Lehrstuhl für Informatik 3
Rheinisch-Westfälische Technische Hochschule
Aachen
Univ.-Prof. Dr.-Ing. M. Nagl

Aachen, 15. 06. 2007

Nagl, Ranger, Wörzberger

Übungen zur Vorlesung
„Grundgebiete der Informatik 2:
Algorithmen und Programmiertechniken“

— Lösung zu Blatt 9 —

20. Aufgabe

Einige Beispiele mit Implementation sind:

Anwendung	Implementation
Puffer für Ein- oder ausgehende Nachrichtenpakete	Feld als Ringliste
Mehrere Meßfühler	Feld (falls Anzahl feststeht), Zeiger
Stufen der Befehlsverarbeitung im Prozessor	Hardware
Bauteileliste einer Schaltung	Papier oder Zeiger
Schritte einer Arbeitsanweisung	Papier oder Zeiger
Werte einer Meßreihe	Feld oder Datei
Meßreihen eines Versuchs	Dateien

21. Aufgabe

```
/* ADT Doppelt verkettete Liste (Double Linked List) -- implementation */
#include <iostream>
#include "DLList.h"

using namespace std;

/* Definition des opaken Typs für ein Listenelement. */
struct DLList::ListItem {
    int value;
    ListItem *next, *prev;
};

DLList::DLList() {
    first = NULL;
    last = NULL;
    current = NULL;
    nonFull = true;
    nonEmpty = false;
}

void DLList::append(int x) {
    ListItem *item = new ListItem();      // Erzeuge neues Listenelement
```

```
    item->value = x;                        // ... und initialisiere es.
    item->next = NULL; item->prev = NULL;
    if (first == NULL) {                    // Liste noch leer?
      first = item; current = item;           // Neues Element ist jetzt erstes
      nonEmpty = true;
    } else {
      last->next = item; item->prev = last; // Neues Element hinten anhaengen
    }
    last = item;                            // Das neue Element ist das letzte
}

void DLList::deleteCurrent() {
    if (current == NULL) {
      return;
    } if (current == first) {              // Soll das erste Element geloescht werden?
      first = first->next;                   // Erstes ueberspringen
      if (first!=NULL) {                     // letztes Element?
         first->prev = NULL;                // Es gibt keinen Vorgaenger
      } else {
         last=NULL;                          // Liste ist leer
      }
      delete current;                        // Element loeschen
      current = first;                       // Cursor richtig setzen
    } else if (current == last) {          // Soll das letzte Element geloescht werden?
      last = last->prev;                     // Letztes ueberspringen
      last->next = NULL;                     // Es gibt keinen Nachfolger
      delete current;                        // Element loeschen
      current = last;                        // Cursor richtig setzen
    } else {
      ListItem *prevItem = current->prev;   // Einen Anker behalten
      prevItem->next = current->next;       // Aktuelles Element überspringen
      current->next->prev = prevItem;       // und noch mal
      delete current;                        // Element loeschen
      current = prevItem;                    // Vorheriges Element ist letztes
    }
}

void DLList::deleteIndex(int index) {
    ListItem *oldCurrent = current;        // Aktuellen Cursor merken
    toStart();                             // Cursor auf Anfang setzen
    while (index > 0) {                    // An die passende Stelle iterieren
      next();                                // Mit readNext Cursor weiterschieben
      index--;                               // Zaehler anpassen
    }
    deleteCurrent();                       // Element loeschen
    current = oldCurrent;                  // Alten Cursor restaurieren
}

void DLList::toStart() {
    current = first;
}

void DLList::toEnd() {
    current = last;
}

void DLList::next() {
    if (current == NULL) {
      cerr << "DLList::next: current == NULL" << endl; return;
    }
    if (current == last) {
```

```
      cerr << "DLList::next: am Ende der Liste" << endl; return;
    }
    current = current->next;
}

void DLList::prev() {
    if (current == NULL) {
      cerr << "DLList::prev: current == NULL" << endl; return;
    }
    if (current == first) {
      cerr << "DLList::next: am Anfang der Liste" << endl; return;
    }
    current = current->prev;
}

int DLList::read() {
    if (current == NULL)
      return -1;
    return current->value;
}

int DLList::readNext() {
    int value = read();                    // Wert lesen
    next();                                // Cursor weiterschieben
    return value;                          // Wert zurueckliefern
}

int DLList::readPrev() {
    int value = read();
    prev();
    return value;
}

bool DLList::isNonEmpty() {
    return nonEmpty;
}

bool DLList::isNonFull() {
    return nonFull;
}


bool DLList::isAtEnd() {
    return current == last;
}

bool DLList::isAtStart() {
    return current == first;
}

ostream& operator<<(ostream& target, DLList& list) {
    list.toStart();
    while (!list.isAtEnd()) {
      target << list.readNext() << " ";
    }
    return target;
}

int main() {
    DLList list;
    for (int i = 0; i < 10; i++) {
```

```
        list.append(i);
    }
    list.toStart(); list.deleteCurrent();
    list.toEnd(); list.deleteCurrent();
    list.toStart(); list.deleteIndex(5);
    list.append(10);
    cout << "Liste nach allen Operationen: " << list << endl;
}
```

22. Aufgabe

Bäume sind als rekursive Datenstrukturen oft sehr gut mit rekursiven Funktionen zu behandeln. Allerdings sind rekursive Funktionen auf den ersten Blick verdächtig simpel, auf den zweiten oft unergründlich und erst bei näherem Nachdenken wirklich verstanden. Hier verzweigt die exportierte Funktion zu einer verborgenen Funktion, die rekursiv den Baum durchläuft und in einem zusätzlichen Parameter die Einrücktiefe herumreicht. Falls Sie mit Rekursion noch nicht vertraut sind, erklären Sie sich, warum die linken und rechten Teilbäume mit der gleichen Einrücktiefe ausgegeben werden.

```
/* Implementation des Moduls BinBaumAusgabe.  */
#include "BinBaumAusgabe.h"     // Eigene Schnittstelle
#include <iomanip>              // Breite der Ausgabe
#include <iostream>             // Allgemeine Ausgabemoeglichkeiten

using namespace std;

/* Gibt einen (Teil-)Baum mit Einrueckung aus.  Dies ist der rekursive Kern
   der exportierten Funktion gebeBaumInorderAus.
Vorbedingung : knoten ein gueltiger Baum, ausgabe ein gueltiger ausgabestrom
Eingabe      : knoten  --- Wurzel des auszugebenden (Teil-)Baums
               tiefe   --- Einruecktiefe des Wertes des Knotens
               ausgabe --- Strom, auf dem die Ausgabe zu erzeugen ist
Ausgabe      : Darstellung in ausgabe
Nachbedingung: Baum unter knoten unveraendert */
static void gebeBaumInorderEingeruecktAus(BinBaumKnotenT const & knoten,
                                          unsigned int tiefe,
                                          ostream & ausgabe)
{
   /// Die Funktion ist rekursiv implementiert, was hier deutlich kuerzer
   /// als die iterative Alternative ist --- die muesste aufwendig im Baum
   /// navigieren.
   if ( linkerAstExistiert(knoten) ) {
      /// Gebe den linken Teilbaum mit tieferer Einrueckung aus.
      gebeBaumInorderEingeruecktAus(*linkerAst(knoten), tiefe+1, ausgabe);
   }
   /* Den Wert am Knoten eingerueckt ausgeben.  */
   ausgabe << setw(tiefe) << "" << leseWert(knoten) << endl;
   ///
   if ( rechterAstExistiert(knoten) ) {
      gebeBaumInorderEingeruecktAus(*rechterAst(knoten), tiefe+1, ausgabe);
   }
}
/* Aufrufschnittstelle der Ausgabe */
void gebeBaumInorderAus(BinBaumKnotenT const & knoten,
                        ostream& ausgabe)
{
   /* Diese Funktion ruft nur den rekursiven Kern mit der anfaenglichen Einruecktiefe auf. */
   gebeBaumInorderEingeruecktAus(knoten, 0, ausgabe);
```

```
}
```