Herzlich willkommen!

Dozent: Dipl.-Ing. Jürgen Wemheuer

Mail: wemheuer@ewla.de
Online: http://cpp.ewla.de/

Disclaimer

- Diese Vorlesungs-/Unterrichtsfolien basieren auf Skripte meiner geschätzten Fachkollegen Prof. Dr. Dietrich Kuhn († 2010) und Prof. Dr. Stefan Enderle der Naturwissenschaftlich-Technischen Akademie (nta) in Isny.
- Das Skript wurde durch den Dozenten ausschließlich für die Gestaltung seines Unterrichts / seiner Vorlesung zusammengestellt bzw. verfasst und ist nicht als Referenz einer Programmiersprache gedacht.
- Dem Vorlesungsskript mangelt es an jeglichem Kontext.
  Dieser ist vielmehr der bestimmende Lehrinhalt in den Vorlesungen.
- Nicht alle Inhalte des Vorlesungsskripts sind prüfungsrelevant.
- Nicht alle prüfungsrelevanten Fakten sind im Vorlesungsskript enthalten.
- Ausschlaggebend für Prüfungen sind deshalb allein die im Unterricht bzw. in den Übungen und/oder Projektbeispielen vorgebrachten Inhalte.
- Aktuelle Änderungen des Vorlesungsskripts sind jederzeit vorbehalten.
- Mit allen auftretenden Fragen zum Fachgebiet und dem Vorlesungsskript sollten sich die SchülerInnen und StudentInnen stets an den Dozenten wenden.
- Das Vorlesungsskript wurde mit bestem Wissen und Gewissen und sorgfältig erarbeitet, jedoch können Irrtümer und Fehler nicht ausgeschlossen werden.
- Jegliche Haftung und Gewährleistung ist ausgeschlossen.

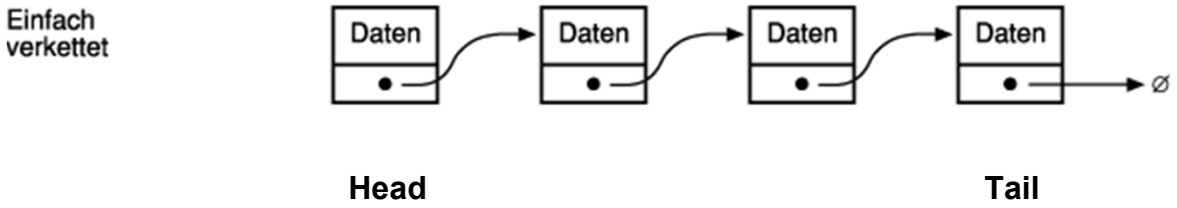
Inhalt:

Dynamische Datenstrukturen:

- Lineare Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- Stapelspeicher (Stacks, LIFO)
- Warteschlangen (Queues, FIFO)
- Bäume (Trees)

Lineare Listen

- Eine lineare Liste (auch „einfach verkettete Liste“) ist eine Art Array mit flexibler Lange.
- Die Elemente in der Liste haben keine festen Indizes, sondern sind über Zeiger „verkettet“.



- Vorteile gegenüber Arrays: Schnelles Erweitern, Einfügen, Löschen von Elementen

Einfach verkettete Liste (EVL)

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

```
struct LLElement
{
 double value;     // der eigentliche Wert
 LLElement* next; // Zeiger auf nächsten Wert
};
```

EVL-Prinzip: Elemente anlegen

- Anlegen von 3 (noch unabhängigen) 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;
```

EVL-Prinzip: Elemente verketten

- Verketten von diesen 3 Elementen:

```
LLElement* head;
head = elem1;    // Zeiger auf 1. Element
elem1->next = elem2;
elem2->next = elem3;
elem3->next = NULL;
```

EVL: Elemente 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 meistens:

```
while(p)
```

Funktionen der Linearen Liste (EVL)

- **`init()`** Initialisiert die Liste ohne Elemente
- **`first()`** Liefert Zeiger auf erstes Element
- **`next(el)`** Liefert Zeiger auf nächstes Element
- **`isEmpty()`** true, wenn die Liste leer ist
- **`insertHead(el)`** Fügt ein Element vorne ein
- **`insertTail(el)`** Fügt ein Element hinten ein
- **`removeHead()`** Löscht erstes Element
- **`remove()`** Löscht die gesamte Liste
- … und weitere, z.B. createOnHead(el) etc.

Implementierung der Funktionen (1)

- Globaler head-Zeiger:

```
LLElement* head;


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

(oder gleich: `LLElement* head=NULL`)

Implementierung der Funktionen (2)

- Liefert Zeiger auf erstes Element

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

Implementierung der Funktionen (3)

- Liefert Zeiger auf nächstes Element

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

Implementierung der Funktionen (4)

- true, wenn die Liste leer ist

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

Implementierung der Funktionen (5)

- Fügt ein Element vorne ein

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

Implementierung der Funktionen (6)


```
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 (7)

- Löscht erstes Element

```
void removeHead()
{
 if (isEmpty()) return;
 LLElement* p = head;
 head = head->next;
 delete p;
}
```

Implementierung der Funktionen (8)

- Löscht die gesamte Liste

```
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();
}
```

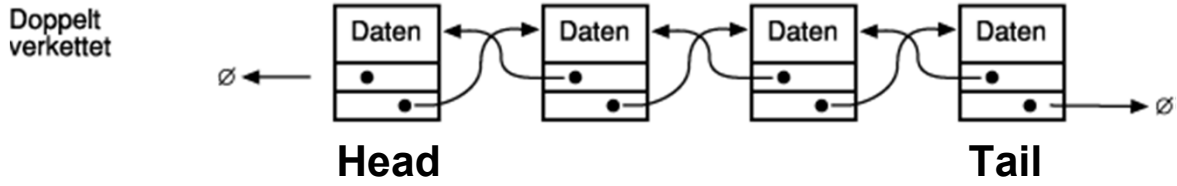
Problem:

- Eine Liste vorwärts auszugeben ist einfach.

- Aber eine Liste rückwärts auszugeben ist schwierig oder sehr umständlich,
  da es keine Rückwärts-Verkettung gibt!

Lösung: Doppelt verkettete Liste

- Eine doppelt verkettete Liste (DVL) besitzt verglichen mit der einfach verketteten Liste zusätzlich einen „Rückwärtszeiger“ (prev).



- Vorteile gegenüber EVL:
  Beliebiges Vorwärts- und Rückwärts-Bewegen

DVL: Elemente

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

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

DVL: 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;
```

DVL: Elemente verketten

- Verketten 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;
```

DVL: Liste vorwärts ausgeben

- Ausgabe aller Elemente durch Iteration:

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

DVL: 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
// optional: p = tail;
while(p!=NULL) {
 cout << p->value << endl;
 p = p->prev;
}
```

Funktionen der Linearen Liste (DVL)

Fast identisch zur einfach verketteten Liste:

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

Implementierung der Funktionen (1)

- Globaler head-Zeiger:

```
DLElement* head;
DLElement* tail; // optional…


void init()
{
 head = NULL;
 tail = NULL; // optional…
}
```

Implementierung der Funktionen (2)

- Findet erstes (bzw. letztes Element

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

DLElement* last() { return tail;}
```

Implementierung der Funktionen (3)


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

Implementierung der Funktionen (4)


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

Implementierung der Funktionen (5)


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

Implementierung der Funktionen (6)


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

 head->prev = el;
 head = el;
}
```

Implementierung der Funktionen (7)


```
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 (6)


```
void removeHead(DLElement* el)
{
 if (isEmpty()) return;
 DLElement* p = head;
 head = head->next;
 if(head!=NULL) head->prev=NULL;
 delete p;
}
```

Implementierung der Funktionen (7)


```
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();
}
```

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 loschen
- Anwendung z.B. bei Funktionsaufrufen, Kellerautomat

Stack-Implementierung mit Array?


```
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];
}
```

Stack-Implementierung

Problem:

- Ideale Größe des Stacks nicht bekannt
- Stack-Überlaufe werden nicht abgefangen!

Deshalb: Implementierung mit

- Fehlerabfrage oder
- Linearer Liste

Warteschlangen (Queues, FIFO)

- 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(el)
  - dequeue = head(); removeHead();
- Aber: insertTail nicht effizient !

→ Bessere Implementierung mit head und tail Zeiger.

Implementierung mit Linearer Liste

Elemente der Linearen Liste wie üblich!

Beispiel mit string:

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

Implementierung mit Linearer Liste

- Initialisierung von head und tail:

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

Implementierung mit Linearer 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 Linearer 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;
}
```

Implementierung mit Linearer Liste

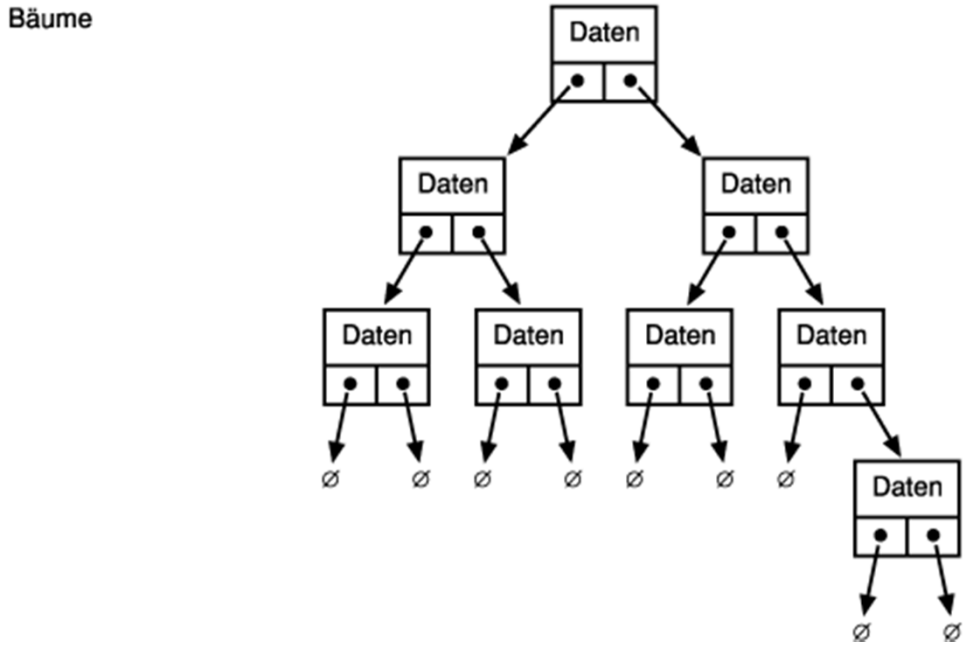

```
int main(int argc, char *argv[]) {
  enqueue("Aufgabe 1");
  enqueue("Aufgabe 2");
  enqueue("Aufgabe 3");
  cout << dequeue() << endl;
  cout << dequeue() << endl;
  cout << dequeue() << endl;
}
```

Ausgabe:

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

Binärbaume

- 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.



Binärbaume

- **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 (1)

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

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

Implementierung (2)

- Initialisierung des Wurzelknotens:

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

Implementierung (3)

- treeOutput:
  Ausgabe von „links nach rechts“ :

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

Implementierung (4)

- 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;
  }
}
```

Ende