BERGISCHE UNIVERSITÄT
GESAMTHOCHSCHULE WUPPERTAL
GAUSS-STRASSE 20
42097 WUPPERTAL
(Korrespondenzanschrift)
42119 WUPPERTAL
(Lieferanschrift)
TELEX 8 592 262 bughw
TELEFAX (0202) 439-2901
TELEFON (0202) 439-1

Fachbereich 7

MATHEMATIK

Prof. Dr. Hans-Jürgen Buhl
Praktische Informatik / Numerik

e-mail: Juergen.Buhl@math.uni-wuppertal.de

Algorithmen und Datenstrukturen (Informatik III)

WS1999/2000 – Übungsblatt 11

Abgabetermin: 9. Februar 2000

Aufgabe 1. *Binäre Suchbäume*

Ein binärer Suchbaum sei ein binärer Baum mit den Eigenschaften: Die Datenfelder keiner zwei Knoten des Baums seien gleich; in jedem Knoten des Baums gelte: die Werte der Datenfelder aller Knoten im linken Teilbaum sind kleiner, die im rechten Teilbaum größer als der Wert des Datenfeldes des Wurzelknotens.

Bringen Sie die folgende Implementierung eines Suchbaums auf Ihrem Computer zum Ablauf. Testen Sie.

```
////////////////////////////////
// Datei:   binTree.cc
// Version: 1.0
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

#include <iostream>
#include <string>

using namespace std;

template<class T>
class searchTree{

  struct Knoten{
    T Daten;
    Knoten *left;
    Knoten *right;
    Knoten(const T& daten, Knoten *p = 0, Knoten *q = 0):
        Daten(daten), left(p),  right(q) {};
  } *Anfang;
```

```
 void clear_all(const Knoten* abStelle){
   if (abStelle != 0) {
     clear_all(abStelle->left);
     clear_all(abStelle->right);
     delete abStelle;
   };
 };

 void show_inOrder(const Knoten* p) const
 {
    if (p != 0)
    {
         show_inOrder(p->left);
         cout << p->Daten << endl;
         show_inOrder(p->right);
    }
 }

  searchTree(Knoten* p) : Anfang(p) {};

public:

  searchTree(): Anfang(0) {};

  ~searchTree() { clear_all(Anfang); };


  class iterator;
  friend class iterator;

  iterator find(const T& value) const
  //
  // returns end(), iff not found
  //         position of found Knoten otherwise
  //
  {
    Knoten *Position(Anfang);
    bool found(false);

    while (!found && (Position != 0))
    {
       if (value < Position->Daten)
          Position = Position->left;
       else if (value > Position->Daten)
          Position = Position->right;
       else      //  value == Position->Daten
          found = true;
    };
```

```
                // (found == true  and  value == Position->Daten)
                // or (found == false   and   value not in "Tree")

  if (found)
    // bitte nicht: return *(new iterator(Position));
     return iterator(Position);
  else
     return iterator();

}

iterator insert(const T& value)
//
//  returns position of Knoten with value
//
{
  Knoten *Position(Anfang);
  Knoten *parent(0);
  bool found(false);

  if (Anfang == 0){
     Anfang = new Knoten(value);
     return iterator(Anfang);
  }


  while (!found && (Position != 0))
  {
     if (value < Position->Daten) {
        parent = Position;
        Position = Position->left;
     } else if (value > Position->Daten) {
        parent = Position;
        Position = Position->right;
     } else      //  value == Position->Daten
        found = true;
  };

 if (found)
      return iterator(Position);

  if (value < parent->Daten) {    // parent->left == 0
      parent->left = new Knoten(value);
      return iterator(parent->left);
  } else {                        // parent->right == 0
      parent->right = new Knoten(value);
      return iterator(parent->right);
  }
```

```
  }

  void show_inOrder() const
  {
    if (Anfang != 0){
     show_inOrder(Anfang->left);
     cout << Anfang->Daten << endl;
     show_inOrder(Anfang->right);
    }
  }

  class iterator {
    Knoten* aktuell;
  public:

    iterator(Knoten* init = 0): aktuell(init) {};

    searchTree SubTree() const {return searchTree(aktuell);};

    T& operator*() { return aktuell->Daten; };
    const T& operator*() const {return aktuell->Daten; };

  };

  void show_inOrder(const iterator& pos) const
  {
    pos.SubTree().show_inOrder();
  }

  iterator begin() const { return iterator(Anfang); };
  iterator end() const   { return iterator(); };

};

int main()
{
  searchTree<int> S;

  S.insert(3);
  S.insert(1); S.insert(0); S.insert(2);
  S.insert(5); S.insert(4); S.insert(6);

  S.show_inOrder();
  cout << endl;

  searchTree<int>::iterator wo(S.find(3));
  S.show_inOrder(wo);

}
```

Überprüfen sie mit dem Speicherleck-Test der C++ Entwicklungsumgebung und geeignet gewählten Testläufen, ob die Implementierung alle mit `new` allokierten Speicherblöcke auch wieder freigibt.

Schreiben Sie eine Spezifikation für die Methoden `find, insert` und `show_inOrder`.

Warum sollte z.B. in `find()` die Anweisung `return iterator(Position);` und **nicht** die Anweisung `return *(new iterator(Position));` benutzt werden, wie es Programmierer mit Vorkenntnissen in Java zuweilen tun.

Legt der Konstruktor `searchTree(Knoten* p)` eine flache (shallow; Objektgleichheit) oder eine tiefe (deep; Wertegleichheit) Kopie des von `p` ausgehenden (Teil-) Baumes an? Warum ist das in `SubTree()` die richtige Kopienart? Bei welchen anderen Operationen auf `searchTree's` wäre die andere Kopienart richtiger?

Ein vollständig besetzter binärer Baum hat 2^N-1, $n \in \mathbb{N}$, Knoten. Wie, das heißt in welcher Reihenfolge ist ein solcher Baum mittels `insert` aufzubauen? Testen Sie Ihren Algorithmus mit Integer-wertigen Bäumen mit den Knotenwerten $0 \ldots 2^N$-2.

Aufgabe 2. *Bearbeitungsreihenfolgen im binären Baum*

Die Methode

```
...
 void show_postOrder(const Knoten* p) const
 {
    if (p != 0)
    {
         show_postOrder(p->left);
         show_postOrder(p->right);
         cout << p->Daten << endl;    }
 }
...
public:

  void show_postOrder() const
  {
    if (Anfang != 0){
     show_postOrder(Anfang->left);
     show_postOrder(Anfang->right);
     cout << Anfang->Daten << endl;    }
  }
...
  void show_postOrder(const iterator& pos) const
  {
    pos.SubTree().show_postOrder();
  }
```

listet die Werte eines Suchbaums in einer anderen Reihenfolge als in der vorigen Aufgabe auf. Sagen sie diese Reihenfolge voraus und testen Sie Ihre Voraussage dann durch Einbau der drei Funktionen an geeigneten Stellen des Codes von Aufgabe 1.
Nicht immer führt die Rekursion zu einer geeigneten Algorithmenidee. Wenn sie die Werte eines Baums Reihe für Reihe von oben nach unten angeben wollen (also: erst die Wurzel, dann alle „Kinder“, dann alle „Enkel“, ...), so ist das geeignete Hilfsmittel eine Warteschlange etwa aus der STL:

```
...
#include <queue>
...
  void show_breadthFirst()  const
  {
    if (Anfang != 0){
     queue<Knoten *> toDo;
     toDo.push(Anfang);

     Knoten* k;
     while (!toDo.empty())
     {
        k = toDo.front();
        toDo.pop();

        if (k->left != 0) toDo.push(k->left);
        if (k->right != 0) toDo.push(k->right);

        cout << k->Daten << endl;
     };
    }
  }
...
  void show_breadthFirst(const iterator& pos) const
  {
    pos.SubTree().show_breadthFirst();
  }
```

Dokumentieren Sie die Algorithmenidee und testen Sie den Algorithmus.

Eine Warteschlange statt einer Rekursion erlaubt es nun, einen Iterator mit `operator++()` zu konzipieren. Dieser Iterator muß als Status eine `toDo`-Warteschlange enthalten und für nichtleere Bäume zur einelementigen Warteschlange initialisiert werden, die lediglich einen Zeiger auf die Wurzel des Baums enthält. Zusätzlich sollte der Iteratorstatus ein Attribut `gueltig` besitzen, das bei einer `insert`- oder `erase`-Operation im Baum für alle instantiierten Iteratoren des Baums auf `false` gesetzt wird. Dazu muß im Status der Baumklasse eine Liste von Zeigern auf alle existierenden Iteratorinstanzen mitgeführt werden. Das Attribut `gueltig` sollte natürlich im Falle des Wertes `false` alle weiteren Iterator-Operationen außer einer Rücksetzung

auf die Baumwurzel verhindern.

Ähnlich kann man auch Iteratoren mit einer Methode `operator++()` für die anderen Durchlaufreihenfolgen konzipieren, wenn man statt einer Warteschlange einen Kellerspeicher benutzt:

- postOrder: Benutze einen Stack mit Elementen, die je einen Baum-Knotenzeiger und einen Zähler enthalten. Der Zähler wird zu Null initialisiert und hält fest, wie häufig ein Baum-Knotenzeiger, der in Sequenz bis zu dreimal in den Stack eingefügt wird, schon vom Stack gepop't wurde: Beim ersten pop'en wird sowohl der Baum-Knotenzeiger als auch (iterativ) alle seine linken Söhne, Enkel, ... auf den Stack gepush't. Beim zweiten pop'en wird sowohl der Baum-Knotenzeiger als auch sein rechter Sohn gepush't und beim dritten pop'en wird der Baum-Knotenzeiger endgültig bearbeitet, ohne auf den Stack zurückgepop't zu werden.
- inOrder: Ähnlich; hier wird ein Baum-Knotenzeiger jedoch höchstens zweimal auf den Stack gepush't: Beim ersten pop'en wird sowohl der Baum-Knotenzeiger als auch sein linker Sohn gepush't, beim zweiten pop'en wird der rechte Sohn gepush't und der aktuelle Baum-Knotenzeiger endgültig bearbeitet.
- preOrder: Benutze einen Stack mit Baum-Knotenzeigern als Elemente. Initialisiere ihn analog wie oben geschrieben (jedoch ohne Zähler). Beim pop'en wird zunächst der linke und rechte Sohn gepush't und dann unmittelbar der aktuelle Baum-Knotenzeiger bearbeitet.

(Vergleiche zu diesen und anderen Problemen des Algorithmendesigns: M. Allen Weiss: Algorithms, data structures and problem solving, Seite 532-538.)

Aufgabe 3. *Löschen und Balancieren*

Überlegen Sie sich einen Algorithmus für eine Methode `void erase(const iterator& pos)`, die einen beliebigen Knoten eines binären Suchbaums löschen soll. Nach vollendeter Löschung soll der „umorganisierte" Baum natürlich weiter ein Suchbaum sein. Implementieren und testen sie.

Überlegen Sie sich eine Methode, die aus einem sortierten Feld der Länge 2^N-1, $n \in \mathbb{N}$, einen vollständigen optimal balancierten binären Suchbaum mit den Werten des vorgegebenen Feldes erzeugt. Implementieren und testen Sie auch hier.

Aufgabe 4. *Subrange*

Benutzen Sie in einem Testrahmen-Programm die folgende Klasse:

```
template<int low, int high>
class Subrange{

  int value;
```

```
public:

  Subrange(const int i = 0){
    if ((i < low) || (i > high)) throw "Subrange-Ausnahmebedingung in Konstrukto
    value = i;
  }

  // operator int() const { return value; }
  int get_value()const { return value; }

  Subrange & operator = (const int i){
    if ((i < low) || (i > high)) throw "Subrange-Ausnahmebedingung in Zuweisung"
    value = i;
    return *this;
  }

};
```

Warum kann hier im Konstruktor nicht die Initialisierung mittels

```
Subrange(const int i = 0): value(i) {}
```

benutzt werden?

Welchen Nachteil im Sinne „abstrakter Datenkapseln“ erkaufen Sie sich, wenn Sie den Operator `int()` programmieren? Welche Vorteile bringt dieser Operator verglichen mit der Benutzung von `get_value()`?

Schreiben Sie einen Operator `Subrange operator + (const Subrange r)`.

Aufgabe 5. *Designfehler*

Die Funktion `displayDate()` auf der folgenden Seite soll y2k Kompatibilität herstellen. Dokumentieren Sie den Designfehler des zugrundeliegenden Algorithmus.

```
Date: Sun, 2 Jan 100 12:45:47 PST
From: "Peter G. Neumann" <neumann@csl.sri.com>
Subject: Y2K early reports

On the whole, Y2K came and went without major immediately noted problems.
Predictions still abound for deferred problems.  In this message, I have
attempted to summarize in one place some of the reported Y2K weirdnesses.
...

There is some lovely Y2K humor on www.2600.com .

On 2 Jan, http://www.giga-byte.com/gigabyte-web/newindex.htm
  (they manufacture motherboards) has the date as Jan 2 2100.
The javascript function is:

// standard date display function with y2k compatibility

function displayDate() {
  var this_month = new makeArray(12);
    this_month[0]  = "January";
    this_month[1]  = "February";
    this_month[2]  = "March";
    this_month[3]  = "April";
    this_month[4]  = "May";
    this_month[5]  = "June";
    this_month[6]  = "July";
    this_month[7]  = "August";
    this_month[8]  = "September";
    this_month[9]  = "October";
    this_month[10] = "November";
    this_month[11] = "December";
  var today = new Date();
  var day   = today.getDate();
  var month = today.getMonth();
  var year  = today.getYear();
  if (year < 100){
    year += 1900;
  }
  else{
    year += 2000;
  }
  return(this_month[month]+" "+day+", "+year);
}
// -->

The flaw is beautifully obvious.  The comment is wonderful.
```