Die Template-Klasse muss folgendes unterstützen:

- Default-Konstruktor, Kopier-Konstruktor, Zuweisungs-Operator und Destruktor,
- eine Methode, mit der ein Schlüssel/Objekt-Paar eingefügt werden kann,
- eine Methode, die die Zahl der eingetragenen Objekte liefert und
- eine Methode, mit der alle Objekte mit einem gegebenen Präfix abgefragt werden können.

Begleitend dazu ist ein kleines Testprogramm zu implementieren. Denkbar wäre es, eine Wortliste (etwa aus */usr/dict/words*) einzulesen und dann für jeden von der Standardeingabe eingelesenen Präfix die dazu passenden Wörter auszugeben.

Hinweise: Es steht Ihnen frei, ob Sie die Template-Klasse *inline* (also innerhalb von *Trie.hpp*) oder separat (also innerhalb einer Datei *Trie.tpp*) implementieren.

Da wir noch keine geeignete Techniken zu Iteratoren kennengelernt haben, erscheint es am einfachsten, die Methode, mit der Objekte abgefragt werden, mit Hilfe eines *Visitor*-Parameters zu realisieren:

```
template <typename Object>
class Trie {
    public:
        // ...
        template <typename Visitor>
        void visit(const std::string& key, Visitor visitor) const {
            // ...
        }
```

Das *Visitor*-Objekt *visitor* ist dann ein beliebiges Objekt, das wie eine Funktion aufgerufen werden kann, deren einziger Parameter das zu betrachtende Objekt ist:

```
/* visit object obj: */
visitor(obj);
```

Das *Visitor*-Objekt kann eine einfache Funktion sein:

```
void visit(const std::string& word) {
    cout << word << endl;
}
```

Dann kann die Ausgabe aller Objekte mit einem gegebenen Präfix so veranlasst werden:

```
Trie<std::string> trie;
// ...
trie.visit(prefix, visit);
```

Sie können wie üblich Ihre Lösung wieder einreichen:

```
theon$ submit cpp 5 Trie.hpp TestTrie.cpp
```

Falls Sie die *Trie*-Klasse in einer separaten Datei implementiert haben sollten:

```
theon$ submit cpp 5 Trie.hpp Trie.tpp TestTrie.cpp
```

Viel Erfolg!