

Inhalt

- 6 Abstrakte Datentypen: Menge und Stapel
 - Begriffsbildung
 - ADT Menge
 - ADT Stapel

Abstrakte Datentypen - Datenstrukturen

Wir haben bisher bereits deutlich unterschieden zwischen den Anforderungen oder Eigenschaften, die wir verlangen wollen und der konkreten Implementierung.

- ▶ Eine abstrakte Beschreibung von Anforderungen nennen wir „**Abstrakten Datentyp (ADT)**“.
- ▶ Die konkreten Implementierungen bezeichnen wir als „**Datenstrukturen (DS)**“.

Beispiel Menge:

- ▶ Die **Anforderungen** wurden in der Definition des **interface Menge** formuliert
- ▶ die **Implementierungen** in den Klassen **class MengeDynArray** bzw **class MengeEVL**.

chatGPT: ADT - DS

ChatGPT 3.5 ▾

 You

was ist der Unterschied zwischen einem abstrakten Datentyp und eine Datenstruktur?

 ChatGPT

Ein abstrakter Datentyp (ADT) und eine Datenstruktur sind zwei miteinander verbundene Konzepte, aber sie sind nicht dasselbe:

1. Abstrakter Datentyp (ADT):

- Ein abstrakter Datentyp ist eine mathematische Spezifikation eines Datentyps, die seine möglichen Werte und die zulässigen Operationen auf diesen Werten definiert, ohne die Implementierungsdetails anzugeben.
- ADTs definieren also nur, welche Operationen möglich sind und welche Bedingungen dabei gelten müssen, aber sie sagen nichts über die interne Struktur oder Implementierung dieser Operationen aus.
- Ein ADT kann als Schnittstelle betrachtet werden, die es erlaubt, mit einer bestimmten Art von Daten auf eine abstrakte Weise zu arbeiten.

2. Datenstruktur:

- Eine Datenstruktur ist die konkrete Implementierung eines abstrakten Datentyps. Sie beschreibt, wie die Daten tatsächlich organisiert und gespeichert werden, um die definierten Operationen effizient ausführen zu können.
- Datenstrukturen bieten die konkreten Algorithmen und Datenorganisationen, um die spezifischen Operationen des abstrakten Datentyps umzusetzen.
- Die Wahl einer bestimmten Datenstruktur hängt von den Anforderungen an Speicherplatz, Zugriffszeiten und den zu erwartenden Operationen ab.

In einfacheren Worten ausgedrückt: Der abstrakte Datentyp definiert, was getan werden kann, während die Datenstruktur definiert, wie es getan wird. Der ADT ist eine ideale Vorstellung davon, was eine Datenstruktur tun soll, während die Datenstruktur die tatsächliche Implementierung ist, die diese Idee in die Praxis umsetzt.



Vereinbarung

Syntax und **Semantik** eines ADT werden i.a. mathematisch formal, dh programmiersprachen-unabhängig definiert.

- ▶ Wir beschreiben die **Syntax** in Java durch Definition eines **Interface** oder durch ein (Pseudo-)UML-Diagramm.
- ▶ Die **Semantik** der Operationen beschreiben wir „informell“ zB durch **Kommentare** oder eine Dokumentation.
- ▶ Sowohl die Interfaces als auch die implementierenden Klassen sind in der Regel **generisch** und **iterierbar** (also eine Erweiterung bzw Implementierung von **Iterable <T>**).
- ▶ Sie verfügen in der Regel über (mindestens) die Methoden **int size()** und **boolean isEmpty()**.

Komplexitätsanalyse

Ziel einer Analyse

- ▶ Bestimmung des (Platz- und) **Zeitbedarfs** einer Implementierung
- ▶ durch Abschätzung der Anzahl der benötigten **Elementaroperationen** (Additionen/Multiplikationen/Vergleiche/Wertzuweisungen)
- ▶ in Abhängigkeit von der **Anzahl n** der gespeicherten Elemente
- ▶ verwende dazu die „Groß-O-Notation“ (\leadsto **Landau-Symbole**), um das **Wachstumsverhalten** des Zeit- bzw Platzbedarfs zu kategorisieren:
 - $\mathcal{O}(1)$ **konstant**: Kosten sind unabhängig von n
 - $\mathcal{O}(\log n)$ **logarithmisch**: Kosten wachsen um Konst., wenn sich n verdoppelt
 - $\mathcal{O}(n)$ **linear**: Kosten wachsen um Konst., wenn n um 1 wächst
 - $\mathcal{O}(n^2)$ **quadratisch**: Kosten vervierfachen sich, wenn sich n verdoppelt
 - $\mathcal{O}(2^n)$ **exponentiell**: Kosten verdoppeln sich, wenn n um 1 wächst

Inhalt

- 6 Abstrakte Datentypen: Menge und Stapel
 - Begriffsbildung
 - ADT Menge
 - ADT Stapel

Beispiel: ADT Menge

- ▶ Die **Syntax** des abstrakten Datentyps **Menge** wird durch das Interface bzw durch das UML-Diagramm definiert.
- ▶ Die **Semantik** haben wir „informell“ definiert:
 - `size()` liefert die Anzahl der Elemente.
 - `isEmpty()` prüft, ob die Menge leer ist.
 - `get()` liefert (irgend)ein Element der Menge (falls diese nicht leer ist), ohne es zu entfernen.
 - `insert(T e)` fügt das Element `e` der Menge hinzu; falls `e` bereits (vorher) Element der Menge war, wird die Menge dadurch nicht verändert.
 - `delete(T e)` entfernt das Element `e` aus der Menge. Falls `e` nicht enthalten ist, hat die Methode keine Wirkung.
 - `contains(T e)` prüft, ob `e` in der Menge enthalten ist.
- ▶ Erst **beides zusammen** definiert den ADT!
- ▶ Wir haben schon Implementierungen durch die Datenstrukturen `MengeDynArray` bzw `MengeEVL` kennengelernt.

Menge als Array: Analyse

Zeitbedarf:

- ▶ `size()`, `isEmpty()`, `get()` verursachen nur konstante Kosten
- ▶ `contains(T e)` verursacht Kosten von $\mathcal{O}(n)$
(sowohl im worst case als auch im average case)
- ▶ `insert(T e)`
 - Ausschluss von Duplikaten: $\mathcal{O}(n)$
 - Einfügen im best case (kein `increase()` nötig): $\mathcal{O}(1)$
 - Einfügen im worst case (`increase()` notwendig):
 \leadsto Analyse der Durchschnittskosten bei n -fachem Aufruf von `add(i)` in
einem dynamischen Array (ohne Prüfung auf Duplikate) $\leadsto \mathcal{O}(1)$
- ▶ `delete(T e)`: (durchschnittlich) Kosten von $\mathcal{O}(n)$
(Finden des Elements und Entfernen jeweils $\mathcal{O}(n)$)

Platzbedarf:

- ▶ im worst case: Array zu $\frac{3}{4}$ leer, also $\mathcal{O}(n)$

Menge als EVL: Analyse

Zeitbedarf:

- ▶ `size()`, `isEmpty()`, `get()`: konstante Kosten
- ▶ `contains(T e)`: Kosten von $\mathcal{O}(n)$
- ▶ `insert(T e)`: $\mathcal{O}(n)$
(Ausschluss von Duplikaten: $\mathcal{O}(n)$, Einfügen: $\mathcal{O}(1)$)
- ▶ `delete(T e)`: $\mathcal{O}(n)$
(Finden des Elements $\mathcal{O}(n)$, Entfernen $\mathcal{O}(1)$)

Platzbedarf:

- ▶ $\mathcal{O}(n)$

Bemerkung (Ausblick auf später)

Die „teuren“ Operation sind die Methoden `contains(T e)` bzw `find(T e)`. Die **Suche** könnte erheblich beschleunigt werden, wenn die Elemente der Menge in **sortierter** Reihenfolge abgelegt werden:

dann wäre **binäre Suche** möglich, die nur Kosten von $\mathcal{O}(\log n)$ verursacht.

- ▶ Voraussetzung: auf dem Basis-Datentyp lässt sich eine (totale) **Ordnungsrelation** definieren! (\leadsto „**Komparatoren**“: später)
- ▶ Dadurch können auch die Kosten für `insert(T e)` und `delete(T e)` entsprechend reduziert werden.
- ▶ Bei Verwendung von Arrays wäre dadurch ein Vorteil zu erreichen.
- ▶ Bei Verwendung von EVL bietet Sortierung keinen Vorteil.
- ▶ Andere verkettete Strukturen ermöglichen aber die binäre Suche (\leadsto „**Suchbäume**“: später)

Inhalt

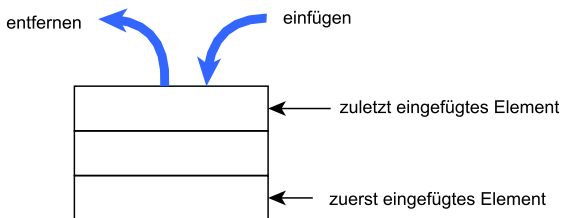
- 6 Abstrakte Datentypen: Menge und Stapel
 - Begriffsbildung
 - ADT Menge
 - ADT Stapel

LIFO

Eine häufige Anforderung an eine Sammlung von Datensätzen ist:

- ▶ Es sollen Datensätze der Sammlung hinzugefügt und auch wieder entfernt werden können.
- ▶ Die **Reihenfolge** der Einfügungen ist für das Entfernen von Bedeutung:
- ▶ Es kann immer nur das **zuletzt eingefügte** Element wieder entfernt werden.

Diese Strategie ist unter dem Namen „**LIFO**“ (Abk. für **last in - first out**) bekannt.



ADT Stapel

Wir definieren einen abstrakten Datentyp **Stapel** durch die Operationen

- ▶ **T top()**: liefert das „oberste“ Element des Stapels
- ▶ **void push(T)**: legt ein neues Element auf den Stapel
- ▶ **void pop()**: entfernt das oberste Element

Unterschiede zum ADT Menge:

- ▶ **Duplikate** (mehrfach vorkommende Elemente) **erlaubt**
- ▶ Löschen: nicht Elemente mit bestimmten **Wert**, sondern Elemente an **bestimmter Position**.

Andere gebräuchliche Bezeichnungen für einen Stapel sind **Keller** oder **Stack**.

Implementierung mittels Arrays

Es bietet sich an, das Dynamische Array um eine Operation `removeLast()` zu erweitern, die das **letzte** Element des Arrays löscht. Dann gilt

- ▶ `top()` entspricht `get(size - 1)`
- ▶ `push(e)` entspricht `add(e)`
- ▶ `pop()` entspricht `removeLast()`

Vorteile

- ▶ **kein** Durchsuchen des gesamten Arrays nötig
- ▶ das zeitaufwändige „nach-vorne-rücken“ von Elementen hinter einem gelöschten Element entfällt

Nachteile

- ▶ bei starker „Fluktuation“ der Stack-Größe ist u.U. häufiges `increase()` oder `decrease()` nötig

Stack als Array: Analyse

- ▶ `top()` kostet konstante Zeit $\mathcal{O}(1)$
- ▶ `push(e)` und `pop()`
 - konstante Zeit, wenn **kein** `increase()` bzw `decrease()` nötig ist
 - ansonsten $\mathcal{O}(n)$ Zeit (für `increase()` bzw `decrease()`)
 - amortisiert: $\mathcal{O}(1)$
- ▶ Speicherbedarf: $\mathcal{O}(n)$

Implementierung mittels Verketteter Listen

Bei Verwendung einer EVL als implementierende Datenstruktur gilt:

- ▶ `top()` entspricht `get()`
- ▶ `push(e)` entspricht `insert(e)`
- ▶ `pop()` entspricht `remove()`
- ▶ ein last-Zeiger ist nicht nötig

Stack als EVL: Analyse

Alle drei Operationen bestehen nur aus dem „Umhängen“ einiger Zeiger (Referenzen).

- ▶ Zeitbedarf für jede der drei Operationen: $\mathcal{O}(1)$
- ▶ Speicherbedarf: $\mathcal{O}(n)$