

Grundlagen der Informatik

Prof. Dr. Stefan Enderle

NTA Isny

2 Datenstrukturen

2.1 Einführung

- Syntax:
 - Definition einer formalen Grammatik, um Regeln einer formalen Sprache (Programmiersprache) festzulegen.
 - Ein Compiler prüft die Syntax und meldet ggfs. Fehler.
- Semantik:
 - Bedeutung der syntaktisch korrekt gebildeten Konstrukte.
 - Kann vom Compiler nicht geprüft werden.

Datentypen

- In Programmiersprachen besitzen Daten (=Variablen und Konstanten) einen zugeordneten Typ.
- Je nach Datentyp können unterschiedliche Operationen durchgeführt werden.
- Der Compiler prüft, ob eine Operation für einen Datentyp zulässig ist.

Grundlegende Datentypen

- Zum Beispiel in C/C++:
 - `int` Ganze Zahlen
 - `float` Reelle Zahlen
 - `char` Zeichen (a..z, A..Z, *,/,&,....)
 - `string` Zeichenketten (Wörter, Sätze)
 - `bool` Wahrheitswerte

2.2 Felder (Arrays)

- Felder oder Arrays dienen der Speicherung von mehreren Daten gleichen Typs.
- Beispiele:
 - 10 ganze Zahlen
 - 100 reelle Zahlen
 - 1000 Strings mit Namen
 - usw.
- Die Elemente eines Arrays werden durchnummeriert.
- Die Elemente eines Arrays stehen direkt hintereinander im Speicher.

2.3 Assoziative Arrays

- Bei einem assoziativen Array werden nicht grundsätzlich ganze Zahlen als Index benutzt, sondern beliebige Datentypen als sogenannte „Schlüssel“ (key).
- Einen Eintrag nennt man „Key-Value-Pair“.
- Beispiele:
 - Name -> Geburtstag
 - Matrikelnummer -> Student

2.4 Strukturen

- Strukturen (structs, auch Verbunde) dienen der Speicherung von zusammengehörigen Daten gleichen oder unterschiedlichen Typs.
- Beispiele:
 - komplexe Zahl = Realteil und Imaginärteil
 - Anschrift = Adresse, Straße, PLZ, Ort
- Die Elemente einer Struktur stehen direkt hintereinander im Speicher.

2.5 Stapelspeicher

- 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

2.6 Warteschlange

- 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

2.7 Zeiger

- Zeiger (Pointer) zeigen auf eine Speicherstelle.
- Zeiger haben trotzdem einen Typ. Der Typ gibt an, wie die entsprechende Speicherzelle interpretiert wird.
- Ein Zeiger, der auf Nichts zeigt besitzt einen definierten Wert (z.B. NULL).

2.8 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 oder Keys, sondern sind über Zeiger „verkettet“.
- (Bild)
- Vorteile zu Array: Schnelles Erweitern, Einfügen, Löschen von Elementen

Lineare Liste: Elemente

- Ein Element einer linearen Liste ist eine Struktur aus:
 - Nutzdaten (data)
 - Zeiger auf Nachfolger (next)
- (Bild)
- Der Zeiger „next“ zeigt auf das nächste Element in der Liste oder auf NULL, wenn keines mehr folgt.
- Zusätzlich muss lediglich ein Zeiger „head“ auf das erste Element verwaltet werden

Lineare Liste: Operationen

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

Zeiger-Schreibweise

- Vereinbarung:
 - Ist `a` ein Zeiger, so ist der Wert von `a` die Speicheradresse, auf die der Zeiger zeigt.
 - `NULL` ist ein definierter Wert für den Zeiger auf „nichts“.
 - Zeigt `a` auf ein Listenelement bestehend aus `data` und `next`, so bezeichnen wir mit `a→data` bzw. `a→next` den referenzierten

Algorithmen

- `init()`:
 - Setze `head` auf `NULL`
- `first()`:
 - Gib `head` zurück
- `next(e1)`:
 - Gib `e1→next` zurück

Algorithmen

- `insertHead(e1)`:
 - Setze `e1→next` auf `head`
 - Setze `head` auf `e1`
- `insertTail(e1)`:
 - Traversiere die Liste bis zum letzten Element
 - Sei `last` der Zeiger auf das letzte Element
 - Setze `last→next` auf `e1`
 - Setze `e1→next` auf `NULL`

Algorithmen

- `removeHead(e1)` entfernt das erste Element aus der Liste und gibt den Speicherplatz mit `delete` wieder frei.
- `removeHead()`:
 - Wenn gilt `head≠NULL`
 - setze `p=head`
 - setze `head=head→next`
 - `delete p`
- `delete()`:
 - Solange `head≠NULL`

Lineare Liste: Problem

- Ein Rückwärts-Durchgehen der Elemente ist nicht ohne weiteres möglich.

2.9 Doppelt verkettete Liste

- Eine doppelt verkettete Liste ist eine lineare Liste mit Rückwärts-Zeiger.
- Jedes Element besitzt
 - Nutzdaten (data)
 - Zeiger auf Nachfolger (next)
 - Zeiger auf Vorgänger (prev)
- (Bild)
- Vorteile gegenüber der einfach verketteten Liste:
 - Auch Rückwärts-Tour ist möglich

Doppelt v. Liste: Operationen

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

Algorithmen

- `init()`:
 - Setze `head` auf `NULL`
- `first()`:
 - Gib `head` zurück
- `next(el)`:
 - Gib `el→next` zurück
- `prev(el)`:

Algorithmen

- insertHead(e1):
 - Setze `e1→next` auf `head`
 - Setze `head` auf `e1`
 - Setze `e1→next→prev` auf `e1`
 - Setze `e1→prev` auf `NULL`

Algorithmen

- `insertTail(e1)`:
 - Traversiere die Liste bis zum letzten Element
 - Sei `last` der Zeiger auf das letzte Element
 - Setze `last→next` auf `e1`
 - Setze `e1→next` auf `NULL`
 - Setze `e1→prev` auf `last`

Algorithmen

- **removeHead():**
 - Wenn gilt `head≠NULL`
 - setze `p=head`
 - setze `head=head→next`
 - setze `head→prev=NULL`
 - delete `p`
- **delete():**
 - Solange `head≠NULL`
 - `removeHead()`

2.10 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.

Binäre Bäume

- **Optimaler binärer Baum:**

Ein binärer Baum heißt **optimal**, wenn jeder innere Knoten mit höchstens einer Ausnahme genau zwei Nachfolger hat.

- **Linksorientierter binärer Baum:**

Ein binärer Baum heißt **linksorientiert**, wenn alle Knoten aller Ebenen so weit wie möglich links liegen.

- **Balancierter binärer Baum:**

Ein binärer Baum heißt **balanciert**, wenn sich für alle inneren Knoten die Höhe des linken und rechten Teilbaumes um

„Heap“

- Ein binärer Baum erfüllt die **Heap-Bedingung**, wenn der Schlüssel in einem beliebigen Knoten kleiner ist als die Schlüssel in allen Nachfolgern.
- Ein optimaler, linksorientierter, balancierter 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

„Heap“

Algorithmus für das Einfügen in einen Heap:

- Das neue Element an das Ende des Heaps anhängen. (Bleibt linksorientiert, optimal und balanciert)
- Heap-Bedingung überprüfen:
 - Ist neues Element größer als Vorgänger → fertig
 - Ist neues Element kleiner → „Bubble up“ bis HeapBedingung wieder erfüllt ist

Speicherung eines binären Baumes

- Binäre Bäume werden meist nicht durch eine Zeigerstruktur gespeichert, sondern als Array.
- Für einen Knoten n gilt nämlich: die beiden Nachfolger befinden sich an den Positionen $(2n)$ und $(2n+1)$:

