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 II)

SS2001 – Übungsblatt 11

Abgabetermin: 16. Juli 2001

Wählen Sie von den folgenden Aufgaben solche im Umfang von mindestens 20 Punkten aus:

Aufgabe 1. *Unterbereichstypen, 3 Punkte*
Testen Sie das Programm

```
#include <iostream>

using namespace std;

template<int low, int high>
class Subrange{

  int value;

public:

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

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

  Subrange operator+ (int diff){
    return Subrange(value+diff);
  };

};
```

```
int main(){

 try{

  Subrange<1,12> Monat;
  Subrange<1,12> neuerMonat;
  Subrange<1,365> laufenderTag(32);

  cout << Monat.get_value() << endl;
  cout << laufenderTag.get_value() << endl;

  // laufenderTag = Monat;

  neuerMonat = 12;
  cout << neuerMonat.get_value() << endl;
  Monat = neuerMonat+1;
  cout << Monat.get_value() << endl;

 } catch(const range_error& e) { cerr << endl << e.what() << endl; }

}
```

Welchen Nutzen hat der Typ `Subrange`? Programmieren Sie die Postfixoperatoren ++ und – – und testen Sie erneut! Welche Vor- und welche Nachteile hat es, wenn Sie statt `get_value()` den Typkonversionsoperator `int()` in der Klasse `Subrange` bereitstellen?

Aufgabe 2. *Polynome mit dynamischer Länge, 10 Punkte*
Ändern Sie die in der Vorlesung besprochene Klasse `Polynom` in

http://www.math.uni-wuppertal.de/∼buhl/teach/exercises/Inf2-SS01/Polynom.cc

so ab, daß die neue Version nicht mehr nur Polynome maximal zehnten Grades bereitstellt, sondern daß mittels `new` und `delete` beliebig hohe Polynomgrade zur Verfügung stehen.

Aufgabe 3. *Bestellung (Fortsetzung), 5 Punkte*
Betrachten Sie Ihre Klasse `Bestellung` von Übungsblatt 6. Wir möchten sie nun so abändern, daß nicht nur ein Bestellartikel sondern bis zu zwanzig solcher möglich werden.

Modifizieren sie dazu zunächst die Klasse `vektor` in

http://www.math.uni-wuppertal.de/∼buhl/teach/exercises/Inf2-SS01/vektor1.cc

so, daß der Komponententyp statt `double` der Typ `Bestellposition` wird. `Bestellposition` ist eine neue Klasse, die Bestellartikel, Bestellnummer, Anzahl, ... zusammenfaßt.

Testen Sie Ihre neue Klasse `Bestellung`.

Aufgabe 4. *Listen, 10 Punkte*
Testen Sie die folgende Implementierung einer einfach geketteten Liste (http://www.math.uni-wuppertal.de/~buhl/teach/exercises/Inf2-SS01/linkedList.cc):

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

#include <iostream>
#include <string>
#include <cassert>

#include <cstdlib>

using namespace std;

template<class T>
class linkedList{

  struct Knoten{
    T Daten;
    Knoten *next;
    Knoten(const T& daten, Knoten *p = 0):
        Daten(daten), next(p) {};
  } *Anfang;

private:

  void clear_all(Knoten* abStelle){
    if (abStelle != 0) {
      clear_all(abStelle->next);
      delete abStelle;
    };
  };
  // danger: dangling pointer at end of restlist, if not used
  //         at starting position of list

public:

  linkedList(): Anfang(0) {};

  ~linkedList() { clear_all(Anfang);
                  cerr << "*** clear_all: " << Anfang << endl; };
```

```
  // noch zu implementieren: Kopierkonstruktor und Wertzuweisungsoperator

  void push_front(const T& daten){
    Knoten* temp = new Knoten(daten, Anfang);
    assert(temp != 0);
    Anfang = temp;
  };

  void pop_front() {
    if (Anfang != 0) {
      Knoten* temp(Anfang);
      Anfang = Anfang->next;
      delete temp;
    };

  }

  class iterator;
  friend class iterator;

  class iterator {
    Knoten* aktuell;
  public:
    iterator(Knoten* init = 0): aktuell(init) {};
    T& operator*() { return aktuell->Daten; };
    const T& operator*() const {return aktuell->Daten; };
    iterator& operator++(){ // praefix
      if (aktuell != 0) aktuell = aktuell->next;
      return *this;
    };
    iterator operator++(int){ // postfix
      iterator temp = *this;
      ++*this;
      return temp;
    };
    bool operator!=(const iterator& x) const {
      return (aktuell != x.aktuell);
    };
  };

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

int main()
{

  linkedList<string> stringList;
  stringList.push_front("Erster eingefuegter Listenknoten");
```

```
  stringList.push_front("Zweiter eingefuegter Listenknoten");
  //stringList.pop_front();
  stringList.push_front("Dritter eingefuegter Listernknoten");
  // ...
  stringList.push_front("Vierter eingefuegter Listernknoten");
  stringList.push_front("ENDE!");

  linkedList<string>::iterator pos(stringList.begin());
  for (; pos != stringList.end(); pos++)
    cout << *pos << endl;

  linkedList<string>::iterator pos2(stringList.begin());
  pos2++; pos2++; pos2++;
  cout << endl;

  pos=pos2;
  for (; pos != stringList.end(); pos++)
    cout << *pos << endl;

  // Defaultkopierkonstruktor und -wertzuweisungsoperator
  // sind hier unbrauchbar, d.h. gefährlich:

  try{
linkedList<string> str2(stringList);
cout << "using copy constructor" << endl;
  }catch(const char * e){ cout << endl << e << endl;};

  try{
        linkedList<string> str2;
        str2 = stringList;
cout << "using assignment operator" << endl;
  }catch(const char * e){ cout << endl << e << endl;};

  return 0;
}
```

Ergänzen Sie eine Methode `void clear()`, die die ganze Liste löschen soll; vergessen Sie weder `delete` noch eine sinnvolle Wertzuweisung an das private Attribut `Anfang`.
Konzipieren Sie eine Methode `void erase_from(iterator p)`, die das Listenendstück, das an der Stelle `p` beginnt, löschen soll. Programmieren und testen Sie (auch die Extremfälle). Vergessen Sie nicht, die `next`-Referenz der Vorgängerstelle auf den Wert `0` zu setzen; finden sie diese durch Suchen ab `Anfang` oder durch Benutzung zweier Iteratoren `aktuell` und `vorgaenger`, die jeweils gleichzeitig inkrementiert werden. Welche Vor- bzw. Nachteile haben die beiden Methoden? Realisieren Sie eine Methode `erase(iterator p)`.
Warum muß hier ein eigener Kopierkonstruktor und eine eigene Wertzuwei-

sung programmiert werden?

Aufgabe 5. *Binäre Suchbäume, 4 Punkte*
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 zum Ablauf:

http://www.math.uni-wuppertal.de/~buhl/teach/exercises/Inf2-SS01/binTree.cc

```
////////////////////////////////
// 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;
      cerr << endl << "deleted: " << abStelle << endl;
    };
  };

 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); };

  // noch zu implementieren: deep-Kopierkonstruktor und deep-Wertzuweisung

  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:

    Knoten* getAktuell() const { return aktuell; };

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

    // fehlerhaft wäre (warum?):
    //    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
  {
    if (pos.getAktuell() != 0){
     show_inOrder(pos.getAktuell()->left);
     cout << pos.getAktuell()->Daten << endl;
     show_inOrder(pos.getAktuell()->right);
    }
  }

  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(1));
  cout << *wo << endl << endl;
  S.show_inOrder(wo);

  try{
        searchTree<int> S2;
        S2 = S;
cout << "using assignment operator" << endl;
  }catch(const char * e){ cout << endl << e << endl;};
```

```
  try{
        searchTree<int> S2(S);
cout << "using assignment operator" << endl;
  }catch(const char * e){ cout << endl << e << endl;};

  cout << "end of program" << endl;

}
```

Testen Sie.

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.

Warum muß hier ein eigener Kopierkonstruktor und eine eigene Wertzuweisung programmiert werden? Implementieren Sie einen von beiden.