ulm university universität
uulm

Institut für Numerische Mathematik
Dr. Andreas F. Borchert und Dr. Michael C. Lehn

8. Dezember 2016
Blatt 7

Objektorientierte Programmierung mit C++ (WS 2016/2017)

Abgabe bis zum 15. Dezember 2016, 16:00 Uhr

Lernziele:

- Entwicklung von Unterklassen zur Navigation in der übergeordneten Klasse
- Implementierung einfacher Iteratoren

Aufgabe 8: Navigieren in Tries

Die zuvor erarbeitete Implementierung für die Trie-Datenstruktur sah noch keine Navigationsmöglichkeit vor. Das ist auch nicht ganz einfach, da wir nicht ohne weiteres die privaten Zeiger nach außen sichtbar werden lassen wollen.

Dies Problem lässt sich mit Unterklassen vermeiden, die die privaten Zeiger verpacken. Im Rahmen dieser Übungsaufgabe können Sie ausgehend von der Ihnen selbst entwickelten Lösung zur Aufgabe 7 oder der Fassung, die am 8. Dezember in den Übungen vorgestellt worden ist, eine Version von *Trie.hpp* entwickeln, die das Navigieren in der Datenstruktur unterstützt.

Konkret ist dabei eine Unterklasse *Trie*::*Pointer* von *Trie* zu entwickeln, die das Navigieren unterstützt, ohne dabei die internen Zeiger preiszugeben.

Folgendes Code-Beispiel demonstriert die gewünschte Funktionalität. Sei *trie* ein bereits gefüllter *Trie* des Typs *Trie*<*std*::*string*> und *key* eine Zeichenkette des Typs *std*::*string*. Dann gibt das Beispiel alle gefundenen Wörter aus, die aus einem Präfix von *key* bestehen:

```
Trie<std::string>::Pointer ptr = trie.descend();
if (ptr.defined()) {
    for (char ch: key) {
        if (!ptr.descend(ch)) break;
        if (ptr.exists()) {
            cout << *ptr << endl;
        }
    }
}
```

trie.descend() liefert einen Zeiger auf die Wurzel. Mit der Methode *defined* lässt sich überprüfen, ob der Zeiger ungleich **nullptr** ist; mit *descend* ist ein Abstieg in den Präfixbaum mit dem gegebenen Zeichen möglich, wobei der zurückgegebene **bool**-Wert signalisiert, ob der Zeiger dadurch zum **nullptr** wurde. Mit *exists* lässt sich überprüfen, ob an dem erreichten Knoten im Präfixbaum ein Objekt existiert. Wenn ja, kann dieses mit einer Dereferenzierung abgerufen werden. Ausgehend von den Wörtern in */usr/dict/words* würde beispielsweise der Schlüssel „schoolgirlish“ zur Ausgabe der Wörter „s“, „SC“, „school“, „schoolgirl“ und „schoolgirlish“ führen.

Sie können wie üblich Ihre Lösung wieder einreichen:

```
thales$ submit cpp 8 Trie.hpp TestTrie.cpp
```

Aufgabe 9: Navigieren in Tries II

Mit dem *Trie*::*Pointer* lässt sich noch nicht auf einfache Weise ermitteln, welche weitergehenden Unterbäume zur Verfügung stehen, d.h. welche Zeichen zu einem Unterbaum führen. Dies könnte mit einem Iterator des Typs *Trie*::*Pointer*::*Iterator* geschehen. Hier ist ein Code-Beispiel, mit dem über die weiterführenden Zeichen iteriert wird:

```
if (ptr.defined()) {
   cout << "continuation possible with: ";
   for (char ch: ptr) {
      cout << ch;
   }
   cout << endl;
}
```

Damit das Iterieren mit einer **for**-Schleife klappt, müssen die folgenden Bedingungen erfüllt werden:

- *Trie*::*Pointer* muss die Methoden *begin* und *end* unterstützen, die jeweils einen Iterator liefern.
- Der Iterator muss den Operator „!=“ unterstützen:
  bool operator!=(**const** *Iterator*& *other*)**const**
- Der Iterator muss das Prefix-Inkrement unterstützen:
  Iterator& **operator**++()
- Der Iterator muss die Dereferenzierung unterstützen:
  char operator∗()**const**

Wenn Sie Iteratoren haben, können Sie mit deren Hilfe das folgende Rätsel lösen: Welches Wort aus */usr/dict/words* enthält die meisten Präfixe, die zugleich auch Wörter aus der gleichen Sammlung sind?

Wie üblich kann die Lösung wieder eingereicht werden:

```
thales$ submit cpp 9 Trie.hpp TestTrie.cpp
```

Viel Erfolg!