

Programmiertechnik (C++)

Prof. Dr. Stefan Enderle

NTA Isny

3. Dynamische Datenstrukturen

3.1 Dynamische Arrays (Vektoren)

- Dynamische Arrays bilden wie statische Arrays einen ganzzahligen Index auf ein Element ab:
 `dyn_array: int -> T`
- Dynamische Arrays bieten jedoch die Möglichkeit, die Anzahl an Elementen beliebig zu verändern.
- Dynamische Arrays werden meist Vektoren genannt.

3.2 Assoziative Arrays

- Bei einem assoziativen Array können beliebige Datentypen als Index benutzt werden.
 `assoc_array: T -> T`
- Man nennt den Index dann den „Schlüssel“ (key).
- Einen Eintrag nennt man „Key-Value-Pair“.
- Beispiele:
 - Name -> Geburtstag
 - Matrikelnummer -> Student

3.3 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 verketteten

- Verketteten von 3 Elementen:

```
LLElement* head;
```

```
head = elem1;
```

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

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

```
elem3->next = NULL; // (unnötig)
```

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)`

Liste rückwärts ausgeben

- Problem: Verkettung einer linearen Liste geht nur in EINE Richtung!
- Keine iterative Möglichkeit.

Liste rückwärts ausgeben

- Idee: Rekursion
- Ausgabe aller Elemente rückwärts:

```
void outputReverse(LLElement* e)
{
    if (e==NULL) return;

    outputReverse(e->next);
    cout << e->value << endl;
}
```

- **Achtung:** Bei Rekursion ist Abbruchkriterium wichtig!

Funktionen der Linearen Liste

- `init()` Initialisiert die Liste ohne Elemente
- `first()` Liefert Zeiger auf erstes Element
- `next(e1)` Liefert Zeiger auf nächstes Element
- `insertHead(e1)` Fügt ein Element vorne ein
- `insertTail(e1)` Fügt ein Element hinten ein
- `remove(e1)` Löscht ein Element
- `delete()` Löscht die gesamte Liste

Implementierung der Funktionen

- `void init()`
- `LLElement* first()`
- `LLElement* next(LLElement* el)`
- `void insertHead(LLElement* el)`
- `void insertTail(LLElement* el)`
- `void remove(LLElement* el)`
- `void delete()`

Lineare Liste: Probleme

- Werden mehrere Listen benötigt, muss man mehrere `head` verwalten!
- Wird eine Liste eines anderen Typs benötigt, schreibt man fast den gleichen Code nochmal.
- Was passiert bei Fehlern?
(Init vergessen, next obwohl keines mehr kommt, remove bei NULL, kein Speicher mehr,...)

Lineare Liste: Lösungen

- Werden mehrere Listen benötigt, muss man mehrere `head` verwalten!
--> **LListe als Klasse**
- Wird eine Liste eines anderen Typs benötigt, schreibt man fast den gleichen Code nochmal.
--> **Templates**
- Was passiert bei Fehlern?
(Init vergessen, next obwohl keines mehr kommt, remove bei NULL, kein Speicher mehr,...)
--> **Exception Handling**

3.4 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; // (unnötig)
```

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

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

```
elem3->next = NULL; // (unnötig)
```

```
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

- Gleich wie bei einfach verketteter 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)
- `insertHead(e1)` Fügt ein Element vorne ein
- `insertTail(e1)` Fügt ein Element hinten ein
- `remove(e1)` Löscht ein Element
- `delete()` Löscht die gesamte Liste

Implementierung der Funktionen

- `void init()`
- `DLElement* first()`
- `DLElement* next(DLElement* el)`
- `DLElement* prev(DLElement* el)`
- `void insertHead(DLElement* el)`
- `void insertTail(DLElement* el)`
- `void remove(DLElement* el)`
- `void delete()`

Implementierung der Funktionen

- `void init()`
- `DLElement* first()`
- `DLElement* next(DLElement* el)`
- `DLElement* prev(DLElement* el)`
- `void insertHead(DLElement* el)`
- `void insertTail(DLElement* el)`
- `void remove(DLElement* el)`
- `void delete()`

3.5 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

Implementierung

- Ein Stack kann z.B. durch eine lineare Liste implementiert sein. Dann ergeben sich folgende Realisierungen von push und pop:
- Beispiel mit Integer-Stack:

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

```
- int pop()
  {
    LLElement* el = first();
    remove(el);
    return el->value;
  }
```

3.6 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:
 - Entkopplung von Prozessen

3.7 Binäre Bäume

- 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.
- Definitionen:
 - Jeder Knoten besitzt höchstens zwei Nachfolger.
 - Nur der Wurzelknoten besitzt keinen Vorgänger.
- (Bild)

Binäre Bäume

- Begriffe:
 - Ein Knoten, der keinen Nachfolger besitzt heißt **Blatt**.
 - Ein Knoten der kein Blatt ist heißt **innerer Knoten**.
 - Knoten sind durch **Kanten** verbunden.
 - Die maximal Anzahl von Kanten, die von der Wurzel bis zu einem Blatt durchlaufen werden kann heißt **Höhe** des Baumes.
 - Ein binärer Baum heißt **optimal**, wenn jeder innere Knoten mit höchstens einer Ausnahme genau zwei Nachfolger hat.
 - Ein binärer Baum heißt **linksorientiert**, wenn alle Knoten aller Ebenen so weit wie möglich links liegen.

„Heap“

- Ein binärer Baum erfüllt die **Heap-Bedingung**, wenn in jedem Knoten der Schlüssel kleiner ist als die Schlüssel in allen (auch nicht direkten) Nachfolgern.
- Ein optimaler, linksorientierter binärer Baum, der die Heap-Bedingung erfüllt heißt **Heap**.
- (Bild mit Namen)
- Eigenschaften eines Heaps:
 - Minimale Höhe
 - Das kleinste Element steht immer an der Wurzel

Einfügen eines Elements

- Algorithmus für das Einfügen in einen Heap:
 - Das neue Element an das Ende des Heaps anhängen. (Bleibt linksorientiert und optimal)
 - Heap-Bedingung überprüfen:
 - Ist neues Element größer als Vorgänger -> fertig
 - Ist neues Element kleiner -> „Bubble up“ bis Heap-Bedingung wieder erfüllt ist