

Programmiersprache 1 (C++)

Prof. Dr. Stefan Enderle
NTA Isny

11. Dynamische Datenstrukturen

11.1 Lineare Liste

- Eine lineare Liste (auch „einfach verkettete Liste“) ist eine Art Array mit flexibler Länge.
- Die Elemente in der Liste haben keine festen Indizes, sondern sind über Zeiger „verkettet“.
- (Bild)
- Vorteile zu Array: Schnelles Erweitern, Einfügen, Löschen von Elementen

Lineare Liste: Elemente

- Sollen z.B. double Werte gespeichert werden, bietet sich an:

```
struct LLElement
{
    double value;
    LLElement* next;
};
```

Elemente anlegen

- Anlegen von 3 Elementen:

```
LLElement* elem1 = new LLElement;  
elem1->value = 1.0;  
elem1->next = NULL;
```

```
LLElement* elem2 = new LLElement;  
elem2->value = 2.0;  
elem2->next = NULL;
```

```
LLElement* elem3 = new LLElement;  
elem3->value = 3.0;  
elem3->next = NULL;
```

Elemente verketten

- Verketten von 3 Elementen:

```
LLElement* head;
```

```
head = elem1;  
elem1->next = elem2;  
elem2->next = elem3;  
elem3->next = NULL;
```

Liste ausgeben

- Ausgabe aller Elemente durch Iteration:

```
LLElement* p = head;

while (p!=NULL) {
    cout << p->value << endl;
    p = p->next;
}
```

- **Anstatt** `while (p!=NULL)` schreibt man meist `while (p)`

Funktionen der Linearen Liste

- `init()` Initialisiert die Liste ohne Elemente
- `first()` Liefert Zeiger auf erstes Element
- `next(e1)` Liefert Zeiger auf nächstes Element
- `isEmpty()` `true`, wenn die Liste leer ist
- `insertHead(e1)` Fügt ein Element vorne ein
- `insertTail(e1)` Fügt ein Element hinten ein
- `removeHead()` Löscht erstes Element
- `remove()` Löscht die gesamte Liste

Implementierung der Funktionen

- Globaler head-Zeiger:

```
LLElement* head;
```

- ```
void init()
{
 head = NULL;
}
```

# Implementierung der Funktionen

- ```
LLElement* first()  
{  
    return head;  
}
```

Implementierung der Funktionen

- `LLElement* next(LLElement* e1)`
 {
 return e1->next;
 }

Implementierung der Funktionen

- ```
void insertHead(LLElement* el)
{
 el->next = head;
 head = el;
}
```

# Implementierung der Funktionen

- ```
void insertTail(LLElement* el)
{
    if (isEmpty()) insertHead(el);

    else {
        LLElement* p = head;
        while (p->next!=NULL) {
            p = p->next;
        }
        p->next = el;
        el->next = NULL;
    }
}
```

Implementierung der Funktionen

- ```
void removeHead(LLElement* e1)
{
 if (isEmpty()) return;

 LLElement* p = head;
 head = head->next;
 delete p;
}
```

# Implementierung der Funktionen

- ```
void remove()  
{  
    while (!isEmpty()) removeHead();  
}
```

Beispiel Hauptprogramm

```
int main(int argc, char *argv[])
{
    init();

    LLElement* p = new LLElement;
    p->value = 1.0;
    insertHead(p);

    p = new LLElement;
    p->value = 2.0;
    insertHead(p);

    cout << "first = " << first()->value << endl;
    cout << "next = " << next(first())->value << endl;

    remove();
}
```

Problem

- Eine Liste vorwärts auszugeben ist einfach.
- Aber Rückwärts-Ausgeben ist schwierig, da es keine Rückwärts-Verkettung gibt!

11.2 Doppelt verkettete Liste

- Eine doppelt verkettete Liste besitzt verglichen mit der einfach verketteten Liste zusätzlich einen „Rückwärtszeiger“ (prev).
- (Bild)
- Vorteile zur EVL: Beliebiges vorwärts/rückwärts Bewegen

Doppelt verk. Liste: Elemente

- Sollen z.B. double Werte gespeichert werden, bietet sich an:

```
struct DLElement
{
    double value;
    DLElement* next;
    DLElement* prev;
};
```

Elemente anlegen

- Anlegen von 3 Elementen:

```
DLElement* elem1 = new DLElement;  
elem1->value = 1.0;  
elem1->next = NULL;  
elem1->prev = NULL;
```

```
DLElement* elem2 = new DLElement;  
elem2->value = 2.0;  
elem2->next = NULL;  
elem3->prev = NULL;
```

```
DLElement* elem3 = new DLElement;  
elem3->value = 3.0;  
elem3->next = NULL;  
elem3->prev = NULL;
```

Elemente verketteten

- Verketteten von 3 Elementen:

```
DLElement* head;
```

```
head = elem1;  
elem1->next = elem2;  
elem1->prev = NULL;
```

```
elem2->next = elem3;  
elem2->prev = elem1;
```

```
elem3->next = NULL;  
elem3->prev = elem2;
```

Liste ausgeben

- Ausgabe aller Elemente durch Iteration:

```
DLElement* p = head;

while (p!=NULL) {
    cout << p->value << endl;
    p = p->next;
}
```

Liste rückwärts ausgeben

- Erinnerung: Ging bei einfach verketteter Liste nur durch Speicherung aller Elemente (z.B. durch Rekursion).
- Ausgabe rückwärts:

```
DLElement* p = head;           // p auf Anfang
if (p==NULL) return;           // Liste leer?
while(p->next) p=p->next;       // suche Ende

while (p!=NULL) {
    cout << p->value << endl;
    p = p->prev;
}
```

Funktionen der dopp. verk. Liste

- Fast identisch zur einfach verketteten Liste:
- `init()` Initialisiert die Liste ohne Elemente
- `first()` Liefert Zeiger auf erstes Element
- `next(e1)` Liefert Zeiger auf nächstes Element
- `prev(e1)` Liefert Zeiger auf voriges Element
- `isEmpty()` `true`, wenn Liste leer ist
- `insertHead(e1)` Fügt ein Element vorne ein
- `insertTail(e1)` Fügt ein Element hinten ein
- `removeHead()` Löscht erstes Element
- `remove()` Löscht die gesamte Liste

Implementierung der Funktionen

- Globaler head-Zeiger:

```
DLElement* head;
```

- ```
void init()
{
 head = NULL;
}
```

# Implementierung der Funktionen

- `DLElement* first()`  
  {  
    return head;  
  }

# Implementierung der Funktionen

- `DLElement* next(DLElement* e1)`  
  {  
    return e1->next;  
  }

# Implementierung der Funktionen

- ```
DLElement* prev(DLElement* e1)
{
    return e1->prev;
}
```

Implementierung der Funktionen

- ```
bool isEmpty()
{
 return (head==NULL);
}
```

# Implementierung der Funktionen

- ```
void insertHead(DLElement* el)
{
    el->next = head;
    el->prev = NULL;

    head->prev = el;

    head = el;
}
```

Implementierung der Funktionen

- ```
void insertTail(DLElement* el)
{
 if (isEmpty()) insertHead(el);

 else {
 DLElement* p = head;
 while (p->next!=NULL) {
 p = p->next;
 }
 p->next = el;
 el->next = NULL;
 el->prev = p;
 }
}
```

# Implementierung der Funktionen

- ```
void removeHead()
{
    if (isEmpty()) return;

    DLElement* p = head;
    head = head->next;

    if (head!=NULL) head->prev = NULL;

    delete p;
}
```

Implementierung der Funktionen

- ```
void remove()
{
 while (!isEmpty()) removeHead();
}
```

# Beispiel Hauptprogramm

```
int main(int argc, char *argv[])
{
 init();

 DLElement* p = new DLElement;
 p->value = 1.0;
 insertHead(p);

 p = new DLElement;
 p->value = 2.0;
 insertHead(p);

 cout << "first = " << first()->value << endl;
 cout << "next = " << next(first()->value) << endl;

 remove();
}
```

# 11.3 Stapelspeicher (Stack)

- In einem Stapelspeicher (Stack) können Objekte gespeichert und nur in umgekehrter Reihenfolge wieder ausgelesen werden.
- LIFO Prinzip: Last In First Out
- Operationen:
  - push Ein Element auf den Stapel legen
  - pop Ein Element vom Stapel lesen
  - top Oberstes Element lesen aber nicht löschen
- Anwendung z.B. bei Funktionsaufrufen, Kellerautomat

# Implementierung mit Array

- Beispiel: Integer-Stack:

```
int stack[1000];
int stackTop = -1;

void push(int val)
{
 stackTop++;
 stack[stackTop]=val;
}

int pop()
{
 stackTop--;
 return stack[stackTop+1];
}

int top()
{
 return stack[stackTop];
}
```

# Implementierung mit Array

- Problem:
  - Ideale Größe des Stacks nicht bekannt
  - Stack-Überläufe werden nicht abgefangen!

# Implementierung mit Array

- Implementierung mit Fehlerabfrage:

```
const int STACK_SIZE = 1000;
int stack[STACK_SIZE];
int stackTop = -1;

void push(int val)
{
 if (stackTop >= STACK_SIZE-1) error();

 else {
 stackTop++;
 stack[stackTop]=val;
 }
}
```

# Implementierung mit Array

```
int pop()
{
 if (stackTop == -1) error();

 else {
 stackTop--;
 return stack[stackTop+1];
 }
}
```

```
int top()
{
 if (stackTop == -1) error();

 else return stack[stackTop];
}
```

# Implementierung mit Lin. Liste

- Beispiel mit Integer-Stack:

```
LLElement* head = NULL;

void push(int val)
{
 LLElement* el = new LLElement;
 el->value = val;
 insertHead(el);
}

int pop()
{
 int value = first()->value;
 removeHead();
 return value;
}

int top()
{
 return first()->value;
}
```

# Implementierung mit Lin. Liste

- Vorteile:
  - Keine Maximal-Größe nötig.
  - Fehlerabfrage in `removeHead()` bereits enthalten.
- Rest-Problem:
  - `new` schlägt fehl, wenn kein Speicher mehr allokiert werden kann!
- Beispiel:

```
void push(int val)
{
 LLElement* el = new LLElement;
 if (el==NULL) error(); // new fehlgeschlagen ?
 else {
 el->value = val;
 insertHead(el);
 }
}
```

# 11.4 Warteschlange (Queue)

- In einer Warteschlange (queue) können Objekte gespeichert und nur in dieser Reihenfolge wieder ausgelesen werden.
- FIFO Prinzip: First In First Out
- Operationen:
  - enqueue            Ein Element einspeichern
  - dequeue            Ein Element auslesen
- Anwendung:
  - Z.B. „TODO“ Listen
  - Entkopplung von Prozessen

# Implementierung

- Array ungeeignet (Ineffizient wegen Umspeichern)
- Lineare Liste (wie bisher) halbwegs geeignet:

Beispiel:

- enqueue = insertTail(e1)
- dequeue = head();  
removeHead();

- Aber: insertTail nicht effizient !  
→ Bessere Implementierung mit head und tail Zeiger.

# Implementierung mit Lin. Liste

- Elemente der Linearen Liste wie üblich!

Beispiel mit string:

```
struct LLElement
{
 string value;
 LLElement* next;
};
```

# Implementierung mit Lin. Liste

- Initialisierung von head und tail:

```
LLElement* head = NULL;
```

```
LLElement* tail = NULL;
```

# Implementierung mit Lin. Liste

- Enqueue:  
Einspeichern in Warteschlange am Ende (tail)

```
void enqueue(string s)
{
 LLElement* el = new LLElement;
 el->value = s;
 el->next = NULL;
 if (tail) tail->next = el;
 tail = el;
 if (head==NULL) head = el;
}
```

# Implementierung mit Lin. Liste

- Dequeue:  
Auslesen aus Warteschlange am Anfang (head):

```
string dequeue()
{
 string s = head->value;
 LLElement* p = head;
 head = head->next;
 delete p;
 if (head==NULL) tail=NULL;
 return s;
}
```

# Beispiel Hauptprogramm

```
int main(int argc, char *argv[])
{
 enqueue("Aufgabe 1");
 enqueue("Aufgabe 2");
 enqueue("Aufgabe 3");

 cout << dequeue() << endl;
 cout << dequeue() << endl;
 cout << dequeue() << endl;
}
```

## Ausgabe:

```
Aufgabe 1
Aufgabe 2
Aufgabe 3
```

# 11.5 Binärbäume

- Bei der Suche in großen Datenmenge sind lineare Listen langsam, da jedes Element sequentiell durchgearbeitet werden muss.
- Gibt es eine Ordnung auf den Elementen (z.B. „größer als“), so bietet sich ein Binärer Baum an.
- (Bild)

# Binärbäume

- **Definitionen:**
  - Jeder Knoten besitzt höchstens zwei Nachfolger.
  - Nur der Wurzelknoten besitzt keinen Vorgänger.
- **Begriffe:**
  - Ein Knoten, der keinen Nachfolger besitzt heißt **Blatt**.
  - Ein Knoten der kein Blatt ist heißt **innerer Knoten**.
  - Knoten sind durch **Kanten** verbunden.
  - Die maximale Anzahl von Kanten, die von der Wurzel bis zu einem Blatt durchlaufen werden kann heißt **Höhe** des Baumes.

# Implementierung

- Die Elemente entsprechen denen einer Liste.
- Beispiel mit int-Werten:

```
struct TreeElement
{
 int value;
 TreeElement* left;
 TreeElement* right;
};
```

# Implementierung

- Initialisierung des Wurzelknotens:

```
TreeElement* root = NULL;
```

# Implementierung

- treeOutput:  
Ausgabe von „links nach rechts“ :

```
void treeOutput(TreeElement* t)
{
 if (t->left) treeOutput(t->left);
 cout << t->value << " ";
 if (t->right) treeOutput(t->right);
}
```

# Implementierung

- **treeInsert:**  
Einspeichern mit Ordnung (n = neues Element):

```
void treeInsert(TreeElement* t, TreeElement* n)
{
 if (n->value > t->value) {
 if (t->right)
 treeInsert(t->right, n);
 else t->right = n;
 }

 else {
 if (t->left)
 treeInsert(t->left, n);
 else t->left = n;
 }
}
```