

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 10 (4 Seiten)

Selbstreferentielle Strukturen und Klassen

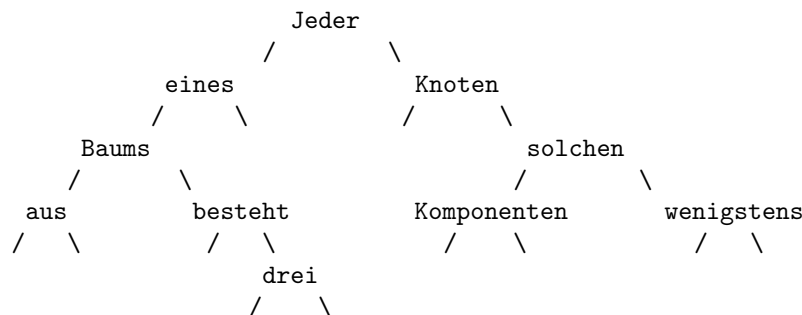
Zum Ordnen einer Folge von “zufällig” gereihten Daten eignet sich die Struktur des Binärbaums:

Jeder Knoten eines solchen Binärbaums besteht aus wenigstens drei Komponenten: den Daten, die zu diesem Knoten gehören sollen, und zwei Zeigern auf jeweils den “linken” und “rechten Teilbaum” zu diesem Knoten, in denen die Daten vorhanden sein sollen, die den Daten des aktuellen Knotens in der Ordnung (z. B. alphabetisch) vorangehen bz. nachfolgen.

Nehmen wir an, wir wollen die Wörter eines Textes auf diese Weise organisieren, z. B. die Wörter des Satzes:

“Jeder Knoten eines solchen Baums besteht aus wenigstens drei Komponenten”

Der Reihenfolge nach verarbeitet, erhalten wir folgende Struktur:



Das Vorgehen ist wie folgt: Zu einem gelesenen Wort wird ein Knoten konstruiert mit dem Wort als Dateneintrag und einem “linken” und “rechten” Zeiger auf einen Knoten gleichen Typs, die zunächst auf “nichts” (= null) zeigen.

Das nächste gelesene Wort wird mit dem ersten verglichen, geht es diesem in der (alphabetischen) Ordnung voran, wird es als “linker Unter-Knoten” eingetragen, folgt es in der Ordnung nach, wird es als “rechter Unter-Knoten” verwendet.

Jedes weitere Wort wird ähnlich behandelt: Es wird mit dem Anfangs- (“Wurzel”)-Knoten verglichen, geht es in der Ordnung diesem voran bzw. folgt es ihm nach, wird in den linken bzw. rechten Teilbaum gegangen und wieder mit dem dort stehenden Wort verglichen, und so wird fortgefahren, bis man zu einem Knoten kommt, an dem noch kein Wort steht. Dort wird es als Knoten eingetragen.

Als Erweiterung kann man den Fall betrachten, dass ein “neues” Wort mit einem bereits eingetragenen Wort identisch ist; man kann für diesen Fall einen Zähler bei den Knoten vorsehen, der ggf. erhöht wird und registriert, wie oft jedes Wort “gesehen” wurde.

Die so entstehende Struktur ist “selbstreferentiell”, weil jeder Knoten auf zwei Objekte *gleichen* Typs zeigt. Eine geordnete Ausgabe erhält man durch die folgende rekursive Vorschrift, bezogen auf die einzelnen Knoten, mit Start an der “Wurzel” des Baums:

1. Ausgabe des linken Teilbaums, falls vorhanden.
2. Ausgabe des Wortes am aktuellen Knoten, falls vorhanden.
3. Ausgabe des rechten Teilbaums, falls vorhanden.

Eine C-Implementation findet man in K&R, 2. Auflage, Chapter 6, Sect. 6.5.

Wir geben hier eine C++-Klasse an, die dies leistet. Um deutsche Wörter inclusive Umlauten korrekt sortieren zu können, verwenden wir die `locale`-Funktionen.

Aufgabe 10.1: Erweitern Sie die Funktionalität, so dass zu jedem Knoten=Wort auch noch die Zeilenzahlen seines Auftretens festgehalten werden.

```

// btree1.cc
// Eine Binaerbaum-Klasse
#include <stdio.h>
#include <ctype.h>
#include <iostream.h>
#include <locale.h>
#include "string-class.h"
template<class T>
class tree {
    struct node{
        T e; int c; node *l, *r;    // Element, Zaehler, linker/rechter Teilbaum
        node() { c = 1; l = NULL; r = NULL; }
    } *n;
public:
    tree() { n = NULL; }
    tree(node *nn) { n = nn;}
    tree& insert(const T &e) {
        if (n == NULL) { n = new node; n->e = e; }
        else if (e==n->e) { n->c++; }
        else if (e< n->e) { tree t = tree(n->l); t.insert(e); n->l = t.n; }
        else { tree t = tree(n->r); t.insert(e); n->r = t.n; }
        return *this;
    }
    friend ostream& inorder(ostream& os, tree t) {
        if (t.n) {
            inorder(os, t.n->l);
            os << t.n->e << '(' << t.n->c << ")\n" ;
            inorder(os, t.n->r);
        }
        return os;
    }
};

void main() {
    char s[300]; // Wort wie gelesen, inclusive Satzzeichen etc.
    char t[300]; // Wort nur mit Buchstaben
    char *p, *q;
    tree<string> tr; // Deklaration eines tree - Knotens
    setlocale(LC_ALL, "");
    while ( cin >> s ) {
        p = s; q = t;
        while (*p) { // Satzzeichen usw. ausfiltern.
            if ( isalpha(*p) )
                *q++ = *p;
            p++;
        }
        *q = '\0';
        // if (t[0]) // Nur wenn das Wort nicht "leer" ist
            tr.insert(t);
    }
    inorder(cout, tr); // Ausgabe der sortierten Woerter
}

```

Das erweiterte Beispiel zeigt auch die Routinen, die erlauben, ein Wort zu löschen, ohne die Ordnung zu zerstören. Das Hauptprogramm `main` hierzu ist auf diesen Seiten nicht gezeigt, siehe Programm-Verzeichnis auf der Web-Seite.

```
// btree.cc
// Eine Binaerbaum-Klasse

#include <iostream.h>
#include "string-class.h"

template<class T>
class tree {
    struct node{
        T e; int c; node *l, *r;    // Element, Zaehler, linker/rechter Teilbaum
        node() { c = 1; l = NULL; r = NULL; }
    } *n;
public:
    tree() { n = NULL; }
    tree(node *nn) { n = nn;}
    tree& insert(const T &e) {
        if (n == NULL) {
            n = new node; n->e = e;
        }
        else if (e==n->e) { n->c++; }
        else if (e< n->e) {
            tree t = tree(n->l); t.insert(e); n->l = t.n;
        }
        else {
            tree t = tree(n->r); t.insert(e); n->r = t.n;
        }
        return *this;
    }
    tree& rightmost() {                // macht Maximum zur Wurzel
        if (n && n->r) {
            node *p = n;
            while (p->r->r) p = p->r;
            node *rm = p->r;
            p->r = rm->l; rm->l = n; n = rm;
        }
        return *this;
    }
    tree& rrightmost() {                // rekursive Version von rightmost
        if ( n && n->r ) {
            tree t = tree(n->r);
            t.rrightmost();
            n->r = t.n->l; t.n->l = n; n = t.n;
        }
        return *this;
    }
    friend ostream& inorder(ostream& os, tree t) {
        if (t.n) {
            inorder(os, t.n->l);
```

```

        os << t.n->e << '(' << t.n->c << ")" " ;
        inorder(os, t.n->r);
    }
    return os;
}

friend ostream& out(ostream& os, tree t) {
    if (t.n) {
        cout << '(';
        cout << t.n->e << ','; out(os,t.n->l)<< ','; out(os,t.n->r);
        cout << ')';
    }
    else cout << '*';
    return os;
}

tree& del(const T &e) {
    if (n) {
        if (e == n->e) {
            if ( --n->c == 0 ) {
                if (n->l) {
                    tree ll = tree(n->l);
                    ll.rightmost(); ll.n->r = n->r;
                    delete n; n = ll.n;
                }
                else {
                    node *nn=n->r; delete n; n = nn;
                }
            }
        }
        else {
            if (e < n->e) {
                tree t = tree(n->l); t.del(e); n->l = t.n;
            }
            else {
                tree t = tree(n->r); t.del(e); n->r = t.n;
            }
        }
    }
    return *this;
}

};

// zu sortierende Arrays:
int a[] = {2,7,1,8,2,8,1,8,2,8,4,5,9,0,4,5,2,3,5,3,6,0,2,8,7,4,7,1,3,5,2,6,6};
// String-Array im 'C-Sil'
char *w[] = { "Sonntag", "Montag", "Dienstag",
    "Mittwoch", "Donnerstag", "Freitag", "Samstag",
    "Sonntag", "Montag", "Dienstag"};
// String-Array im Stil der C++Klasse aus string-class.h
string s[] = { "Sonntag", "Montag", "Dienstag",
    "Mittwoch", "Donnerstag", "Freitag", "Samstag",
    "Sonntag", "Montag", "Dienstag", "Mittwoch"};

```