SS 2021

Algorithmen und Datenstrukturen

2. Kapitel

Fundamentale Datentypen und Datenstrukturen

Martin Dietzfelbinger

Mai 2021

Lesehinweise und Kommentare

- Willkommen in Kapitel 2!
- Thema sind grundlegende Datentypen und Datenstrukturen.
- Inhaltlich eine Menge Wiederholung. Strukturell, methodisch: vielleicht einige Neuigkeiten.
- Legen Sie sich das entsprechende Material aus „Algorithmen und Programmierung“ zurecht, besonders Kap. 10. Verschaffen Sie sich Zugang zum Buch von Saake und Sattler (s. Kap. 0).
- Das erste Ziel des Kapitels ist eine Methode zur präzisen **Spezifikation** des Verhaltens von Datenstrukturen, mitsamt Verwendung von solchen Spezifikationen in **Korrektheitsbeweisen**.
- Unsere Methode heißt „Signatur + Mathematisches Modell“. Sie weicht von der in AuP angegebenen axiomatischen Methode ab. Sie sollen lernen, die Methode selbst zu benutzen.
- Wesentliche Datentypen: Stack und dynamisches Array, Queue, veränderliche Menge, Wörterbuch (das ist dasselbe wie ein „assoziatives Array“).
- Zweites Ziel: Überblick über die einfachen Datenstrukturen, d. h. die Standardimplementierungen für diese Datentypen und die relevanten Rechenzeiten. (Nur wenige dieser Implementierungen benötigen wirklich eine ernsthafte Rechnung, um die Rechenzeit zu ermitteln. Beispiele: Stacks als Arrays mit Verdoppelung, Binäre Suche bei Mengen und Wörterbüchern, als Arrays implementiert.)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen
- Wörterbücher oder Assoziative Arrays

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen
- Wörterbücher oder Assoziative Arrays

Basis-Datenstrukturen (als bekannt vorausgesetzt)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen
- Wörterbücher oder Assoziative Arrays

Basis-Datenstrukturen (als bekannt vorausgesetzt)

- Statische Arrays (Kap. 2.3.5 im Buch von Saake und Sattler)

Thema: Datentypen und Datenstrukturen

Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen
- Wörterbücher oder Assoziative Arrays

Basis-Datenstrukturen (als bekannt vorausgesetzt)

- Statische Arrays (Kap. 2.3.5 im Buch von Saake und Sattler)
- Einfach und doppelt verkettete Listen (Kap. 13.2 und 13.3 im Buch von Saake und Sattler)

Lesehinweise und Kommentare

- Und was ist der Unterschied zwischen „Datentyp“ und „Datenstruktur“?
- In der Literatur fliegen die Begriffe durcheinander.
- Wir unterscheiden, (fast) konsequent!
- **Datentypen** werden **spezifiziert**, sie geben (mathematisch exakt) ein Wunschverhalten einer Struktur an.
- Analog zu Datentypen bei Algorithmen (Kapitel 1): Berechnungsprobleme.
- Methoden zur Spezifikation von Datentypen: ADTs in der AuP-Vorlesung, mathematische Modelle in dieser Vorlesung. (Andere sind denkbar.)
- **Datenstrukturen** sind **Implementierungen** von Datentypen.
- Analog zu Datenstrukturen bei Algorithmen (Kapitel 1): Algorithmen, die ein Berechnungsproblem lösen.
- Mehrere unterschiedliche Datenstrukturen können ein und denselben Datentyp realisieren.
- (Mehrere unterschiedliche Algorithmen können ein Berechnungsproblem lösen!)

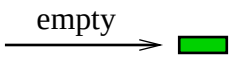
Lesehinweise und Kommentare

- Wir beginnen mit der Diskussion des Datentyps „Stack“.
- Synonym: Keller, Stapel, Pushdown-Speicher, LIFO-Speicher, LIFO-Liste.
- Zunächst: Illustration der Aktionen der Operationen auf einem Stack, hier als „Stapel“.
- Erläuterung folgt.
- Bitte beachten: In vielen Implementierungen löscht „pop“ nicht nur den obersten Eintrag, sondern liefert ihn auch als Resultat. Unser Datentyp müsste dazu erst „top“, dann „pop“ ausführen, was ein harmloses Problem ist.
- „pop“ auf der leeren Datenstruktur ist verboten; der Benutzer ist verpflichtet, dies zu verhindern – normalerweise mittels des Tests „isempty“.

2.1 Stacks* und dynamische Arrays

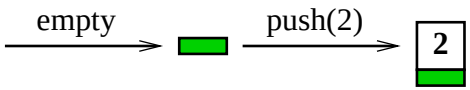
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



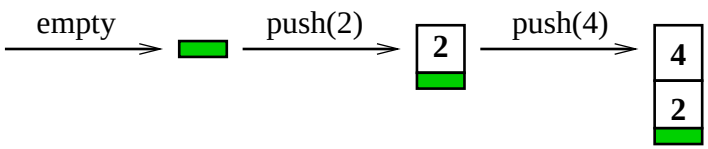
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



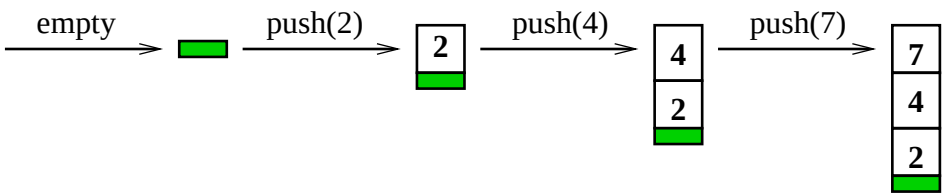
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



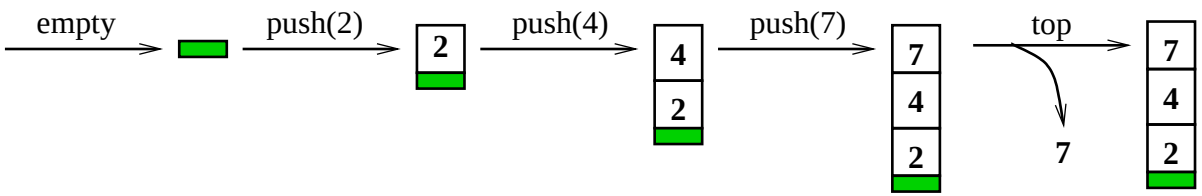
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



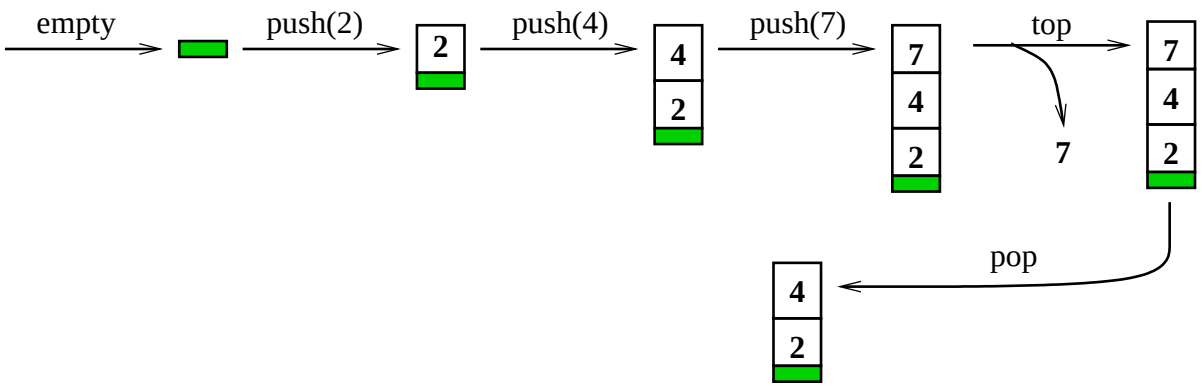
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



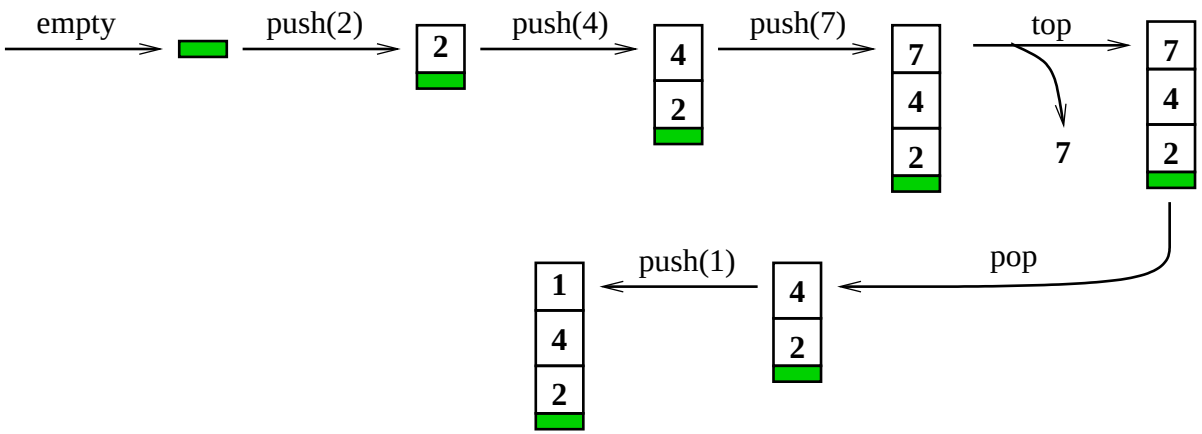
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



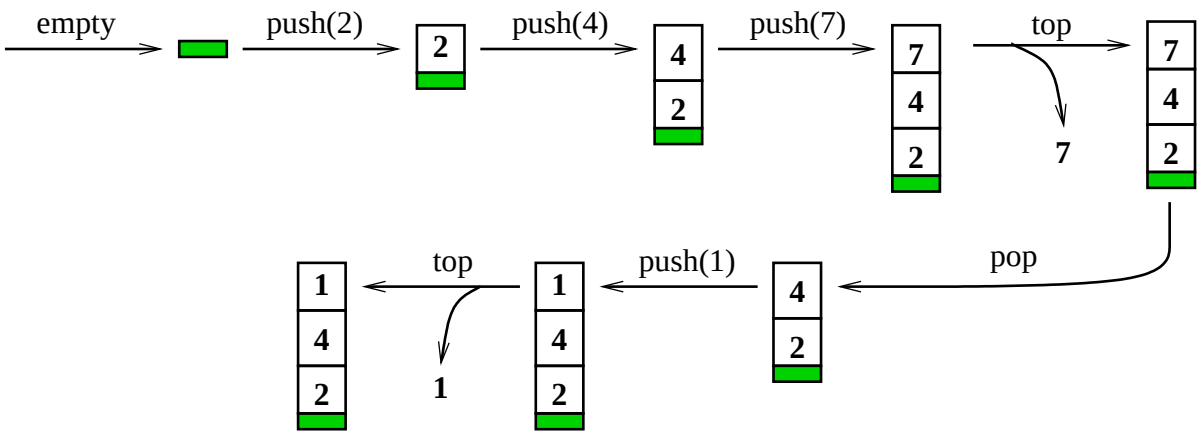
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



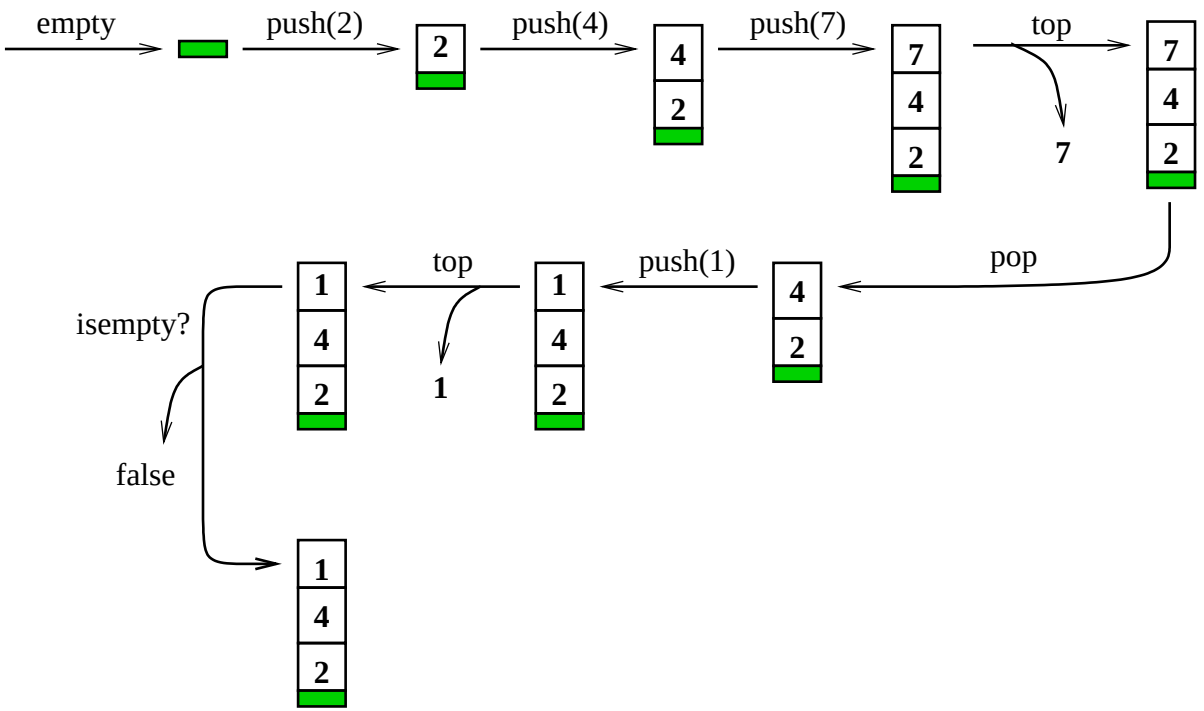
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



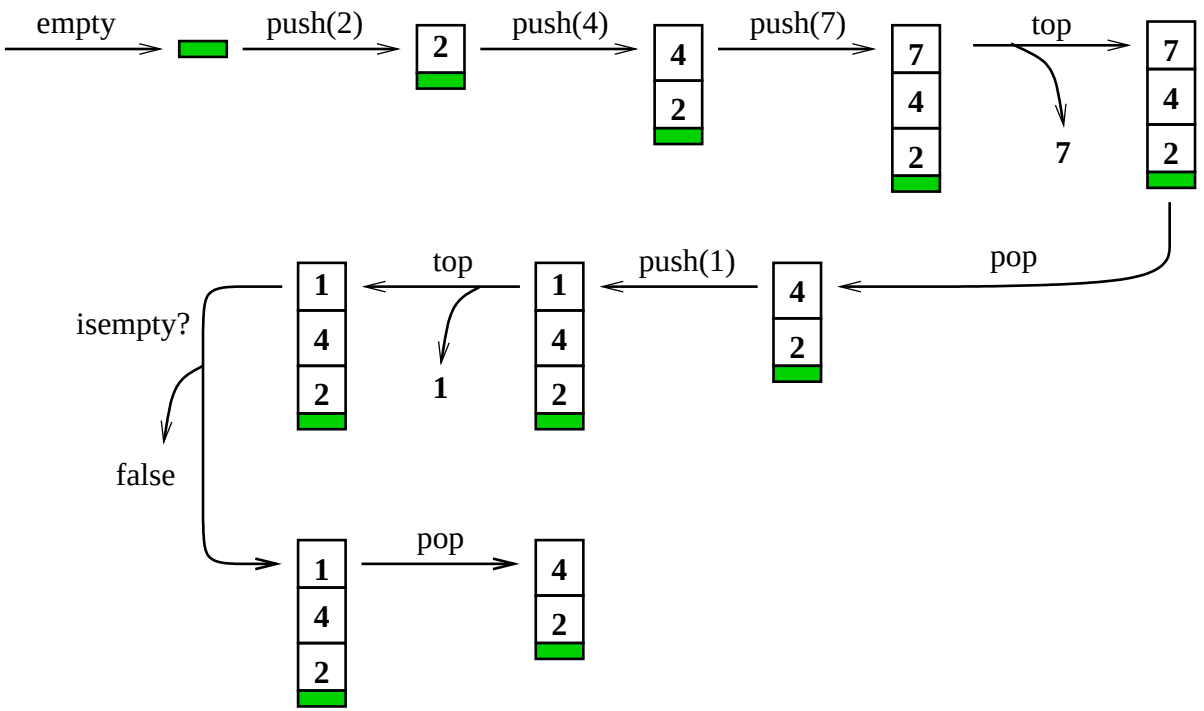
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



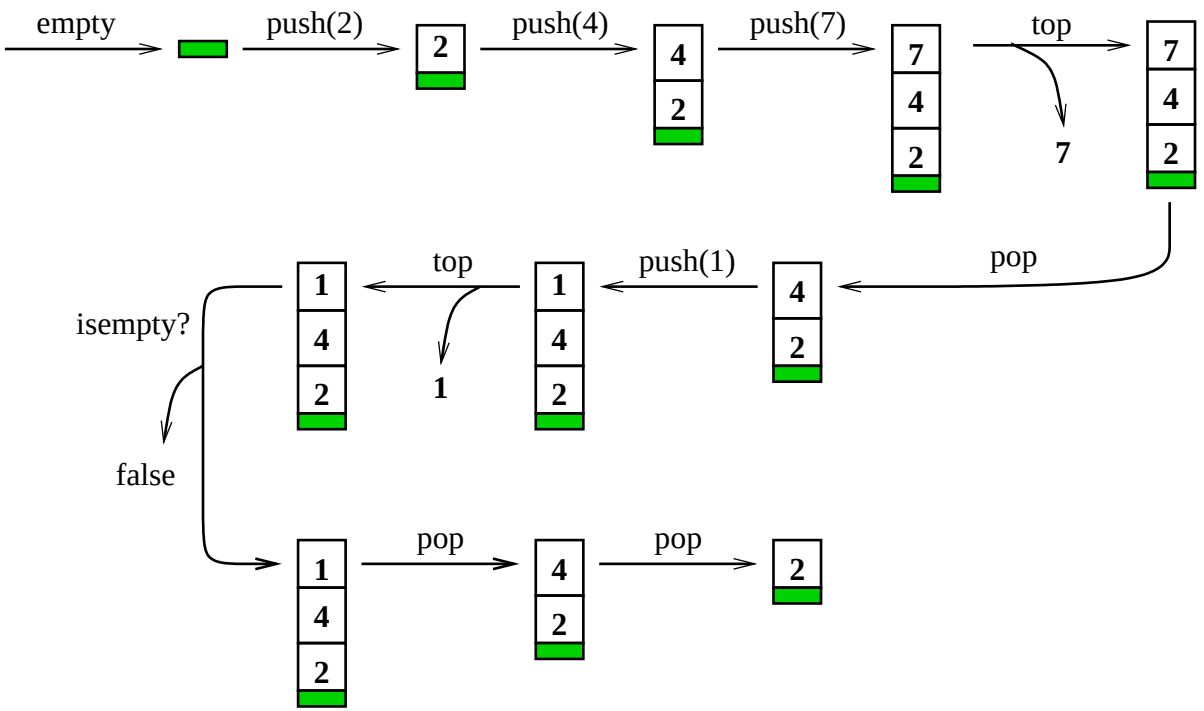
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



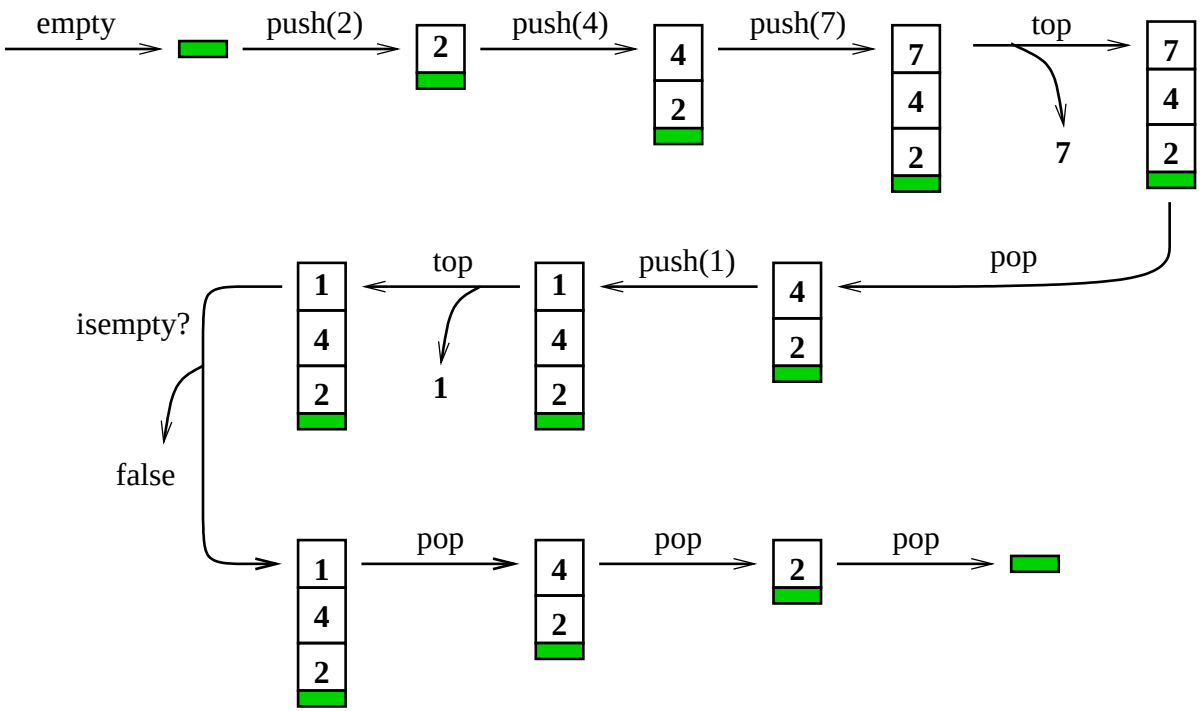
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



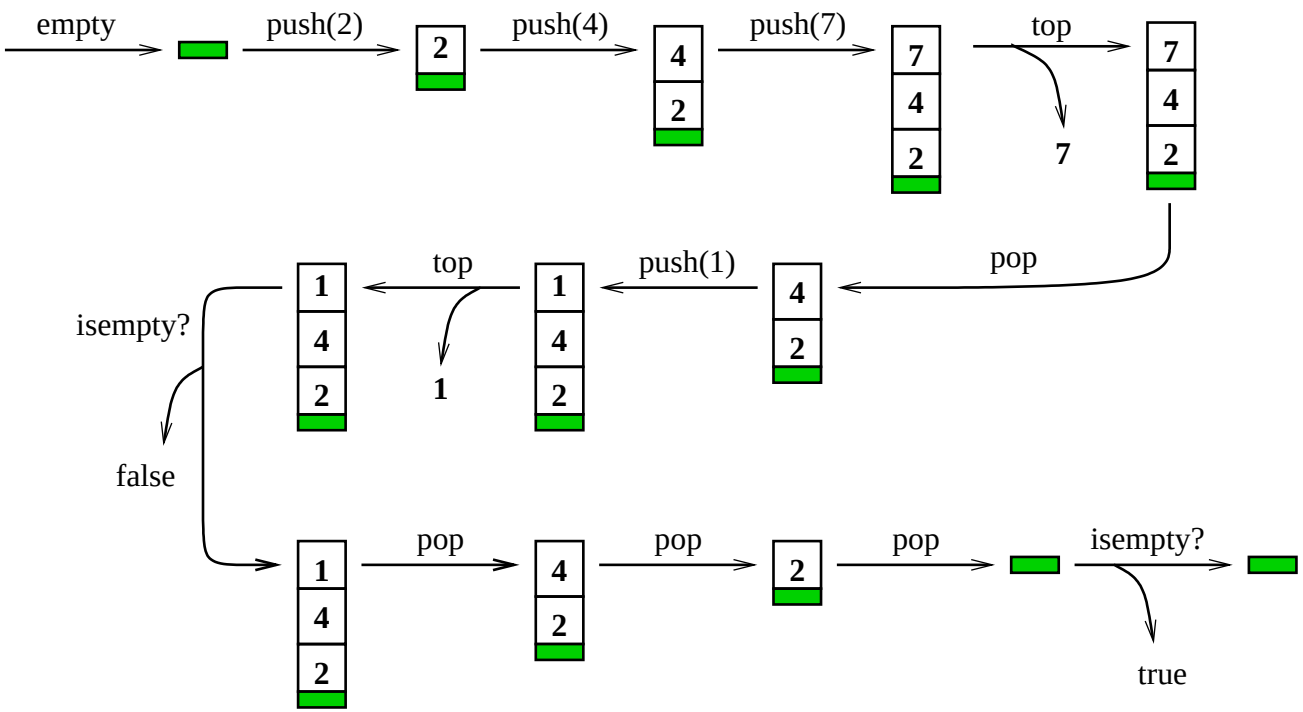
* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

2.1 Stacks* und dynamische Arrays



* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty(): erzeuge leeren Stapel („Konstruktor“).

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.
pop(s):	entferne oberstes Element von Stapel s (falls s leer: Fehler).

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.
pop(s):	entferne oberstes Element von Stapel s (falls s leer: Fehler).
top(s):	gib oberstes Element des Stapels s aus (falls s leer: Fehler).

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.
pop(s):	entferne oberstes Element von Stapel s (falls s leer: Fehler).
top(s):	gib oberstes Element des Stapels s aus (falls s leer: Fehler).
isempty(s):	Ausgabe „$true$“ falls Stapel s leer, „$false$“ sonst.

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.
pop(s):	entferne oberstes Element von Stapel s (falls s leer: Fehler).
top(s):	gib oberstes Element des Stapels s aus (falls s leer: Fehler).
isempty(s):	Ausgabe „$true$“ falls Stapel s leer, „$false$“ sonst.

Eine Beschreibung in Umgangssprache und durch ein Beispiel genügt aber nicht. Um die Frage nach der **Korrektheit einer Implementierung** überhaupt sinnvoll stellen zu können, benötigen wir eine präzise Beschreibung des Verhaltens, also eine **Spezifikation**.

Lesehinweise und Kommentare

- Unser Spezifikationsmechanismus besteht aus den Teilen „Signatur“ und „Mathematisches Modell“.
- **Signaturen** haben Sie in der AuP-Vorlesung gesehen (Kapitel 8 über Abstrakte Datentypen). Wir wiederholen dies.
- Signaturen sind eine rein syntaktische Angelegenheit.
- Eine Signatur hat zwei Teile: Erst „Sorten“ und dann „Operationen“.
- Unter „Sorten“ sind nur einige Namen aufgelistet.
  (Dass diese Namen später Mengen von Objekten benennen, ist hier noch nicht sichtbar.)
- Unter „Operationen“ findet man Zeilen mit dem Format

  $$\text{Op-Name} : \text{S-Name}_1 \times \cdots \times \text{S-Name}_k \to \text{S-Name}_{k+1}$$

  wobei „Op-name“ ein neuer Name (für eine „Operation“) ist und „S-Name$_i$“ irgendeiner der Sortennamen ist, für $i = 1, \ldots, k+1$. Diese Sortennamen dürfen sich beliebig wiederholen.
- Die Zeichen $:$ und $\to$ müssen in jeder Zeile vorkommen. $\times$ kommt vor, wenn $k \geq 2$ ist.
  Es darf auch $k = 0$ sein, dann steht „nichts“ links vom Doppelpunkt. (Beispiel: empty.)

Spezifikation des Datentyps „Stack über D“

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten: *Elements*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten: *Elements*
Stacks
Boolean

Operationen:

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*
	pop: *Stacks* $\rightarrow$ *Stacks*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*
	pop: *Stacks* $\rightarrow$ *Stacks*
	top: *Stacks* $\rightarrow$ *Elements*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements* *Stacks* *Boolean*
Operationen:	*empty*: $\rightarrow$ *Stacks* *push*: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks* *pop*: *Stacks* $\rightarrow$ *Stacks* *top*: *Stacks* $\rightarrow$ *Elements* *isempty*: *Stacks* $\rightarrow$ *Boolean*

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*
	pop: *Stacks* $\rightarrow$ *Stacks*
	top: *Stacks* $\rightarrow$ *Elements*
	isempty: *Stacks* $\rightarrow$ *Boolean*

Rein syntaktische Vorschriften!

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*
	pop: *Stacks* $\rightarrow$ *Stacks*
	top: *Stacks* $\rightarrow$ *Elements*
	isempty: *Stacks* $\rightarrow$ *Boolean*

Rein syntaktische Vorschriften! Verhalten (noch) ungeklärt!

Lesehinweise und Kommentare

- Wichtig: Die Signatur sagt überhaupt nichts über die Funktionalität des zu spezifizierenden Datentyps aus. Sie listet nur Namen von Sorten, (noch) ohne Bedeutung, . . .
- . . . und Stelligkeiten der Operationen, die zum Datentyp gehören sollen, und wie die Sorten heißen, aus denen Argumente und Ergebnisse der Operationen kommen.
- In C++ entspricht die Signatur der `.h`-Datei (ohne Kommentare).
- Das Java-Konstrukt „Interface“ ist ähnlich einer Signatur.
- Um die Spezifikation mit Inhalt zu füllen, muss erklärt werden, was die Namen „bedeuten“ sollen, d. h. die *Semantik* muss beschrieben werden.

Lesehinweise und Kommentare

- Unser Ansatz: Ein „mathematisches Modell“.
- Grob gesprochen: Das mathematische Modell beschreibt ein eindeutig bestimmtes Verhalten der Operationen. Dieses Verhalten sollen korrekte Implementierungen dann nachbauen.
- (Eher für Spezialisten, nicht prüfungsrelevant: In der Terminologie der AuP-Vorlesung (Kap. 8) gesprochen geben wir eine konkrete, mathematisch formulierte **Algebra** für unsere Signatur an. Implementierungen werden dann als korrekt angesehen, wenn sie dasselbe Ein-/Ausgabeverhalten haben wie die konkrete Algebra. Das Ganze ist in gewissem Sinn eine Anwendung der Technik der „operationellen Semantik“.)

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

mit $\mathsf{Seq}(D) = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\}$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

mit $\mathsf{Seq}(D) = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\}$

Boolean: $\{true, false\}$ bzw. $\{1, 0\}$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

mit $\mathsf{Seq}(D) = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\}$

Boolean: $\{true, false\}$ bzw. $\{1, 0\}$

Ein Stack wird also als Folge $(\boldsymbol{a_1}, a_2, \ldots, a_n)$, mit $n \geq 0$, dargestellt.

Das **linke Ende** der Folge, Element $\boldsymbol{a_1}$, steht im Stack **oben**.

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

mit $\mathsf{Seq}(D) = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\}$

Boolean: $\{true, false\}$ bzw. $\{1, 0\}$

Ein Stack wird also als Folge $(\boldsymbol{a_1}, a_2, \ldots, a_n)$, mit $n \geq 0$, dargestellt.

Das **linke Ende** der Folge, Element $\boldsymbol{a_1}$, steht im Stack **oben**.

Die leere Folge ( ) stellt den leeren Stack dar.

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen:

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: $\mathit{empty}() := ()$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

push($(a_1, \ldots, a_n), x$) :=

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

push$((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*$() := ()$

push$((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$

pop$((a_1, \ldots, a_n)) :=$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

push$((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$

$$\textit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n) \\ \end{cases}$$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

$$\mathit{push}((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$$

$$\mathit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

push$((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$

$$\textit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

top$((a_1, \ldots, a_n)) :=$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

$$\mathit{push}((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$$

$$\mathit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{top}((a_1, \ldots, a_n)) := \begin{cases} a_1 \end{cases}$$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

$$\mathit{push}((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$$

$$\mathit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{top}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

$$\mathit{push}((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$$

$$\mathit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{top}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{isempty}((a_1, \ldots, a_n)) := \begin{cases} \mathit{false}, & \text{falls } n \geq 1 \\ \mathit{true}, & \text{falls } n = 0 \end{cases}$$

Lesehinweise und Kommentare

- Aus Op-Name: S-Name$_1 \times \cdots \times$ S-Name$_k \to$ S-Name$_{k+1}$ wird durch Ersetzen der Sortennamen durch die Mengen, die den Sorten zugeordnet sind, die konkrete Signatur einer Funktion.
  Beispiel: *push*: Seq$(D) \times D \to$ Seq(D).
- Was die Funktion, die zu einem Operationsnamen gehört, tatsächlich tut, muss man (mit mathematischen Formulierungen, die eine eindeutige Beschreibung liefern) hinschreiben.
- Man achte auf *empty*. Diese Funktion bekommt als Input eine leere Liste von Argumenten (angedeutet durch „ *empty*()“). Als Ergebnis liefert sie das Tupel $() \in$ Seq(D) der Länge 0.
- Die Stapelfunktionalität ist durch die Beschreibung von *push*, *top* und *pop* gegeben. *push* fügt die neue Komponente x *vorne* an das Argumenttupel an (also „oben“ auf den Stapel), *top* liest die erste Komponente aus, *pop* entfernt die erste Komponente (also jeweils die „oberste“, zuletzt hinzugefügte).
- *pop* und *top* haben die Besonderheit, dass die Operation auf der Eingabe $()$, dem leeren Stack mit Länge $n = 0$, undefiniert ist – also im mathematischen Modell zu einem „Fehler“ führt.
- Wie mit Fehlern im mathematischen Modell dann in Implementierungen umzugehen ist (exceptions, Fehlercodes usw.), muss außerhalb unseres Formalismus beschrieben werden.

Lesehinweise und Kommentare

- Puristen haben sicher ihre Freude an der Schreibweise $pop((a_1, \ldots, a_n))$ mit zwei Klammernpaaren. Das äußere Klammernpaar ( ) schließt die Liste der Argumente von *pop* ein. Es gibt genau ein Argument, nämlich das Tupel $(a_1, \ldots, a_n)$, zu dem das innere Klammernpaar gehört.

Lesehinweise und Kommentare

- In der Vorlesung AuP, Kap. 8, wurden Abstrakte Datentypen (ADTen) mit einem anderen Mechanismus als dem hier verwendeten spezifiziert.
- Der Signaturteil ist identisch.
- Die Semantik des ADTs wurde mit „Axiomen“ beschrieben, die einen Zusammenhang zwischen dem Verhalten verschiedener Operationen herstellen.
- Beispiel: Das Axiom $\mathit{pop}(\mathit{push}(s, x)) = s$ besagt intuitiv: Wenn man x oben auf einen Stapel s legt und dann das oberste Element entfernt, erhält man wieder s.
- Wir geben die vollständige Liste der Axiome an, aber ohne Einbeziehung von Fehlern.
  (Man müsste eigentlich hinzufügen, dass $\mathit{top}(\mathit{empty}())$ und $\mathit{pop}(\mathit{empty}())$ nicht definiert sind.)

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative:

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, s$: *Stacks*, $\forall\, x$: *Elements*

$$pop(push(s, x)) = s$$
$$top(push(s, x)) = x$$
$$isempty(empty()) = true$$
$$isempty(push(s, x)) = false$$

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, s$: *Stacks*, $\forall\, x$: *Elements*

$$pop(push(s, x)) = s$$
$$top(push(s, x)) = x$$
$$isempty(empty()) = true$$
$$isempty(push(s, x)) = false$$

Vorteil: Zugänglich für automatisches Beweisen von Eigenschaften des Datentyps.

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, s$: *Stacks*, $\forall\, x$: *Elements*

$$\mathit{pop}(\mathit{push}(s, x)) = s$$
$$\mathit{top}(\mathit{push}(s, x)) = x$$
$$\mathit{isempty}(\mathit{empty}()) = \mathit{true}$$
$$\mathit{isempty}(\mathit{push}(s, x)) = \mathit{false}$$

Vorteil: Zugänglich für automatisches Beweisen von Eigenschaften des Datentyps.

Nachteil: Finden eines geeigneten Axiomensystems für eine geplante Semantik evtl. nicht offensichtlich.

Lesehinweise und Kommentare

- Die Methode, ADTs mit Axiomen zu spezifizieren, ist sehr effizient.
- Sie erleichtert Korrektheitsbeweise für Implementierungen, da man nur nachprüfen muss, dass die Implementierungen der Operationen, also die Methoden, die Axiome erfüllen. Das ist oft einfacher als mit unserem mathematischen Modell.
- Auch automatische Korrektheitsbeweise für Implementierungen werden so ermöglicht.
- In AuP wurde angedeutet, wie man auf dem Weg über Terme mit „Quotientenbildung“ zu einer besonderen mathematischen Struktur („initiale Algebra“) kommt, die die Semantik der Spezifikation darstellt. Interessierte sollten das nochmal nachlesen; für unsere Vorlesung ist es nicht relevant.
- Eine ähnliche Konstruktion werden Sie in der Vorlesung „Logik und Logikprogrammierung“ sehen.
- Bei Verwendung der Axiomen-Methode ist die zentrale Schwierigkeit, einen geeigneten vollständigen Satz von Axiomen zu finden.

Lesehinweise und Kommentare

- Wir haben einen Datentyp *spezifiziert*. Nun wollen wir ihn durch Algorithmen und am Ende durch ein Programm (z. B. in Java) realisieren. Diese algorithmische Realisierung erfolgt durch eine „Datenstruktur“. Die Signatur gibt dabei den Plan vor.
- Aus Sorten werden Klassen. Die Elemente einer Sorte werden also durch Objekte der entsprechenden Klasse realisiert.
- Aus Operationen werden Methoden einer passenden Klasse. Die Operationen der Signatur sind alle öffentlich („public“), also durch den Benutzer der Datenstruktur aufrufbar.
- Beispiel Stacks: Die Klasse die Sorte „Elemente“ ist frei wählbar, über Templates in C++. In Java benutzt man oft die allgemeinste Klasse `object` oder „generische Programmierung“.
- Die Klasse „Boolean“ mit Konstanten $true$ und $false$ ist in Programmiersprachen eingebaut.
- Wir bauen eine Klasse für „Stacks“. Diese muss (öffentliche) Methoden für *empty* (den Konstruktor, der ein neues Stackobjekt liefert), für *push*, *top* und *pop* anbieten.
- Üblich: Ein Stack-Objekt wird der Methode nicht als Argument übergeben, die Methode wird für dieses Objekt aufgerufen. Daher fehlt das Stack-Objekt in der Parameterliste.
- In Java-Terminologie: Wenn man die Signatur als „interface“ aufschreibt, ist die Datenstruktur eine Implementierung zu diesem „interface“.

Implementierung von Stacks

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Klasse für Elemente (ein Parameter, bereitgestellt über Generic- bzw. Template-Mechanismus): `elements`. Dazu: `boolean` aus der Programmiersprache.

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Klasse für Elemente (ein Parameter, bereitgestellt über Generic- bzw. Template-Mechanismus): `elements`. Dazu: `boolean` aus der Programmiersprache.

Prinzip: Kapselung – Information Hiding

Der Benutzer sieht die Details der Implementierung nicht. Zugriff auf die Objekte der Datenstruktur erfolgt ausschließlich über die (öffentlichen) Methoden der Klassen, entsprechend den Operationen des Datentyps.

Zwei strukturell unterschiedliche Möglichkeiten (mindestens!) für Implementierung von Stackobjekten:

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Klasse für Elemente (ein Parameter, bereitgestellt über Generic- bzw. Template-Mechanismus): `elements`. Dazu: `boolean` aus der Programmiersprache.

Prinzip: Kapselung – Information Hiding

Der Benutzer sieht die Details der Implementierung nicht. Zugriff auf die Objekte der Datenstruktur erfolgt ausschließlich über die (öffentlichen) Methoden der Klassen, entsprechend den Operationen des Datentyps.

Zwei strukturell unterschiedliche Möglichkeiten (mindestens!) für Implementierung von Stackobjekten:

(Einfach verkettete) **Lineare Liste**

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Klasse für Elemente (ein Parameter, bereitgestellt über Generic- bzw. Template-Mechanismus): `elements`. Dazu: `boolean` aus der Programmiersprache.

Prinzip: Kapselung – Information Hiding

Der Benutzer sieht die Details der Implementierung nicht. Zugriff auf die Objekte der Datenstruktur erfolgt ausschließlich über die (öffentlichen) Methoden der Klassen, entsprechend den Operationen des Datentyps.

Zwei strukturell unterschiedliche Möglichkeiten (mindestens!) für Implementierung von Stackobjekten:

(Einfach verkettete) **Lineare Liste** und **Array**.

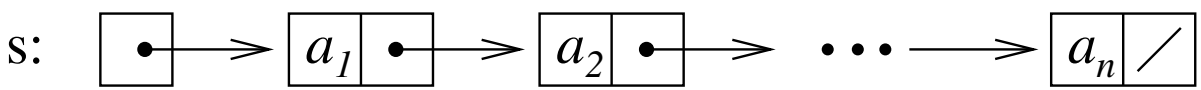
Lesehinweise und Kommentare

- Wir beschreiben zunächst die erste Implementierungsmöglichkeit über verkettete Listen.
- Information über den elementaren Datentyp „verkettete Liste“ und die Implementierung in Java findet man z. B. in der AuP-Vorlesung [AuP], Kap. 10 oder in [Saake/Sattler], Kap. 13.2.

Listenimplementierung von Stacks

Listenimplementierung von Stacks

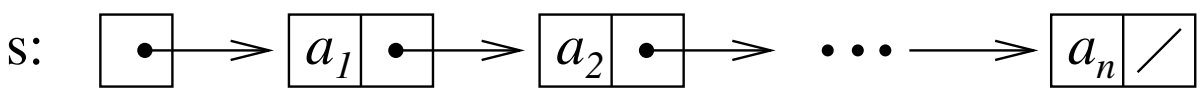
Ein Stack $(a_1, \ldots, a_n)$ („oben“ ist bei a_1. Die Einträge stammen aus der Klasse `elements`) wird durch eine einfach verkettete Liste s dargestellt. Die Einträge stehen in der Liste in Reihenfolge $a_1, \ldots, a_n$:



(Das Symbol ⁄ steht für den Nullzeiger/die Nullreferenz)

Listenimplementierung von Stacks

Ein Stack $(a_1, \ldots, a_n)$ („oben“ ist bei a_1. Die Einträge stammen aus der Klasse `elements`) wird durch eine einfach verkettete Liste s dargestellt. Die Einträge stehen in der Liste in Reihenfolge $a_1, \ldots, a_n$:

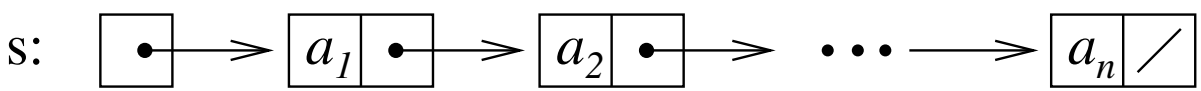


(Das Symbol $\not$ steht für den Nullzeiger/die Nullreferenz)

Zur Stack-Klasse gehören Methoden, die den Operationen entsprechen.

Listenimplementierung von Stacks

Ein Stack $(a_1, \ldots, a_n)$ („oben“ ist bei a_1. Die Einträge stammen aus der Klasse `elements`) wird durch eine einfach verkettete Liste s dargestellt. Die Einträge stehen in der Liste in Reihenfolge $a_1, \ldots, a_n$:



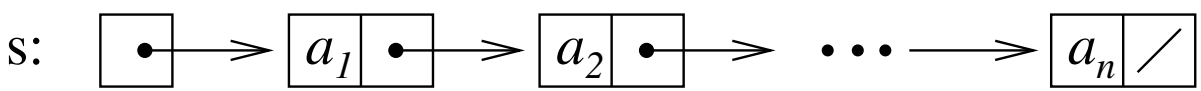
(Das Symbol $\not$ steht für den Nullzeiger/die Nullreferenz)

Zur Stack-Klasse gehören Methoden, die den Operationen entsprechen.

`s.empty`:

Listenimplementierung von Stacks

Ein Stack $(a_1, \ldots, a_n)$ („oben“ ist bei a_1. Die Einträge stammen aus der Klasse `elements`) wird durch eine einfach verkettete Liste s dargestellt. Die Einträge stehen in der Liste in Reihenfolge $a_1, \ldots, a_n$:



(Das Symbol ⁄ steht für den Nullzeiger/die Nullreferenz)

Zur Stack-Klasse gehören Methoden, die den Operationen entsprechen.

`s.empty`: Erzeugt neue leere Liste.

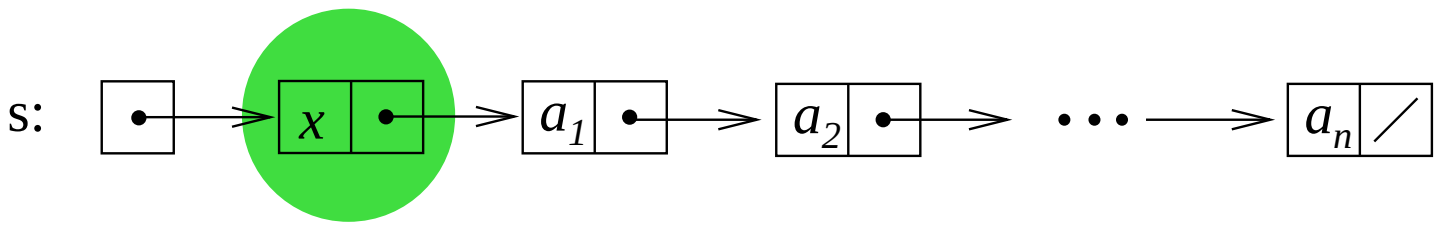
Listenimplementierung von Stacks

Listenimplementierung von Stacks

`s.push(`x`)`:

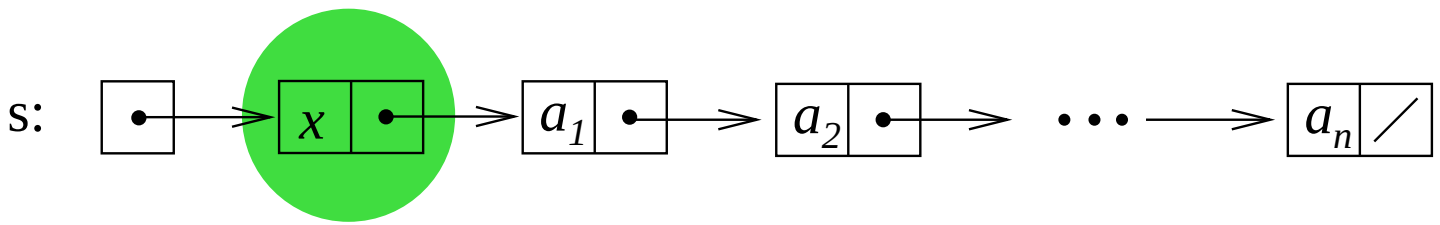
Listenimplementierung von Stacks

`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste.
Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also



Listenimplementierung von Stacks

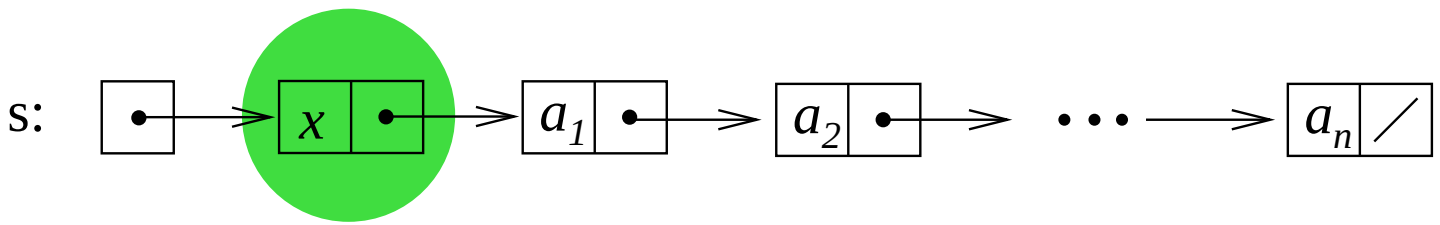
`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste.
Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also



`s.pop`:

Listenimplementierung von Stacks

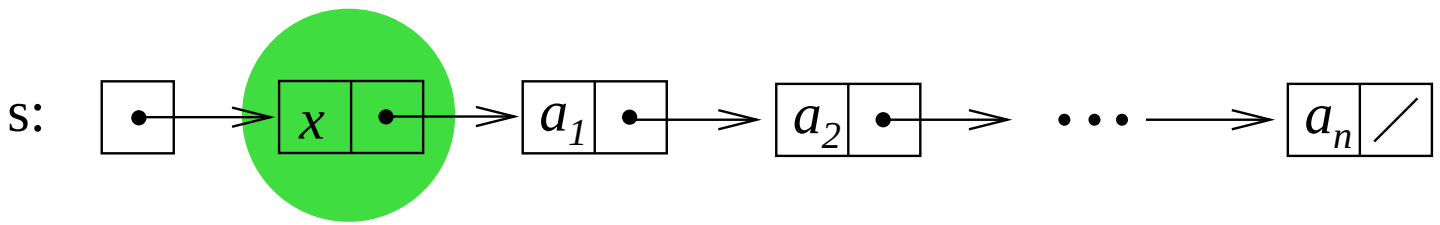
`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste. Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also



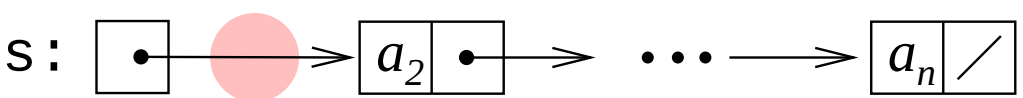
`s.pop`: Entferne ersten Listenknoten (falls vorhanden). Aus der Liste für $(a_1, \ldots, a_n)$ mit $n \geq 1$ wird also (an der Stelle des roten Kreises stand vorher a_1):

Listenimplementierung von Stacks

`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste. Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also

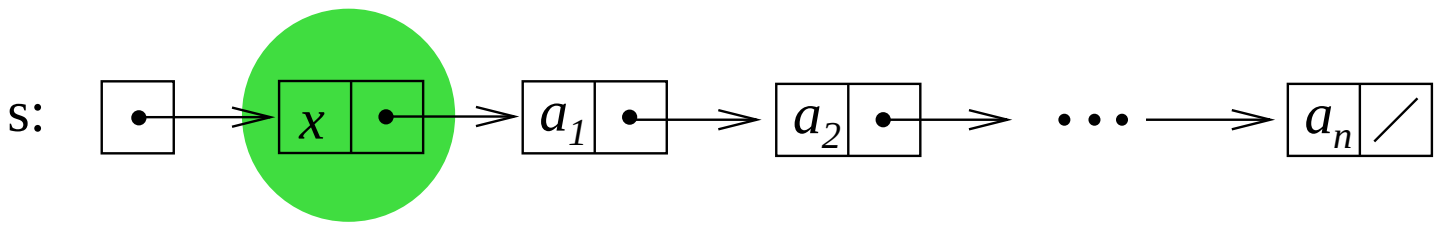


`s.pop`: Entferne ersten Listenknoten (falls vorhanden). Aus der Liste für $(a_1, \ldots, a_n)$ mit $n \geq 1$ wird also (an der Stelle des roten Kreises stand vorher a_1):

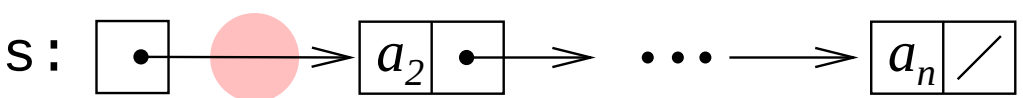


Listenimplementierung von Stacks

`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste. Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also



`s.pop`: Entferne ersten Listenknoten (falls vorhanden). Aus der Liste für $(a_1, \ldots, a_n)$ mit $n \geq 1$ wird also (an der Stelle des roten Kreises stand vorher a_1):



Wenn `s.pop` für eine leere Liste aufgerufen wird, muss eine Fehlermeldung (exception o. ä.) erfolgen.

Listenimplementierung von Stacks

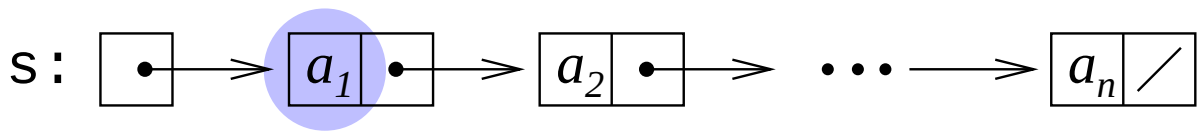
`s.top`:

Listenimplementierung von Stacks

`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)

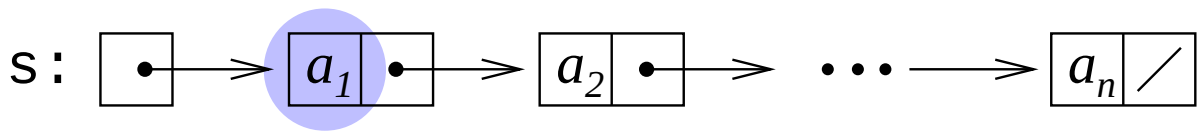
Listenimplementierung von Stacks

`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)



Listenimplementierung von Stacks

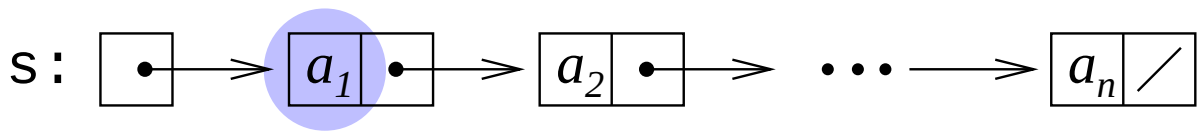
`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)



`s.isempty`:

Listenimplementierung von Stacks

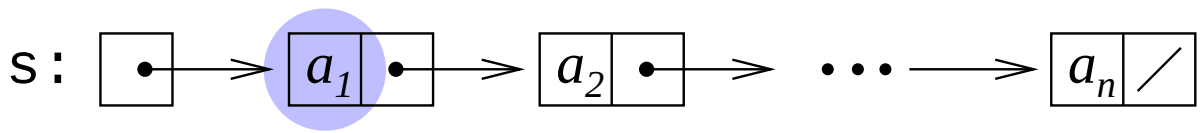
`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)



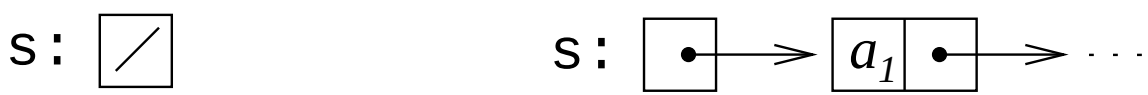
`s.isempty`: Falls Liste leer: Ausgabe $true$, sonst $false$, Ausgabetyp `boolean`.

Listenimplementierung von Stacks

`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)



`s.isempty`: Falls Liste leer: Ausgabe *true*, sonst *false*, Ausgabetyp `boolean`.



Listenimplementierung von Stacks

Listenimplementierung von Stacks

Beobachtung

Wenn der Benutzer nie den Fehler macht, *pop* oder *top* für den leeren Stack (die leere Liste) aufzurufen, dann ist der einzige Fehler, der bei der Benutzung dieser Implementierung auftreten kann, ein Speicherüberlauf bei der Operation `s.push(`x`)`, weil für das neue Listenelement im für das Programm verfügbaren Speicher kein Platz mehr ist.

Listenimplementierung von Stacks

Beobachtung

Wenn der Benutzer nie den Fehler macht, *pop* oder *top* für den leeren Stack (die leere Liste) aufzurufen, dann ist der einzige Fehler, der bei der Benutzung dieser Implementierung auftreten kann, ein Speicherüberlauf bei der Operation `s.push(`x`)`, weil für das neue Listenelement im für das Programm verfügbaren Speicher kein Platz mehr ist.

Beobachtung

Jede der Methoden hat Rechenzeit $O(1)$ (also konstant, unabhängig von n).

Wir bemerken aber, dass in vielen Programmiersprachen die Aktion, ein neues Listenelement zu kreieren, wie es in `s.push` benötigt wird, eine deutlich größere (konstante) Rechenzeit erfordert als etwa ein einfacher Speicherzugriff oder der Zugriff auf den ersten Eintrag einer linearen Liste. – Eine Möglichkeit, diesen Effekt abzuschwächen, wäre es, von Zeit zu Zeit ein ganzes Array von Listenelementen auf einen Schlag zu allozieren und die Verwaltung der Listenelemente direkt, als Teil der Datenstruktur, durchzuführen. Spart Rechenzeit, erhöht aber den Programmieraufwand.

PAUSE

Lesehinweise und Kommentare

- Nächstes Thema: Wann soll eine Implementierung eines Datentyps durch eine Datenstruktur, also eine Klasse mit Hilfsklassen und Methoden, *korrekt* heißen?
- Ist dann die Listenimplementierung von Stacks in diesem Sinn korrekt?

Korrektheit der Listenimplementierung

Korrektheit der Listenimplementierung

Eine beliebige Menge D sei gegeben. Wir betrachten eine Folge

$$\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N \quad (\textit{push} \text{ hat immer ein Argument aus } D)$$

von Stackoperationen

Korrektheit der Listenimplementierung

Eine beliebige Menge D sei gegeben. Wir betrachten eine Folge

$$\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N \quad (\textit{push} \text{ hat immer ein Argument aus } D)$$

von Stackoperationen, wobei *empty* nur als Op_0 vorkommt.

Korrektheit der Listenimplementierung

Eine beliebige Menge D sei gegeben. Wir betrachten eine Folge

$$\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N \quad (\textit{push} \text{ hat immer ein Argument aus } D)$$

von Stackoperationen, wobei *empty* nur als Op_0 vorkommt.
Wenn man die Operationen dieser Folge im mathematischen Modell der Reihe nach anwendet, und es keinen Fehler gibt, entsteht eine Folge

$$s_0, s_1, s_2, \ldots, s_N$$

von Stacks (Tupeln), die man als Zustände auffasst,

Korrektheit der Listenimplementierung

Eine beliebige Menge D sei gegeben. Wir betrachten eine Folge

$$\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N \quad (\textit{push} \text{ hat immer ein Argument aus } D)$$

von Stackoperationen, wobei *empty* nur als Op_0 vorkommt.
Wenn man die Operationen dieser Folge im mathematischen Modell der Reihe nach anwendet, und es keinen Fehler gibt, entsteht eine Folge

$$s_0, s_1, s_2, \ldots, s_N$$

von Stacks (Tupeln), die man als Zustände auffasst, und eine Folge

$$z_0, z_1, z_2, \ldots, z_N$$

von Ausgaben (Werte in D bzw. $\{true, false\}$).

Lesehinweise und Kommentare

- Genauer gesagt:
  - Die Menge der Zustände ist Seq(D) plus ein Fehlerzustand „↯“.
  - s_0 ist $() = \mathit{empty}()$, das leere Tupel.
  - Für $i > 0$ entsteht s_i aus s_{i-1} durch Anwendung der Operation Op_i (mit oder ohne weiterem Argument neben s_{i-1}), nach den Angaben im mathematischen Modell.
  - Wenn die Anwendung von Op_i auf s_i eine Ausgabe (in D oder $\{true, false\}$) liefert, dann ist z_i dieser Wert, sonst gibt es keinen Wert z_i.
  - Wenn und sobald ein Fehler aufgetreten ist (*top* oder *pop* wird auf den leeren Stack angewendet), schalten wir auf den Fehlerzustand „↯“, der dann auch als Ausgabe ausgegeben wird.
  - Das Anwenden einer beliebigen Operation auf den Fehlerzustand liefert wieder den Fehlerzustand.
- Betrachten Sie das folgende Beispiel!

Beispiel (mit $D = \mathbb{N}$) :

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–
12	*pop*	$()$	–

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–
12	*pop*	$()$	–
13	*isempty*	$()$	$true$

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–
12	*pop*	$()$	–
13	*isempty*	$()$	$true$
14	*pop*	$\lightning$	$\lightning$

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–
12	*pop*	$()$	–
13	*isempty*	$()$	$true$
14	*pop*	$\lightning$	$\lightning$
15	*push*(3)	$\lightning$	$\lightning$
⋮	⋮	⋮	⋮

Korrektheit der Listenimplementierung

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. $(^1)$.

Korrektheit der Listenimplementierung

$\mathsf{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathsf{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathsf{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. $(^1)$.
$\mathsf{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathsf{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. $(^1)$.

Korrektheit der Listenimplementierung

$\mathsf{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathsf{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathsf{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. $(^1)$.
$\mathsf{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathsf{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. $(^1)$.
$\mathsf{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathsf{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. $(^1)$.

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. (1).
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. (1).

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. (1).
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. (1).

(1): jeweils solange $s_{i-1} \neq \lightning$ und in Op_i kein Fehler auftritt.

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. $(^1)$.
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. $(^1)$.
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. $(^1)$.
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. $(^1)$.

$(^1)$: jeweils solange $s_{i-1} \neq \lightning$ und in Op_i kein Fehler auftritt.

Fehlerfall: Wenn $\mathrm{Op}_i = \mathit{top}$ oder $\mathrm{Op}_i = \mathit{pop}$ und $s_{i-1} = ()$

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. (1).
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. (1).

(1): jeweils solange $s_{i-1} \neq \lightning$ und in Op_i kein Fehler auftritt.

Fehlerfall: Wenn $\mathrm{Op}_i = \mathit{top}$ oder $\mathrm{Op}_i = \mathit{pop}$ und $s_{i-1} = ()$, oder wenn $s_{i-1} = \lightning$, dann ist $s_i = \lightning$ und $z_i = \lightning$.

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. (1).
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. (1).

(1): jeweils solange $s_{i-1} \neq \lightning$ und in Op_i kein Fehler auftritt.

Fehlerfall: Wenn $\mathrm{Op}_i = \mathit{top}$ oder $\mathrm{Op}_i = \mathit{pop}$ und $s_{i-1} = ()$, oder wenn $s_{i-1} = \lightning$, dann ist $s_i = \lightning$ und $z_i = \lightning$.

(Keine „Erholung“ nach dem ersten Fehler!)

Lesehinweise und Kommentare

- Wir „vergessen“ nun, was „im Innern“ des mathematischen Modells stattfindet, und konzentrieren uns nur noch auf das „Ein-/Ausgabeverhalten“.
- D. h.: Ignoriere $s_0, \ldots, s_N$, beachte nur die Operationenfolge (inklusive Argumente bei *push*) und die Folge $z_0, \ldots, z_N$ der Ausgaben.
- Idee: Eine Implementierung des Datentyps „Stack“ heißt dann **korrekt**, wenn sie auf jede gegebene Operationenfolge als Eingabe genau dieselbe Ausgabefolge erzeugt wie das mathematische Modell.
- Man muss noch eine Ausnahme einbauen: Die Datenstruktur kann nicht für einen Speicherüberlauf verantwortlich gemacht werden. (Aber dies darf die einzige mögliche Fehlerursache sein, wenn die Operationsfolge im mathematischen Modell keinen Fehler erzeugt.)

Beispiel für Ein-/Ausgabeverhalten

Korrektheit der Implementierung bedeutet, dass das Ein-/Ausgabeverhalten des mathematischen Modells auf beliebigen Operationsfolgen nachgebaut wird.

OP_i	z_i
empty	–
push(2)	–
push(4)	–
push(7)	–
top	7
pop	–
push(1)	–
top	1
isempty	$false$
top	1
pop	–
pop	–
pop	–
isempty	$true$
pop	$\lightning$
push(3)	$\lightning$
$\vdots$	$\vdots$

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell,

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack)

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell und in der Implementierung kein Fehler aufgetreten ist, dann sind die Einträge in der Liste (der Datenstruktur) genau die Einträge in s_i (dem Zustand des mathematischen Modells)

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell und in der Implementierung kein Fehler aufgetreten ist, dann sind die Einträge in der Liste (der Datenstruktur) genau die Einträge in s_i (dem Zustand des mathematischen Modells), und die Ausgabe der Datenstruktur in Schritt i ist z_i.

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell und in der Implementierung kein Fehler aufgetreten ist, dann sind die Einträge in der Liste (der Datenstruktur) genau die Einträge in s_i (dem Zustand des mathematischen Modells), und die Ausgabe der Datenstruktur in Schritt i ist z_i.

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i)

> Wenn in Schritten $0, \ldots, i$ im mathematischen Modell und in der Implementierung kein Fehler aufgetreten ist, dann sind die Einträge in der Liste (der Datenstruktur) genau die Einträge in s_i (dem Zustand des mathematischen Modells), und die Ausgabe der Datenstruktur in Schritt i ist z_i.

(Details des Induktionsbeweises: Druckfolien.)

Beweis von (IB$_i$) durch Induktion über i.

I.A.: $i = 0$. Wir haben Op$_0$ = *empty*. Also: $s_0 = ()$ und Op$_0$ liefert die leere Liste, und (IB$_0$) gilt.

I.V.: $i > 0$ und (IB$_{i-1}$) gilt.

I.-Schritt: Wenn $s_{i-1} = \lightning$, müssen wir nichts zeigen. Nehmen wir also $s_{i-1} = (a_1, \ldots, a_n)$ an. Nach I.V. enthält die Datenstruktur eine Liste mit Einträgen $a_1, \ldots, a_n$.

Nun betrachtet man Op$_i$. Wenn dies *top* ist, gibt das mathematische Modell $z_i = a_1$ aus, ebenso wie die Implementierung, vorausgesetzt, es ist $n \geq 1$. Wenn $n = 0$ ist, wechselt das mathematische Modell in den Fehlerzustand $\lightning$, gibt „Fehler“ aus, und auch die Implementierung meldet „Fehler“. Es folgt (IB$_i$).

Wenn Op$_i$ = *pop* ist, liefert das mathematische Modell $s_i = (a_2, \ldots, a_n)$, und die Implementierung entfernt den ersten Eintrag a_1 aus der Liste, vorausgesetzt, es gilt $n \geq 1$. Sonst wechseln beide in einen Fehlerzustand. Es folgt (IB$_i$).

Wenn Op$_i$ = *push*(x), dann liefert das mathematische Modell $s_i = (x, a_1, \ldots, a_n)$, und die Implementierung fügt x vorne in die Liste ein. Es folgt (IB$_i$).

Wenn Op$_i$ = *isempty*, liefert das mathematische Modell $z_i = true$ bzw. $z_i = false$, je nachdem ob $n \geq 1$ oder $n = 0$ gilt. Genau dieselbe Ausgabe wird von der Implementierung erzeugt. Also gilt (IB$_i$).

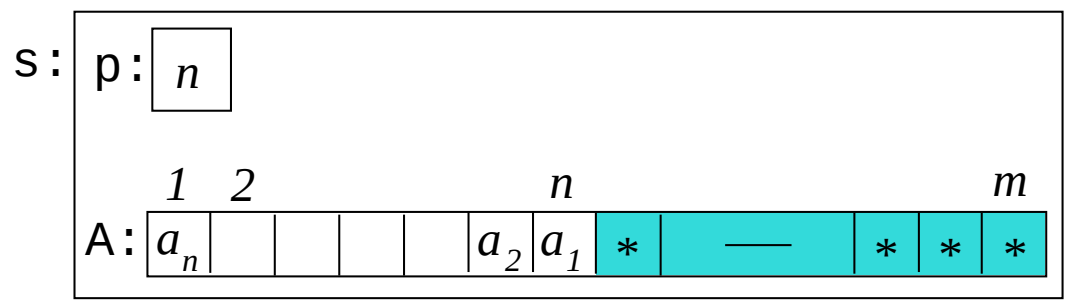
Kommentar

- Wenn man diesen Beweis betrachtet, drängt sich der Gedanke auf, dass es sich um eine gewaltige Banalität handelt. Man überprüft ja nur, was bei der Definition des mathematischen Modells und bei der Programmierung der Datenstruktur geplant war, nämlich dass die Implementierung genau das mathematische Modell nachahmt.
- Diese Situation tritt bei der Implementierung von nicht zu komplexen Datenstrukturen recht häufig ein.
- Die Korrektheitsbeweise werden dann „trivial“ oder „banal“, weil sie nur das Offensichtliche nachkontrollieren.
- Wir werden daher solche Beweise im weiteren Verlauf nicht mehr durchführen, sondern werden uns üblicherweise darauf beschränken, die (sorgfältig formulierte) Induktionsbehauptung anzugeben. Das muss man aber wirklich auch machen! Auf das mechanische, langweilige Durchspielen von Fällen werden wir verzichten.
- Beweise dieser Art muss man in der Prüfung nicht durchführen können, aber man muss gegebenenfalls erklären können, was zu tun ist (und die Induktionsbehauptung aufschreiben können).

Arrayimplementierung von Stacks

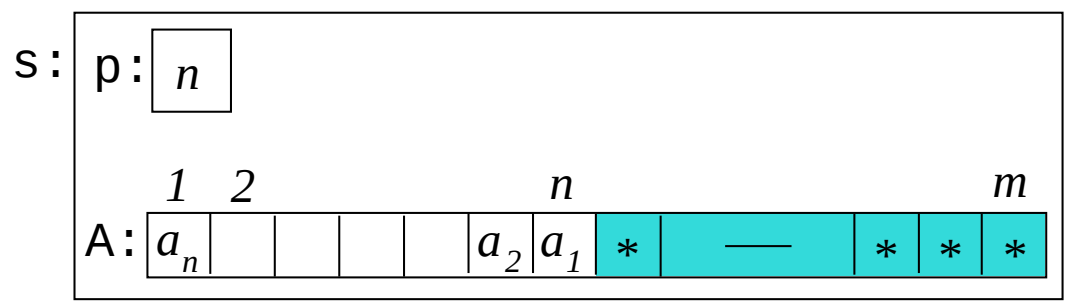
Arrayimplementierung von Stacks

(Vgl. [Saake/Sattler], Kap. 13.1, oder [AuP], Kap. 10.) Das Stackobjekt heißt `s`; es hat als Attribute einen Zeiger/eine Referenz auf ein Array `A[1..`m`]` und eine Integervariable `p` („Pegel“).



Arrayimplementierung von Stacks

(Vgl. [Saake/Sattler], Kap. 13.1, oder [AuP], Kap. 10.) Das Stackobjekt heißt `s`; es hat als Attribute einen Zeiger/eine Referenz auf ein Array `A[1..`m`]` und eine Integervariable `p` („Pegel“).



Idee: Wenn der Stack im mathematischen Modell momentan $(a_1, \ldots, a_n)$ ist, steht n in `p` und in `A[1]` steht a_n (unterster Eintrag), . . . , in `A[`n`]` steht a_1 (oberster Eintrag). **Achtung: Reihenfolge umgedreht!**

„$*$“ steht für einen beliebigen Arrayeintrag, in `A[`i`]`, für $n + 1 \leq i \leq m$.

Arrayimplementierung von Stacks: Methoden

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $=$ `A.length` **then „Fehler“** (z.B. *OverflowException*)

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $=$ `A.length` **then „Fehler“** (z.B. *OverflowException*)
else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $=$ `A.length` **then „Fehler“** (z.B. *OverflowException*)
else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
else `p` $\leftarrow$ `p-1`;

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
`p` $\leftarrow 0$;

`s.push(x)`:
if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
else `p` $\leftarrow$ `p-1`;

`s.top`:

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else return `A[p]`;

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else return `A[p]`;

`s.isempty`:

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else return `A[p]`;

`s.isempty`:
  if `p` $= 0$ **then return** „$true$“

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else return `A[p]`;

`s.isempty`:
  if `p` $= 0$ **then return** „$true$“ **else return** „$false$“;

Lesehinweise und Kommentare

- Die Details der Implementierung drängen sich geradezu auf.
- Der wichtigste Punkt ist, dass das obere Ende des Stacks an der Position n (in `p`) in `A` sitzt, so dass die Operationen konstante Zeit benötigen.
- Es gibt eine neue Fehlermöglichkeit in der Implementierung: Das Array `A`$[1..m]$ ist voll, und es soll *push*(x) ausgeführt werden.
- Es folgt: Korrektheitsbeweis für die Arrayimplementierung.

Korrektheit der Arrayimplementierung

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen (wobei *empty* nur als Op_0 vorkommen darf). Im mathematischen Modell wird davon eine Folge $s_0, s_1, s_2, \ldots, s_N$ von Stacks (Tupeln) und eine Folge $z_0, z_1, z_2, \ldots, z_N$ von Ausgaben erzeugt.

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen (wobei *empty* nur als Op_0 vorkommen darf). Im mathematischen Modell wird davon eine Folge $s_0, s_1, s_2, \ldots, s_N$ von Stacks (Tupeln) und eine Folge $z_0, z_1, z_2, \ldots, z_N$ von Ausgaben erzeugt.

Satz 2.1.2

Die Arrayimplementierung ist **korrekt**, d. h. sie erzeugt auf Eingabe $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell,

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen (wobei *empty* nur als Op_0 vorkommen darf). Im mathematischen Modell wird davon eine Folge $s_0, s_1, s_2, \ldots, s_N$ von Stacks (Tupeln) und eine Folge $z_0, z_1, z_2, \ldots, z_N$ von Ausgaben erzeugt.

Satz 2.1.2

Die Arrayimplementierung ist **korrekt**, d. h. sie erzeugt auf Eingabe $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler (*top* oder *pop* auf leerem Stack) auftritt

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen (wobei *empty* nur als Op_0 vorkommen darf). Im mathematischen Modell wird davon eine Folge $s_0, s_1, s_2, \ldots, s_N$ von Stacks (Tupeln) und eine Folge $z_0, z_1, z_2, \ldots, z_N$ von Ausgaben erzeugt.

Satz 2.1.2

Die Arrayimplementierung ist **korrekt**, d. h. sie erzeugt auf Eingabe $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler (*top* oder *pop* auf leerem Stack) auftritt **noch** die Höhe des Stacks s_i (d. h. die Länge der Folge s_i) die Arraygröße m überschreitet.

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

> Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

(IB_i)

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB$_i$) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- `p` enthält die Länge n_i der Folge s_i

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB$_i$) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- p enthält die Länge n_i der Folge s_i und
- $s_i = (\mathtt{A}[n_i], \ldots, \mathtt{A}[1])$ und

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

(IB_i)
- p enthält die Länge n_i der Folge s_i und
- $s_i = (\mathtt{A}[n_i], \ldots, \mathtt{A}[1])$ und
- die Implementierung gibt in Schritt i genau z_i aus.

D. h.: `A[1..p]` stellt genau s_i dar, in umgekehrter Reihenfolge.

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i)

Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- p enthält die Länge n_i der Folge s_i und
- $s_i = (\texttt{A}[n_i], \ldots, \texttt{A}[1])$ und
- die Implementierung gibt in Schritt i genau z_i aus.

D. h.: `A[1..p]` stellt genau s_i dar, in umgekehrter Reihenfolge.

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i)

Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- p enthält die Länge n_i der Folge s_i und
- $s_i = (\mathtt{A}[n_i], \ldots, \mathtt{A}[1])$ und
- die Implementierung gibt in Schritt i genau z_i aus.

D. h.: $\mathtt{A}[1\,..\,\mathtt{p}]$ stellt genau s_i dar, in umgekehrter Reihenfolge.

Beachte die folgende Subtilität, die die Arrayimplementierung von der Listenimplementierung unterscheidet: Der Teil $\mathtt{A}[n_i + 1\,..\,m]$ des Arrays kann völlig beliebige Einträge enthalten. (Das sind die $*$-Einträge im Bild auf Folie 19.)

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i)

Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- p enthält die Länge n_i der Folge s_i und
- $s_i = (\mathtt{A}[n_i], \ldots, \mathtt{A}[1])$ und
- die Implementierung gibt in Schritt i genau z_i aus.

D. h.: $\mathtt{A}[1..\mathtt{p}]$ stellt genau s_i dar, in umgekehrter Reihenfolge.

Beachte die folgende Subtilität, die die Arrayimplementierung von der Listenimplementierung unterscheidet: Der Teil $\mathtt{A}[n_i + 1..m]$ des Arrays kann völlig beliebige Einträge enthalten. (Das sind die $*$-Einträge im Bild auf Folie 19.) D. h.: Ein und derselbe Stack s im Sinn des mathematischen Modells kann durch verschieden beschriftete Arrays $\mathtt{A}[1..m]$ dargestellt werden.

Lesehinweise und Kommentare

- Wir schreiben den Beweis teilweise auf, ein letztes Mal (Druckfolien).
- Besser ist: Nicht gleich lesen, sondern entlang des Beweises für die Listenimplementierung selber konstruieren, dann kontrollieren.
- Wieder zeigt sich, dass solche Beweise recht umständlich sind, und auch eher langweilig, wenn die richtige Induktionsbehauptung erst einmal gefunden ist.
- Korrektheitsbeweise auf der Basis der axiomatischen Methode sind viel einfacher! Man muss da nicht ganze Operationsfolgen betrachten und Induktionsbeweise führen, sondern kann sich auf die Effekte von maximal zwei aufeinanderfolgenden Operationen beschränken.
- Die Schwierigkeiten bei der axiomatischen Methode liegen an anderer Stelle, nämlich beim Finden eines geeigneten Axiomensatzes für einen geplanten Datentyp.

Details des Induktionsbeweises von Satz 2.1.2:

I.A.: $i = 0$.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$.

Details des Induktionsbeweises von Satz 2.1.2:

I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB$_{i-1}$) gilt.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB$_{i-1}$) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und
$s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow$ $(\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und
$s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.
Fall 1: $\text{Op}_i = \texttt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und $s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.
Fall 1: $\text{Op}_i = \texttt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.
Wenn $n_{i-1} < m$ ist, wird von der Prozedur $\texttt{s.push}(x)$ das Objekt x an die Stelle $\texttt{A}[n_{i-1} + 1]$ gesetzt und p auf den Wert $n_{i-1} + 1 = n_i$ erhöht. Daraus folgt (IB_i).

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und $s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.
Fall 1: $\text{Op}_i = \texttt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.
Wenn $n_{i-1} < m$ ist, wird von der Prozedur $\texttt{s.push}(x)$ das Objekt x an die Stelle $\texttt{A}[n_{i-1} + 1]$ gesetzt und p auf den Wert $n_{i-1} + 1 = n_i$ erhöht. Daraus folgt (IB_i). Wenn aber $n_{i-1} = m$ ist, tritt in der Implementierung ein Fehler ein, und (IB_i) ist trivialerweise erfüllt.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und $s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.
Fall 1: $\text{Op}_i = \texttt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.
Wenn $n_{i-1} < m$ ist, wird von der Prozedur $\texttt{s.push}(x)$ das Objekt x an die Stelle $\texttt{A}[n_{i-1} + 1]$ gesetzt und p auf den Wert $n_{i-1} + 1 = n_i$ erhöht. Daraus folgt (IB_i). Wenn aber $n_{i-1} = m$ ist, tritt in der Implementierung ein Fehler ein, und (IB_i) ist trivialerweise erfüllt.
Fall 2: $\text{Op}_i = \texttt{s.top()}$. – Dann ist $s_i = (a_1, \ldots, a_{n_{i-1}})$, und auch in der Implementierung ändert sich nichts im Stackobjekt.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \text{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und $s_{i-1} = (\texttt{A}[n_{i-1}], \ldots, \texttt{A}[1])$.
Fall 1: $\text{Op}_i = \texttt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.
Wenn $n_{i-1} < m$ ist, wird von der Prozedur $\texttt{s.push}(x)$ das Objekt x an die Stelle $\texttt{A}[n_{i-1} + 1]$ gesetzt und p auf den Wert $n_{i-1} + 1 = n_i$ erhöht. Daraus folgt (IB_i). Wenn aber $n_{i-1} = m$ ist, tritt in der Implementierung ein Fehler ein, und (IB_i) ist trivialerweise erfüllt.
Fall 2: $\text{Op}_i = \texttt{s.top()}$. – Dann ist $s_i = (a_1, \ldots, a_{n_{i-1}})$, und auch in der Implementierung ändert sich nichts im Stackobjekt. Wenn $n_{i-1} \geq 1$ ist, wird in der Implementierung $\texttt{A}[n_{i-1}] = a_1$ ausgegeben, ebenso wie im mathematischen Modell. Wenn $n_{i-1} = 0$, erzeugt die top-Operation im mathematischen Modell einen Fehler, und wieder gilt (IB_i). (Mathematisches Modell und Implementierung melden den Fehler.)
Die anderen Fälle, entsprechend den anderen Operationen, behandelt man analog.

Zeitaufwand der Arrayimplementierung

Zeitaufwand der Arrayimplementierung

Behauptung 2.1.3

Bei der Arrayimplementierung von Stacks hat jede einzelne Operation Kosten $O(1)$.

Zeitaufwand der Arrayimplementierung

Behauptung 2.1.3

Bei der Arrayimplementierung von Stacks hat jede einzelne Operation Kosten $O(1)$.

Das sieht man durch Betrachtung der einzelnen Operationen.

Zeitaufwand der Arrayimplementierung

Behauptung 2.1.3

Bei der Arrayimplementierung von Stacks hat jede einzelne Operation Kosten $O(1)$.

Das sieht man durch Betrachtung der einzelnen Operationen.

Bei der Initialisierung `empty()` (Aufruf des Konstruktors) muss man aber auf Folgendes achten: Wenn man m, die Arraygröße, vom Benutzer als Parameter angeben lässt (was sehr sinnvoll ist!), dann darf das Array `A[`$1..m$`]` vom Konstruktor nur **alloziert**, nicht initialisiert werden. In C++ ist dies möglich. Andere Programmiersprachen wie etwa Java initialisieren ein neu angelegtes Array automatisch. In diesem Fall hat `empty()` natürlich Rechenzeit $\Theta(m)$.

Zeitaufwand der Arrayimplementierung

Behauptung 2.1.3

Bei der Arrayimplementierung von Stacks hat jede einzelne Operation Kosten $O(1)$.

Das sieht man durch Betrachtung der einzelnen Operationen.

Bei der Initialisierung `empty()` (Aufruf des Konstruktors) muss man aber auf Folgendes achten: Wenn man m, die Arraygröße, vom Benutzer als Parameter angeben lässt (was sehr sinnvoll ist!), dann darf das Array `A[`$1..m$`]` vom Konstruktor nur **alloziert**, nicht initialisiert werden. In C++ ist dies möglich. Andere Programmiersprachen wie etwa Java initialisieren ein neu angelegtes Array automatisch. In diesem Fall hat `empty()` natürlich Rechenzeit $\Theta(m)$.

Man beobachte, dass die Korrektheit nicht beeinträchtigt wird, wenn das Array anfangs einen völlig beliebigen Inhalt hat.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

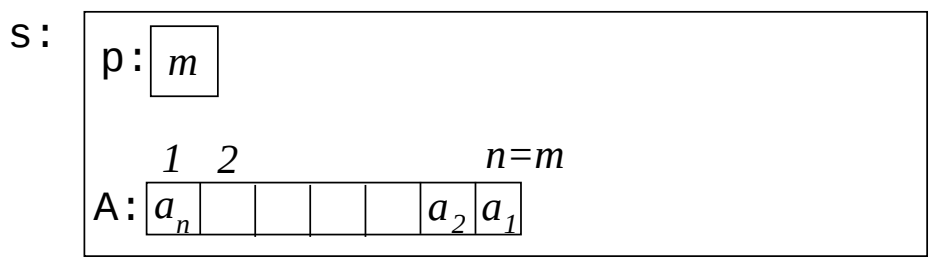
Vermeidung von Platzproblemen: Verdoppelungsstrategie

`s.push(`x`)` bei voller Tabelle.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

`s.push(`x`)` bei voller Tabelle.

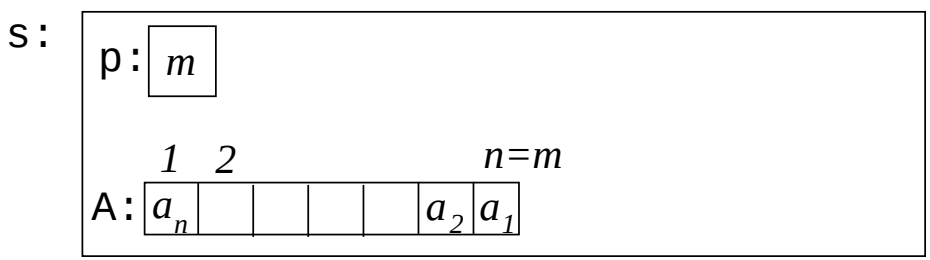
Vorher:



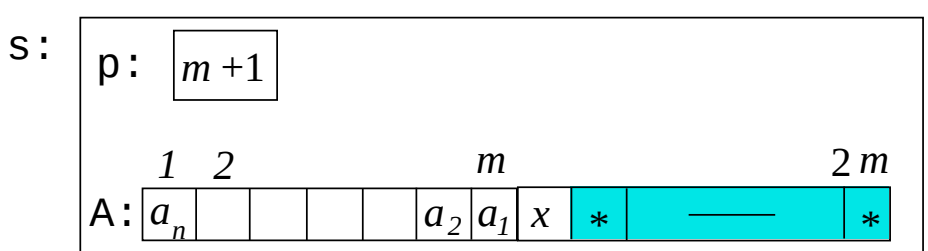
Vermeidung von Platzproblemen: Verdoppelungsstrategie

`s.push(`x`)` bei voller Tabelle.

Vorher:



Nachher:



Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.
3. x an Stelle $n+1$ schreiben, Pegel auf $n+1$ setzen.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.
3. x an Stelle $n+1$ schreiben, Pegel auf $n+1$ setzen.
4. `AA` in `A` umbenennen/Referenz umsetzen.

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.
3. x an Stelle $n+1$ schreiben, Pegel auf $n+1$ setzen.
4. `AA` in `A` umbenennen/Referenz umsetzen.

   (Altes Array freigeben, falls in der Programmiersprache sinnvoll. (C++ ja, Java nein.))

Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.
3. x an Stelle $n+1$ schreiben, Pegel auf $n+1$ setzen.
4. `AA` in `A` umbenennen/Referenz umsetzen.
   (Altes Array freigeben, falls in der Programmiersprache sinnvoll. (C++ ja, Java nein.))

Kosten: $\Theta(m)$, wo $m = n =$ aktuelle Größe des Stacks.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$,

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1})$$

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big)$$

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big) \overset{(**)}{=} O\big(m_0 \cdot 2^L\big)$$

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big) \overset{(**)}{=} O\big(m_0 \cdot 2^L\big) = O(k).$$

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big) \overset{(**)}{=} O\big(m_0 \cdot 2^L\big) = O(k).$$

Die **Gesamtkosten** für alle Verdopplungen zusammen sind also $N \cdot \Theta(1) + O(k) = \Theta(N)$.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein:
von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big) \overset{(**)}{=} O\big(m_0 \cdot 2^L\big) = O(k).$$

Die **Gesamtkosten** für alle Verdopplungen zusammen sind also $N \cdot \Theta(1) + O(k) = \Theta(N)$.

Bemerkung: Wir haben für $\overset{(*)}{=}$ die Summationsregel für die O-Notation und für $\overset{(**)}{=}$ die Summenformel für geometrische Reihen benutzt.

Satz 2.1.4

Wenn ein Stack mit **Arrays** und der **Verdoppelungsstrategie** implementiert wird, gilt:
(a) Die Gesamtkosten für N Operationen betragen $\Theta(N)$.

Satz 2.1.4

Wenn ein Stack mit **Arrays** und der **Verdoppelungsstrategie** implementiert wird, gilt:
(a) Die Gesamtkosten für N Operationen betragen $\Theta(N)$.
(b) Der gesamte Platzbedarf ist $\Theta(k)$, wenn k die maximale je erreichte Stackhöhe ist.

Satz 2.1.4

Wenn ein Stack mit **Arrays** und der **Verdoppelungsstrategie** implementiert wird, gilt:
(a) Die Gesamtkosten für N Operationen betragen $\Theta(N)$.
(b) Der gesamte Platzbedarf ist $\Theta(k)$, wenn k die maximale je erreichte Stackhöhe ist.

Bemerkung: Selbst wenn der Speicher nie bereinigt wird (d. h. wenn kleinere, nicht mehr benutzte Arrays nicht freigegeben werden), tritt ein Speicherüberlauf erst ein, wenn die Zahl der Stackeinträge zu einem bestimmten Zeitpunkt mehr Speicher beansprucht als $\frac{1}{4}$ des gesamten zur Verfügung stehenden Speichers.

Satz 2.1.4

Wenn ein Stack mit **Arrays** und der **Verdoppelungsstrategie** implementiert wird, gilt:
(a) Die Gesamtkosten für N Operationen betragen $\Theta(N)$.
(b) Der gesamte Platzbedarf ist $\Theta(k)$, wenn k die maximale je erreichte Stackhöhe ist.

Bemerkung: Selbst wenn der Speicher nie bereinigt wird (d. h. wenn kleinere, nicht mehr benutzte Arrays nicht freigegeben werden), tritt ein Speicherüberlauf erst ein, wenn die Zahl der Stackeinträge zu einem bestimmten Zeitpunkt mehr Speicher beansprucht als $\frac{1}{4}$ des gesamten zur Verfügung stehenden Speichers.

Bemerkung: Was wir gemacht haben, nämlich die *Summe* der Kosten einer Reihe von Operationen abzuschätzen (anstatt einfach die Anzahl der Operationen mit den maximalen Kosten zu multiplizieren), nennt man **„amortisierte Analyse“**. Dieser Ansatz wird uns noch mehrfach begegnen.

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),
- Höchstens 50% des allozierten Speicherplatzes bleibt ungenutzt.

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),
- Höchstens 50% des allozierten Speicherplatzes bleibt ungenutzt.

Listenimplementierung

- $O(1)$ Kosten pro Operation **im schlechtesten Fall**
  (jedoch: relativ hohe Allokationskosten bei *push*-Operationen),

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),
- Höchstens 50% des allozierten Speicherplatzes bleibt ungenutzt.

Listenimplementierung

- $O(1)$ Kosten pro Operation **im schlechtesten Fall**
  (jedoch: relativ hohe Allokationskosten bei *push*-Operationen),
- Zusätzlicher Platz für Zeiger benötigt.

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),
- Höchstens 50% des allozierten Speicherplatzes bleibt ungenutzt.

Listenimplementierung

- $O(1)$ Kosten pro Operation **im schlechtesten Fall**
  (jedoch: relativ hohe Allokationskosten bei *push*-Operationen),
- Zusätzlicher Platz für Zeiger benötigt.

Experimente zeigen, dass die Arrayimplementierung spürbar kleinere Rechenzeiten aufweist.

Lesehinweise und Kommentare

- Damit endet die gründliche Diskussion des Datentyps „Stack“ mit zwei Implementierungsvarianten. Wir haben gesehen:
  - Spezifikationsmethode über mathematische Modelle.
  - Verwendung eines solchen Modells in Korrektheitsbeweisen, per Induktion über Operationen auf der Datenstruktur.
  - Listen- und Arrayimplementierung.
  - Prinzipiell sehr einfache Kostenanalyse in beiden Fällen.
  - Eine einfache amortisierte Analyse, für die Arrayimplementierung mit Verdoppelungsstrategie.
- Generelle Erkenntnis: Datentypen lassen sich ebenso exakt spezifizieren wie Berechnungsprobleme.

PAUSE

Lesehinweise und Kommentare

- Wir benutzen jetzt unsere Erkenntnisse über Stacks, um einen vielseitig verwendbaren Datentyp zu spezifizieren und zu implementieren, nämlich ein Array mit variabler Länge.
- Man kann ein solches Array an einer Seite („hinten“ bzw. „oben“) um einen Eintrag verlängern oder auch verkürzen, so wie bei Stacks.
- Gleichzeitig kann man jederzeit über Indizes lesend und schreibend auf beliebige Positionen zugreifen, wie bei Arrays.
- Die simplen Arrays in Java haben immer eine feste Länge, die bei der Erzeugung fixiert wird, ebenso die in C++.
- Unsere Funktionalität ist ein Teil der Funktionalität der Klasse `std::vector` in C++. Wir werden die Prinzipien hinter der Implementierung dieser Klasse sehen.
- Die `vector`-Klasse in Java hat eine andere Funktionalität!
- Achtung: Neu ist hier die Verwendung eines Datentyps für die natürlichen Zahlen, der als gegeben angesehen wird. In [AuP], Kap. 8, wurde dies als ADT Nat mit den Operationen 0, *succ* und *add* beschrieben.

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.
Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert.

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:

	1	2	3	4	5	6	7	8	9	10
A:	e	h	x	a	c	f	u	z	l	q

Aktuelle Länge: $n = 10$

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:

	1	2	3	4	5	6	7	8	9	10
A:	e	h	x	a	c	f	u	z	l	q

Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:

	1	2	3	4	5	6	7	8	9	10
A:	e	h	x	a	c	f	u	z	l	q

Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

empty(): liefert leeres Array, Länge 0

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:

	1	2	3	4	5	6	7	8	9	10
A:	e	h	x	a	c	f	u	z	l	q

Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

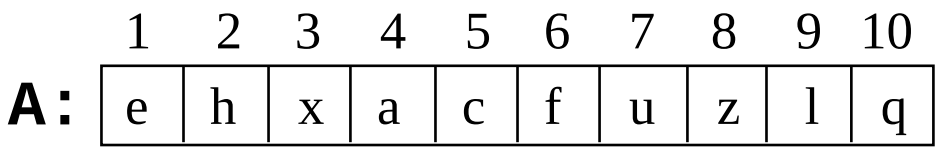
empty(): liefert leeres Array, Länge 0

read(A, i): liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

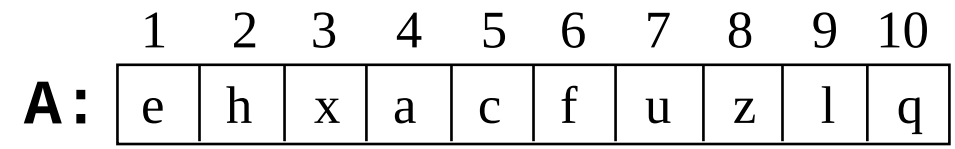
Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

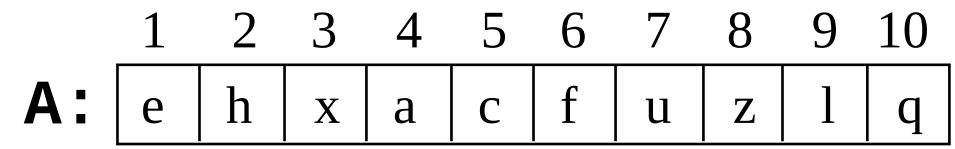
Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

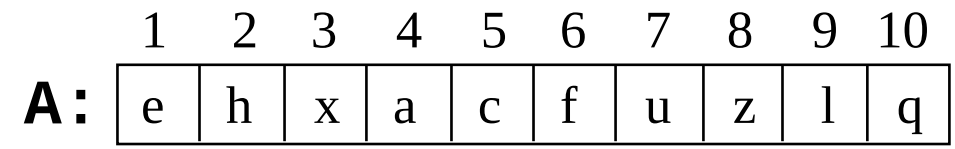
Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.
addLast(A, x):	verlängert Array um 1 Position, schreibt x an letzte Stelle ($\hat{=}$ push).

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.
addLast(A, x):	verlängert Array um 1 Position, schreibt x an letzte Stelle ($\hat{=}$ push).
removeLast(A):	verkürzt Array um eine Position ($\hat{=}$ pop).

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.
Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:

	1	2	3	4	5	6	7	8	9	10
A:	e	h	x	a	c	f	u	z	l	q

Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

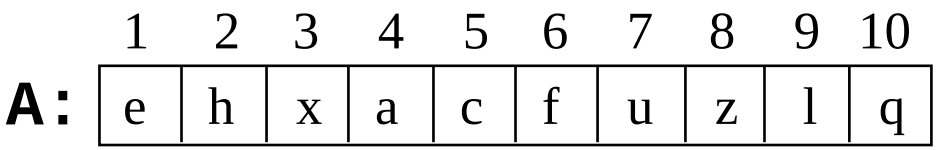
empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.
addLast(A, x):	verlängert Array um 1 Position, schreibt x an letzte Stelle ($\hat{=}$ push).
removeLast(A):	verkürzt Array um eine Position ($\hat{=}$ pop).

Wieso gibt es kein „isempty“?

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.
addLast(A, x):	verlängert Array um 1 Position, schreibt x an letzte Stelle ($\hat{=}$ push).
removeLast(A):	verkürzt Array um eine Position ($\hat{=}$ pop).

Wieso gibt es kein „isempty“? (Teste *length*(A) auf 0.)

Dynamische Arrays

1. Signatur:

Dynamische Arrays

1. Signatur:
Sorten:

Dynamische Arrays

1. Signatur:

Sorten: *Elements*

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat

Dynamische Arrays

1. Signatur:

Sorten: *Elements*
Arrays
Nat

Operationen:

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$
	$removeLast: Arrays \rightarrow Arrays$

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$
	$removeLast: Arrays \rightarrow Arrays$
	$read: Arrays \times Nat \rightarrow Elements$

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$
	$removeLast: Arrays \rightarrow Arrays$
	$read: Arrays \times Nat \rightarrow Elements$
	$write: Arrays \times Nat \times Elements \rightarrow Arrays$

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$
	$removeLast: Arrays \rightarrow Arrays$
	$read: Arrays \times Nat \rightarrow Elements$
	$write: Arrays \times Nat \times Elements \rightarrow Arrays$
	$length: Arrays \rightarrow Nat$

2. Mathematisches Modell

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)

2. Mathematisches Modell

Sorten:	*Elements*:	(nichtleere) Menge D	(Parameter)
	Arrays:	Seq(D)	

2. Mathematisches Modell

Sorten:	*Elements*:	(nichtleere) Menge D	(Parameter)
	Arrays:	Seq(D)	
	Nat:	$\mathbb{N}$	

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$empty() := ()$

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$\mathit{empty}() := ()$

$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$\mathit{empty}() := ()$

$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$

$$\mathit{removeLast}((a_1, \ldots, a_n)) := \begin{cases} (a_1, \ldots, a_{n-1}), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{removeLast}((a_1, \ldots, a_n)) := \begin{cases} (a_1, \ldots, a_{n-1}), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{read}((a_1, \ldots, a_n), i) := \begin{cases} a_i, & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{removeLast}((a_1, \ldots, a_n)) := \begin{cases} (a_1, \ldots, a_{n-1}), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{read}((a_1, \ldots, a_n), i) := \begin{cases} a_i, & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

$$\mathit{write}((a_1, \ldots, a_n), i, x) := \begin{cases} (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n), & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{removeLast}((a_1, \ldots, a_n)) := \begin{cases} (a_1, \ldots, a_{n-1}), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{read}((a_1, \ldots, a_n), i) := \begin{cases} a_i, & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

$$\mathit{write}((a_1, \ldots, a_n), i, x) := \begin{cases} (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n), & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

$$\mathit{length}((a_1, \ldots, a_n)) := n$$

Lesehinweise und Kommentare

- Die Erläuterungen zu den Festlegungen im mathematischen Modell entsprechen denen für Stacks.
- Welche Fehlermöglichkeiten gibt es in diesem Modell? *removeLast* bei aktueller Länge $n = 0$, und „index out of bounds“-Fehler, wenn *read* oder *write* mit einem Index aufgerufen werden, der nicht zwischen 1 und n liegt.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten:

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.
Nachteil: Diese Operation kostet $\Theta(n)$ Zeit.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.
Nachteil: Diese Operation kostet $\Theta(n)$ Zeit.
Vorteil: Umfang der Datenstruktur ist **stets** $\leq 4n$, für aktuelle Arraylänge n.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.
Nachteil: Diese Operation kostet $\Theta(n)$ Zeit.
Vorteil: Umfang der Datenstruktur ist **stets** $\leq 4n$, für aktuelle Arraylänge n.

Mitteilung
Startend mit einem leeren dynamischen Array, kosten k *addLast*- und *removeLast*-Operationen bei der Arrayimplementierung mit Verdoppelung und Halbierung insgesamt Zeit $O(k)$.

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.
Nachteil: Diese Operation kostet $\Theta(n)$ Zeit.
Vorteil: Umfang der Datenstruktur ist **stets** $\leq 4n$, für aktuelle Arraylänge n.

Mitteilung

Startend mit einem leeren dynamischen Array, kosten k *addLast*- und *removeLast*-Operationen bei der Arrayimplementierung mit Verdoppelung und Halbierung insgesamt Zeit $O(k)$.
Der Beweis benutzt etwas fortgeschrittenere Techniken zur „amortisierten Analyse“.

Lesehinweise und Kommentare

- Wir wenden uns nun einem weiteren grundlegenden Datentyp, der „Queue“ (gesprochen „kju“) oder „Warteschlange“ oder „FIFO-Liste“ zu.
- Es gibt Ähnlichkeiten mit Stacks, aber große Unterschiede in der Funktionalität.
- Wir werden interessante Anwendungen von Queues im Kontext von Graphalgorithmen sehen („Breitensuche“).
- Eine Queue kann man sich als Kette mit Einträgen vorstellen, mit einem vorderen Ende („head“) und einem hinteren Ende („tail“).
- **„enqueue“:** Man fügt ein Element ein, indem man hinten an die Kette anhängt.
- **„dequeue“:** Man entnimmt ein Element am vorderen Ende der Kette.
- **„first“:** Man kann den vordersten Eintrag auch lesen.
- Durch die Regeln werden die Elemente in genau derselben Reihenfolge entnommen, in der sie eingefügt wurden („First-In-First-Out“).
- Queues werden in [AuP], Kap. 8, als Beispiel eines ADTs betrachtet.

2.2 Queues (Warteschlangen, FIFO*-Listen)

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

2.2 Queues (Warteschlangen, FIFO*-Listen)

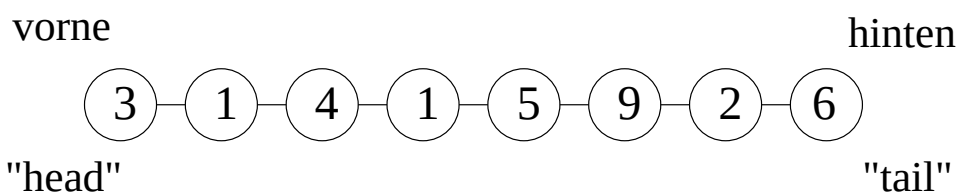
Beispiel:

* **F**irst-**I**n-**F**irst-**O**ut-Listen

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

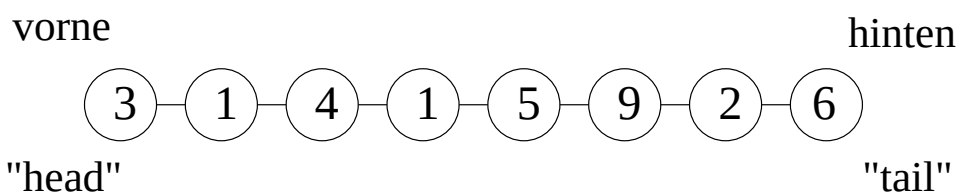
* **F**irst-**I**n-**F**irst-**O**ut-Listen



2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

* First-In-First-Out-Listen

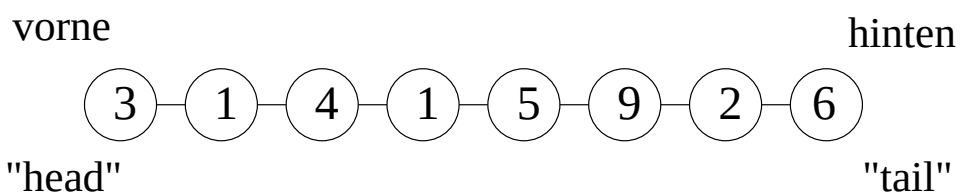


isempty: *false*

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

* **F**irst-**I**n-**F**irst-**O**ut-Listen



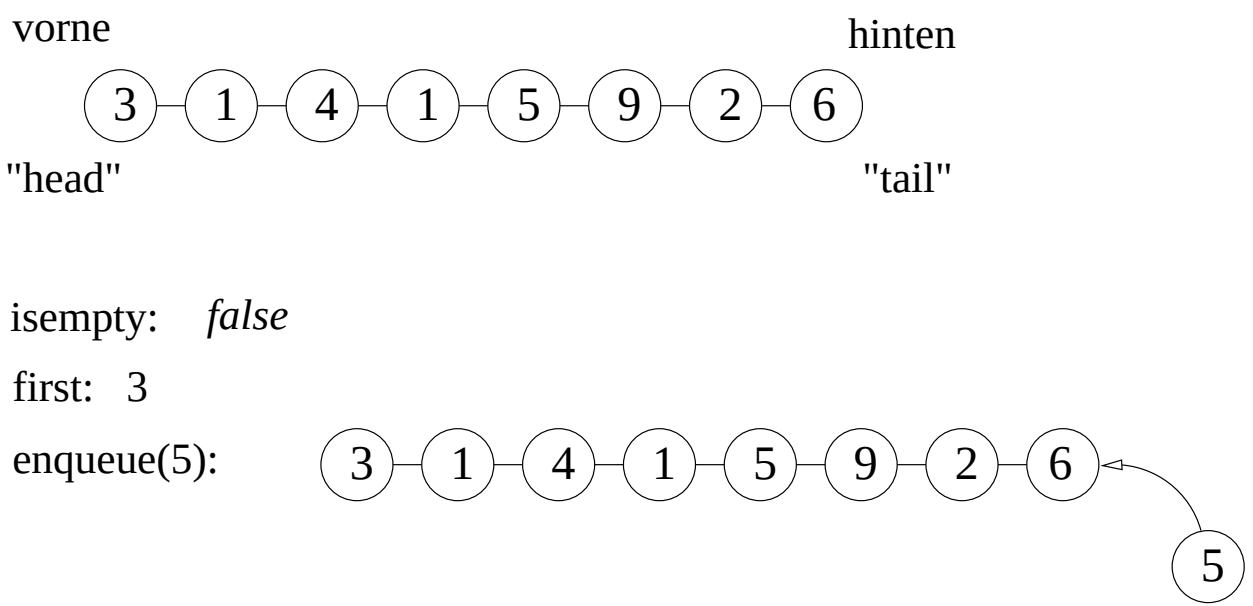
isempty: *false*

first: 3

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

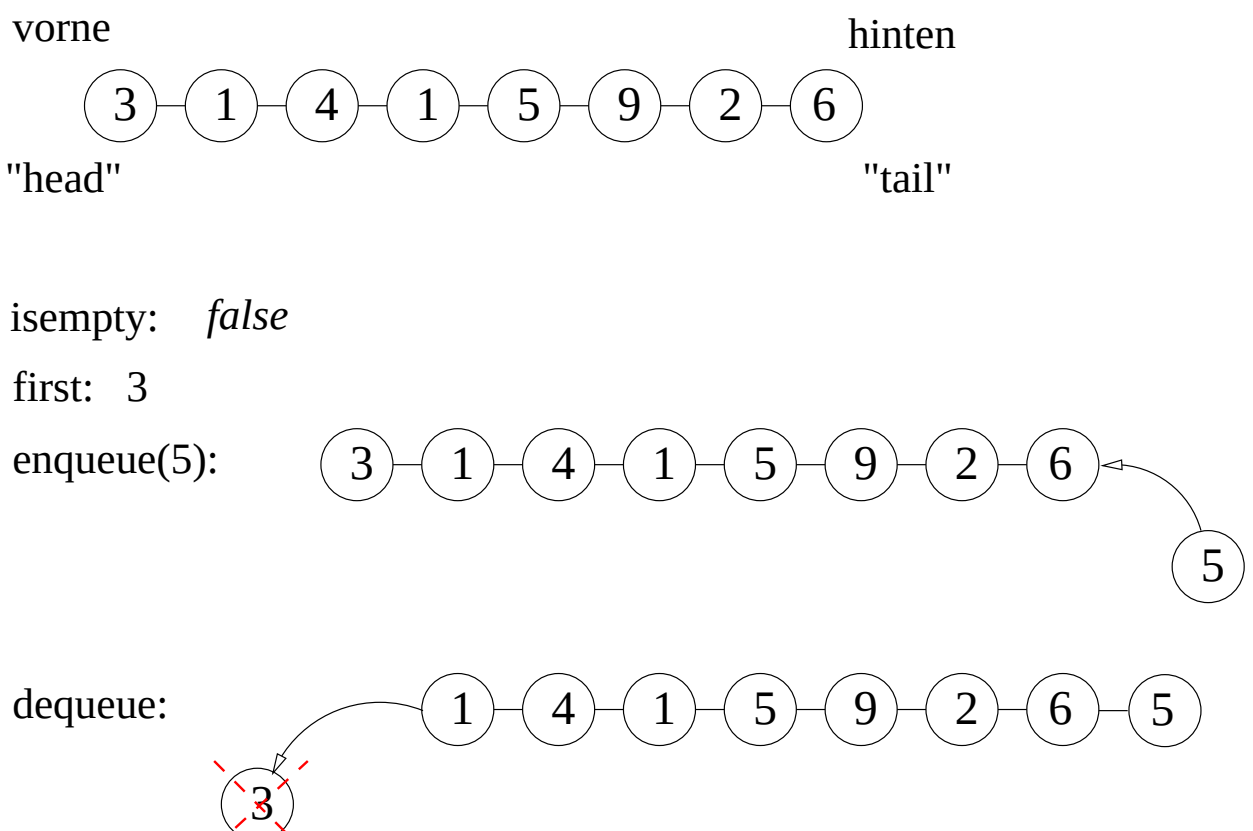
* **F**irst-**I**n-**F**irst-**O**ut-Listen



2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

* **F**irst-**I**n-**F**irst-**O**ut-Listen



Lesehinweise und Kommentare

- Wir gehen nun zur Spezifikation und zur Implementierung ganz genau wie bei Stacks vor.
- Bemerkenswert an der folgenden Signatur ist, dass sie bis auf Umbenennungen identisch ist zur Signatur der Stack-Spezifikation.
- Stacks und Queues verhalten sich sehr unterschiedlich.
- Dadurch wird nochmals deutlich, dass die Signatur über die Funktionalität eines Datentyps praktisch gar nichts aussagt.

Spezifikation des Datentyps „Queue“

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten: *Elements*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten: *Elements*
Queues
Boolean

Operationen:

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*
	isempty: *Queues* $\rightarrow$ *Boolean*

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*
	isempty: *Queues* $\rightarrow$ *Boolean*

Beachte: Die Signatur ist **identisch** zur **Signatur von Stacks** – bis auf Umbenennungen.

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*
	isempty: *Queues* $\rightarrow$ *Boolean*

Beachte: Die Signatur ist **identisch** zur **Signatur von Stacks** – bis auf Umbenennungen.
Rein syntaktische Vorschriften!

Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*
	isempty: *Queues* $\rightarrow$ *Boolean*

Beachte: Die Signatur ist **identisch** zur **Signatur von Stacks** – bis auf Umbenennungen.
Rein syntaktische Vorschriften! Verhalten (noch) ungeklärt!

2. Mathematisches Modell

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)
Queues: Seq(D)

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)
Queues: Seq(D)
Boolean: $\{true, false\}$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)
Queues: Seq(D)
Boolean: $\{true, false\}$

Die leere Folge $()$ stellt die leere Warteschlange dar.

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen:

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: $\mathit{empty}() := ()$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: *empty*() := ()

enqueue$((a_1, \ldots, a_n), x) :=$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: *empty*() := ()

enqueue$((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: *empty*() := ()

enqueue$((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$

dequeue$((a_1, \ldots, a_n)) :=$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: *empty*() := ()

enqueue$((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$

$$\textit{dequeue}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{enqueue}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{dequeue}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{first}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen: *empty*() := ()

$$\textit{enqueue}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\textit{dequeue}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\textit{first}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\textit{isempty}((a_1, \ldots, a_n)) := \begin{cases} \mathit{false}, & \text{falls } n \geq 1 \\ \mathit{true}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des ADTs „Queue“ über D

Alternative:

Spezifikation des ADTs „Queue“ über D

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

Spezifikation des ADTs „Queue“ über D

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, q$: *Queue*, $\forall\, x, y$: *Elements*

$$\mathit{dequeue}(\mathit{enqueue}(\mathit{empty}(), x)) = \mathit{empty}()$$

$$\mathit{first}(\mathit{enqueue}(\mathit{empty}(), x)) = x$$

$$\mathit{dequeue}(\mathit{enqueue}(\mathit{enqueue}(q, x), y)) = \mathit{enqueue}(\mathit{dequeue}(\mathit{enqueue}(q, x)), y)$$

$$\mathit{first}(\mathit{enqueue}(\mathit{enqueue}(q, x), y)) = \mathit{first}(\mathit{enqueue}(q, x))$$

$$\mathit{isempty}(\mathit{empty}()) = \mathit{true}$$

$$\mathit{isempty}(\mathit{enqueue}(q, x)) = \mathit{false}$$

Kommentar

- Das mathematische Modell ist fast identisch zu dem für Stacks. Nur fügt *enqueue*(x) das Element x hinten an die Folge $(a_1, \ldots, a_n)$ an, während *push*(x) das Element x vorne anfügt.
- Wir sagen noch etwas zur Spezifikation mit Axiomen. Nächste Folie: Sechs Axiome für „Queues“.
- Die ersten beiden Axiome beschreiben das Verhalten, wenn man an eine leere Queue ein Element anhängt: Entnehmen des ersten Elements liefert die leere Queue, Lesen des ersten Elements liefert, was gerade angehängt wurde.
- Die mittleren Axiome beschreiben das Ergebnis q'', wenn man an eine nichtleere Queue $q' =$*enqueue*(q, x) ein Element y anhängt. Der erste Eintrag von q'' ist einfach der erste Eintrag von q'. Entnehmen des ersten Eintrags aus q'' liefert dieselbe Queue, die sich ergibt, wenn man erst aus q' den ersten Eintrag entnimmt und dann y anhängt.
- Die letzten beiden Axiome beschreiben, wie bei Stacks, das korrekte Verhalten von *isempty*.
- Die Axiome sind natürlich, wenn man sie liest und überlegt, was sie bedeuten, aber es gibt zwei Fragen, für Queues und für allgemeine (neue) Datentypen:
  - Wie findet man geeignete Axiome?
  - Woher weiß man, dass die Axiome den geplanten Datentyp komplett beschreiben?
- Diese Fragen stellen sich für jeden neuen Datentyp neu und sind i. a. nicht leicht zu beantworten.

Lesehinweise und Kommentare

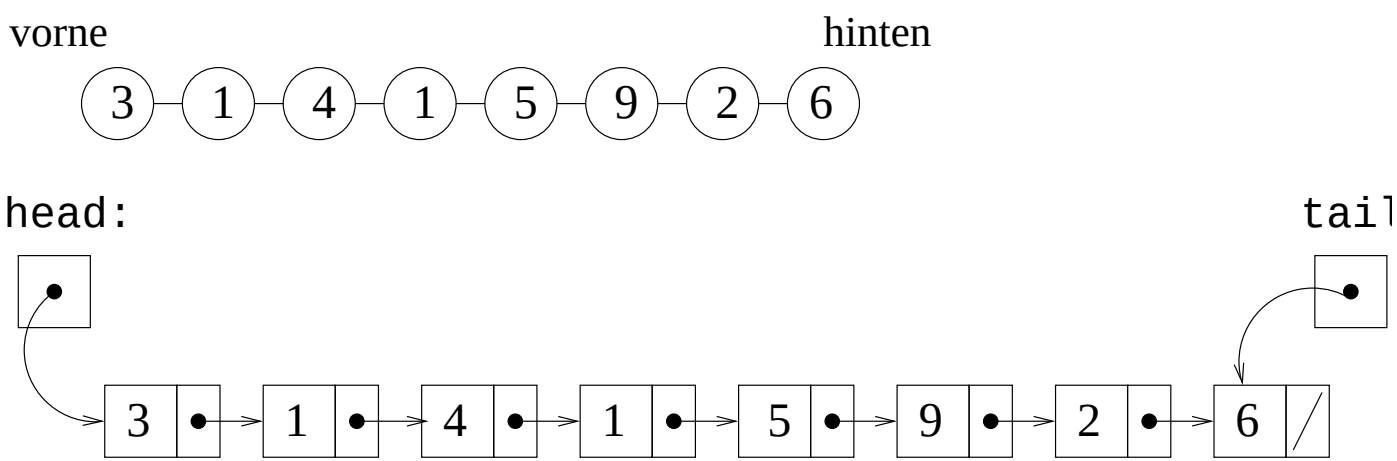
- Wir kommen nun zu Implementierungen von Queues. Wie bei Stacks gibt es die beiden Möglichkeiten, verkettete Listen oder auch Arrays zu benutzen.
- Bei der Verwendung von Listen ist zu beachten, dass man mittels Zeigern/Referenzen Zugang zu **beiden Enden** der Liste benötigt: mit einem „head“-Zeiger hat man Zugriff auf den Anfang, mit einem „tail“-Zeiger Zugriff auf das Ende.

Listenimplementierung von Queues

Implementierung mittels einfach verketteter Listen
mit Listenendezeiger/-referenz: Skizze.

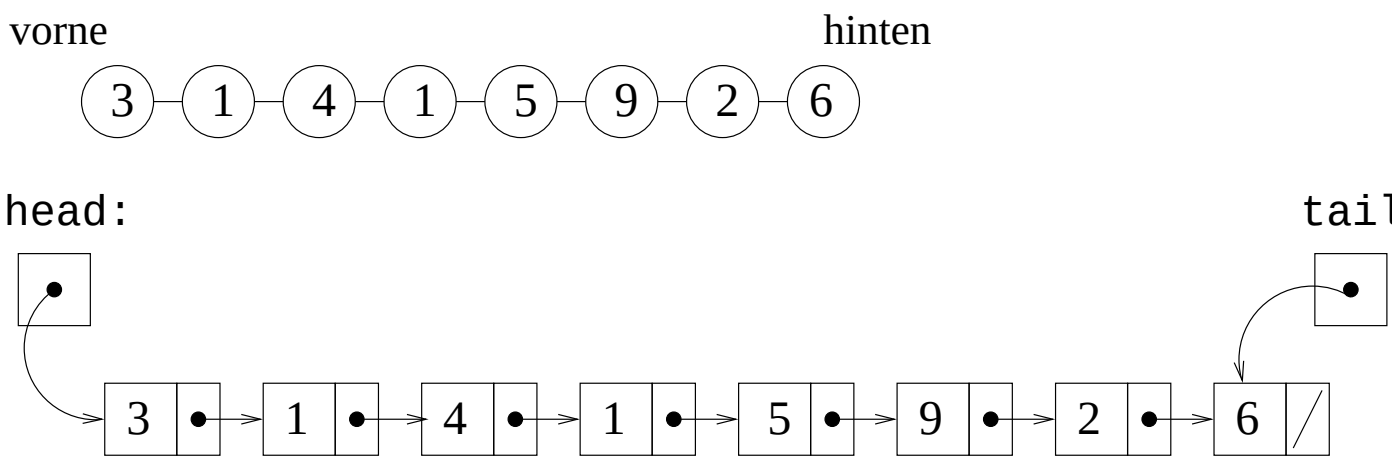
Listenimplementierung von Queues

Implementierung mittels einfach verketteter Listen
mit Listenendezeiger/-referenz: Skizze.



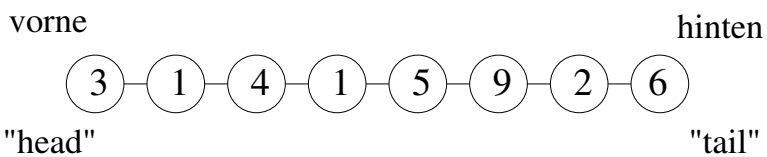
Listenimplementierung von Queues

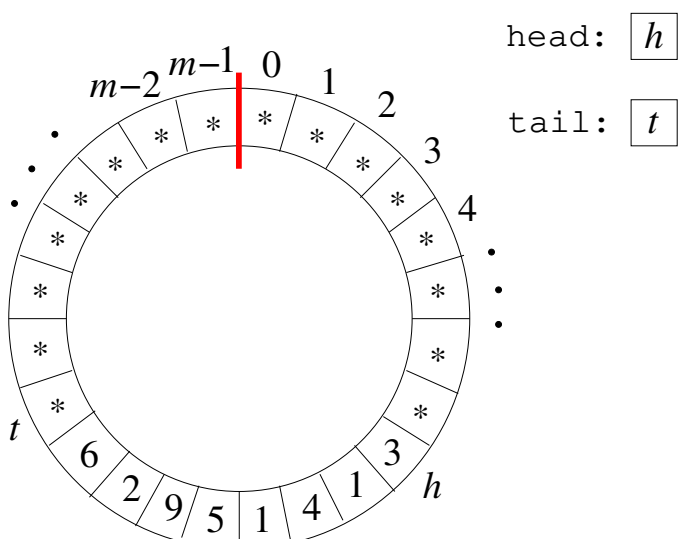
Implementierung mittels einfach verketteter Listen
mit Listenendezeiger/-referenz: Skizze.



Zu Übung überlege man sich, wie die Operationen zu implementieren wären.
Etwas knifflig: Umgang mit der Situation, dass Liste leer wird. (Übung!)

Arrayimplementierung von Queues





Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:

Das Queueobjekt besteht aus einem Array `A`$[0..m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array `A`$[0..m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt.

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:

Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:
Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.
Die Queue ist leer genau dann wenn $h = t$ gilt.

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:
Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.
Die Queue ist leer genau dann wenn $h = t$ gilt.

Die Zellen $\texttt{A}[t], \texttt{A}[t+1], \ldots, \texttt{A}[h-1]$ sind leer (im Bild mit $*$ markiert). Mindestens $\texttt{A}[t]$ ist leer. D. h.: Die maximale Anzahl n der Einträge im Array **(die Kapazität)** ist $m-1$.

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:
Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.
Die Queue ist leer genau dann wenn $h = t$ gilt.

Die Zellen $\texttt{A}[t], \texttt{A}[t+1], \ldots, \texttt{A}[h-1]$ sind leer (im Bild mit $*$ markiert). Mindestens $\texttt{A}[t]$ ist leer. D. h.: Die maximale Anzahl n der Einträge im Array **(die Kapazität)** ist $m-1$.

Um x einzufügen, wird (sofern Platz ist), x nach $\texttt{A}[t]$ geschrieben und dann `tail` um 1 erhöht.

Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:
Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.
Die Queue ist leer genau dann wenn $h = t$ gilt.

Die Zellen $\texttt{A}[t], \texttt{A}[t+1], \ldots, \texttt{A}[h-1]$ sind leer (im Bild mit $*$ markiert). Mindestens $\texttt{A}[t]$ ist leer. D. h.: Die maximale Anzahl n der Einträge im Array **(die Kapazität)** ist $m-1$.

Um x einzufügen, wird (sofern Platz ist), x nach $\texttt{A}[t]$ geschrieben und dann `tail` um 1 erhöht.

Ein Überlauf (oder eine Verdopplung) wird ausgelöst, wenn beim Einfügen die Variable `tail` denselben Wert erhalten würde wie `head`. Dieses Kriterium macht es überflüssig, die Beladung n mitzuführen. – Wir beschreiben die Operationen im Einzelnen.

Implementierung der Operationen auf einer Queue `q`:

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements`

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:

if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:

if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument

Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;

`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:

if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:

if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument
Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else return `A[head]` ;

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument
Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else return `A[head]` ;

`q.dequeue`:

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument
Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else return `A[head]` ;

`q.dequeue`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)

Implementierung der Operationen auf einer Queue `q`:

`q.empty()`: // Konstruktor; Option: m als Argument
Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else return `A[head]` ;

`q.dequeue`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument
Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
`head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else return `A[head]` ;

`q.dequeue`:
if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

`q.enqueue(`x`)`:

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument
  Erzeuge Array `A[0..`$m-1$`] of elements` und zwei Variable `head, tail: int` ;
  `head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
  if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else return `A[head]` ;

`q.dequeue`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

`q.enqueue(`x`)`:
  `tt` $\leftarrow (\texttt{tail} + 1) \bmod m$;

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument
  Erzeuge Array `A[0..`$m-1$`] of elements` und zwei Variable `head, tail: int` ;
  `head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
  if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else return `A[head]` ;

`q.dequeue`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

`q.enqueue(`x`)`:
  `tt` $\leftarrow (\texttt{tail} + 1) \bmod m$;
  if `tt` $=$ `head` **then „Überlauf“** (z.B. *QOverflowException* (oder Verdopplung))

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument
  Erzeuge Array `A[0..`$m-1$`] of elements` und zwei Variable `head, tail: int` ;
  `head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
  if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else return `A[head]` ;

`q.dequeue`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

`q.enqueue(`x`)`:
  `tt` $\leftarrow (\texttt{tail} + 1) \bmod m$;
  if `tt` $=$ `head` **then „Überlauf“** (z.B. *QOverflowException* (oder Verdopplung))
    else `A[tail]` $\leftarrow x$; `tail` $\leftarrow$ `tt` ;

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Beweis: Betrachte eine Folge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Queueoperationen

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Beweis: Betrachte eine Folge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Queueoperationen, wobei *empty* nur ganz am Anfang vorkommen darf. Zeige dann durch Induktion über $i = 0, \ldots, N$ unter der Annahme, dass **im mathematischen Modell kein Fehler** auftritt, folgende **Behauptung** $(\mathrm{IB})_i$:

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Beweis: Betrachte eine Folge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Queueoperationen, wobei *empty* nur ganz am Anfang vorkommen darf. Zeige dann durch Induktion über $i = 0, \ldots, N$ unter der Annahme, dass **im mathematischen Modell kein Fehler** auftritt, folgende **Behauptung** $(\mathrm{IB})_i$:
Wenn nach Operationen $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_i$ der Inhalt der Queue $q_i = (a_1, \ldots, a_{n_i})$ ist, dann stehen nach Ausführung der Operationen in der Implementierung die Elemente

$$a_1, \ldots, a_{n_i} \text{ in den Positionen } \texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

(Indizes modulo m gerechnet), wobei h der Inhalt von `head` und t der Inhalt von `tail` ist.

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Beweis: Betrachte eine Folge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Queueoperationen, wobei *empty* nur ganz am Anfang vorkommen darf. Zeige dann durch Induktion über $i = 0, \ldots, N$ unter der Annahme, dass **im mathematischen Modell kein Fehler** auftritt, folgende **Behauptung** $(\mathrm{IB})_i$:
Wenn nach Operationen $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_i$ der Inhalt der Queue $q_i = (a_1, \ldots, a_{n_i})$ ist, dann stehen nach Ausführung der Operationen in der Implementierung die Elemente

$$a_1, \ldots, a_{n_i} \text{ in den Positionen } \texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

(Indizes modulo m gerechnet), wobei h der Inhalt von `head` und t der Inhalt von `tail` ist.
Insbesondere ist die Warteschlange leer genau dann wenn $h = t$ ist.

Satz 2.2.2

Bei einer Queue mit Arrays und der Verdoppelungsstrategie sind die Gesamtkosten für N Operationen $O(N)$.

Satz 2.2.2

Bei einer Queue mit Arrays und der Verdoppelungsstrategie sind die Gesamtkosten für N Operationen $O(N)$.
Platzbedarf: $O(k)$, wenn k die maximale je erreichte Länge der Queue ist.

Satz 2.2.2

Bei einer Queue mit Arrays und der Verdoppelungsstrategie sind die Gesamtkosten für N Operationen $O(N)$.
Platzbedarf: $O(k)$, wenn k die maximale je erreichte Länge der Queue ist.

Beweise: Analog zu Stacks.

Lesehinweise und Kommentare

- Zu guter Letzt diskutieren wir eine Verallgemeinerung von Warteschlangen, bei denen man sich nicht nur hinten, sondern auch vorne anstellen kann. Sie heißen „Doppelschlangen“, „double ended queues“ oder kurz „Deques“.
- Das ist nicht nur bei der Bedienung von gehandicapten Personen im Postamt oder beim Einkaufen nützlich, sondern auch in technischen Zusammenhängen, wo Objekte mehrfach bearbeitet werden und eventuell schnell weiterverarbeitet werden müssen und daher an einer Station mit Warteschlange priorisiert behandelt werden müssen.
- Durch die Erweiterung kann die Datenstruktur sowohl wie ein Stack als auch wie eine Queue operieren, aber eben auch beides mischen.
- Man kann eine solche Doppelschlange direkt als **doppelt verkettete Liste** oder in einem **Array** implementieren, mit im Wesentlichen denselben Kosten pro Operation wie bei Stacks oder Queues.
- Auch eine Implementierung mit zwei Stacks ist möglich. Dabei ist die Kostenanalyse eine nette Übung.

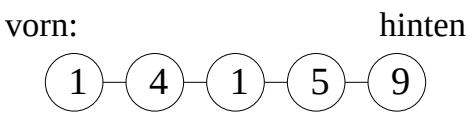
Datentyp Doppelschlange – Deque

Schlange mit zwei Enden; Einfügen und Entnehmen an beiden Enden möglich.

Datentyp Doppelschlange – Deque

Schlange mit zwei Enden; Einfügen und Entnehmen an beiden Enden möglich.

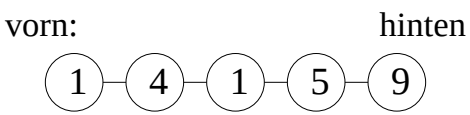
Schema:



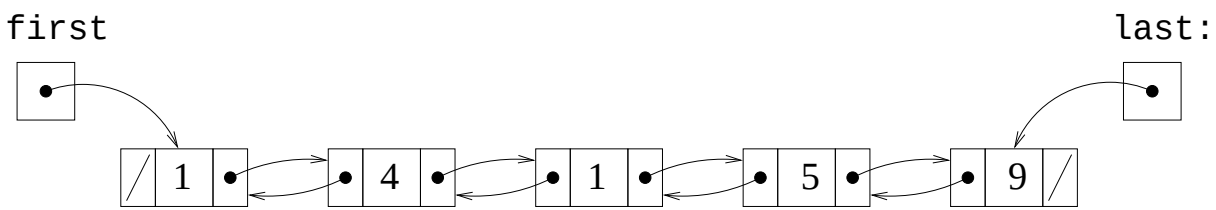
Datentyp Doppelschlange – Deque

Schlange mit zwei Enden; Einfügen und Entnehmen an beiden Enden möglich.

Schema:



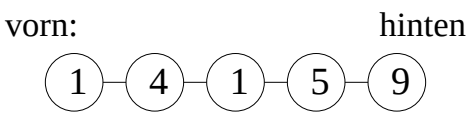
Einfache Implementierung:



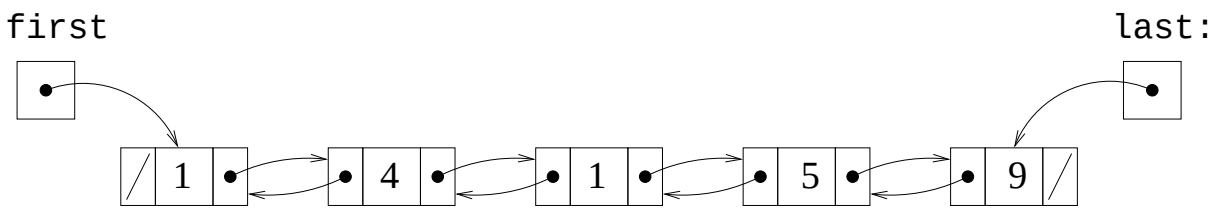
Datentyp Doppelschlange – Deque

Schlange mit zwei Enden; Einfügen und Entnehmen an beiden Enden möglich.

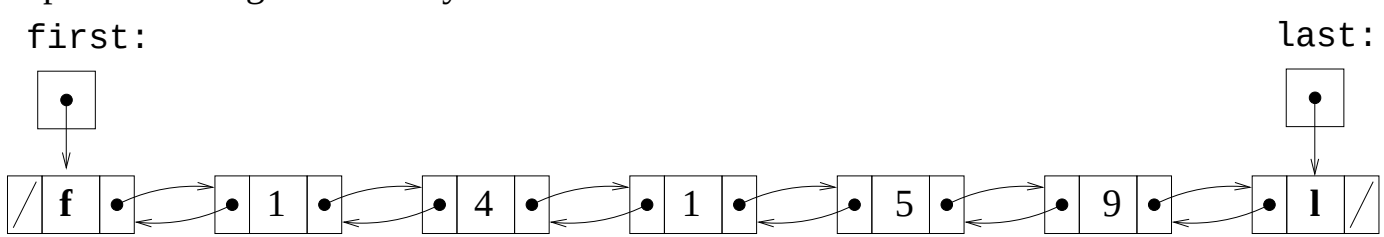
Schema:



Einfache Implementierung:



Implementierung mit Dummyelementen:



1. Signatur:

Sorten:	*Elements*	*Operationen*:	*empty*: . . .
	Deques		*pushFront*: . . .
	Boolean		*popFront*: . . .
			first: . . .
			pushBack: . . .
			popBack: . . .
			last: . . .
			isempty: . . .

Details
2. Mathematisches Modell
Implementierung mit Arrays
Implementierung mit doppelt verketteten Listen

} Übung.

Lesehinweise und Kommentare

- *empty* soll eine leere Datenstruktur erzeugen.
- *pushFront*(x) fügt x vorne an.
- *popFront* nimmt das vorderste Element weg.
- *first* liest das erste Element aus.
- *pushBack*(x) fügt x hinten an.
- *popBack* nimmt das letzte Element weg.
- *last* liest das letzte Element aus.
- Die Übungsaufgaben sind dann, analog zu dem, was wir bei Stacks gemacht haben:
  - (i) Vervollständigen Sie die Signatur.
  - (ii) Stellen Sie ein mathematisches Modell auf.
  - (iii) Beschreiben Sie eine Implementierung mit doppelt verketteten Listen. Was ist der Zeitbedarf pro Operation?
  - (iv) Beschreiben Sie eine Implementierung mit Arrays und Verdoppelungsstrategie. (Achtung: Hier muss man die Queue-Variante mit einem zirkulären Array benutzen.) Analysieren Sie die Rechenzeit für N Operationen.

PAUSE

Kommentar

- Ein neuer Datentyp, geliehen aus der Mathematik: „Endliche Mengen“.
- Wie üblich, gibt es eine Grundmenge D, die endlich oder unendlich sein kann. Denken Sie an natürliche oder ganze Zahlen, an Strings beliebiger Länge, aber auch an die Menge aller 64-Bit-Wörter.
- Anhand von Mengen diskutieren wir Techniken, die später für den sehr wichtigen allgemeineren Datentyp „Wörterbuch“ (alias „assoziatives Array“) verwendet werden.
- Zunächst gibt es nur sehr einfache Operationen. Man kann eine Menge S erzeugen (die als leere Menge initialisiert wird), man kann ein Element $x \in U$ zu S hinzufügen, man kann ein Element aus S streichen, und man kann ein beliebiges $x \in U$ darauf testen, ob es Element von S ist oder nicht.
- Kompliziertere Situationen folgen später.
- Durch eine Folge solcher Operationen können nur endliche Mengen S entstehen.

2.3 Einfache Datenstrukturen für Mengen

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

Initialisierung: $S \leftarrow \emptyset$; // anfangs ist S leer

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

Initialisierung:	$S \leftarrow \emptyset$;	// anfangs ist S leer
Suche:	$x \in S$?;	// für $x \in U$ teste, ob x Element von S ist

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

Initialisierung:	$S \leftarrow \emptyset$;	// anfangs ist S leer
Suche:	$x \in S$?;	// für $x \in U$ teste, ob x Element von S ist
Hinzufügen:	$S \leftarrow S \cup \{x\}$;	// füge $x \in U$ zu S hinzu

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

Initialisierung:	$S \leftarrow \emptyset$;	// anfangs ist S leer
Suche:	$x \in S$?;	// für $x \in U$ teste, ob x Element von S ist
Hinzufügen:	$S \leftarrow S \cup \{x\}$;	// füge $x \in U$ zu S hinzu
Löschen:	$S \leftarrow S - \{x\}$.	// entferne $x \in U$ aus S

Lesehinweise und Kommentare

- Einige Details der Beschreibung sind unklar.
- Was passiert beim Hinzufügen, wenn x schon in S ist?
- Was passiert beim Löschen, wenn x gar kein Element von S ist?
- Wir benutzen unsere Spezifikationsmethode mit einem mathematischen Modell, um diese Dinge ohne jeden Zweifel zu klären.
- Damit hätte man eine perfekte Vertragsgrundlage für die Herstellung und den Verkauf einer Datenstruktur, die den Datentyp „Menge über D“ implementiert!

- Die Signatur kann man aus der informalen Beschreibung praktisch direkt ablesen.
- Mengen heißten auf englisch „sets“.
- „x ist Element von S“ heißt auf englisch „x is a member of S“ oder „x is an element of S“.
- Unser Datentyp heißt „*dynamische* Menge“, weil sich die Menge verändern kann.
- Gegensatz wäre eine „statische Menge“, bei der es nur die Initialisierung auf eine gegebene Menge und die *member*-Operation gibt.

Datentyp Dynamische Menge: DynSet

Datentyp Dynamische Menge: DynSet

1. Signatur:

Sorten:	*Elements*
	Sets
	Boolean
Operationen:	$empty: \rightarrow Sets$
	$insert: Sets \times Elements \rightarrow Sets$
	$delete: Sets \times Elements \rightarrow Sets$
	$member: Sets \times Elements \rightarrow Boolean$

Datentyp Dynamische Menge: DynSet

2. Mathematisches Modell:

Sorten: *Elements*: (nichtleere) Menge D **(Parameter)**

2. Mathematisches Modell:

Sorten:	*Elements*:	(nichtleere) Menge D **(Parameter)**
	Sets:	$\boldsymbol{\mathcal{P}^{<\infty}(D)} = \{S \subseteq D \mid S \text{ endlich}\}$

2. Mathematisches Modell:

Sorten:	*Elements*:	(nichtleere) Menge D **(Parameter)**
	Sets:	$\mathcal{P}^{<\infty}(D) = \{S \subseteq D \mid S \text{ endlich}\}$
	Boolean:	$\{true, false\}$

Datentyp Dynamische Menge: DynSet

2. Mathematisches Modell:

Sorten:

Elements: (nichtleere) Menge D **(Parameter)**

Sets: $\boldsymbol{\mathcal{P}^{<\infty}(D)} = \{S \subseteq D \mid S \text{ endlich}\}$

Boolean: $\{true, false\}$

Operationen:

$empty() := \emptyset$

$insert(S, x) := S \cup \{x\}$

$delete(S, x) := S - \{x\}$

Datentyp Dynamische Menge: DynSet

2. Mathematisches Modell:

Sorten:

Elements: (nichtleere) Menge D **(Parameter)**

Sets: $\mathcal{P}^{<\infty}(D) = \{S \subseteq D \mid S \text{ endlich}\}$

Boolean: $\{true, false\}$

Operationen:

$empty() := \emptyset$

$insert(S, x) := S \cup \{x\}$

$delete(S, x) := S - \{x\}$

$$member(S, x) := \begin{cases} true, & \text{falls } x \in S \\ false, & \text{falls } x \notin S \end{cases}$$

Lesehinweise und Kommentare

- D ist ein Parameter, wie bei Stacks und Queues.
- $\mathcal{P}(D)$ wäre die volle Potenzmenge von D, also die Menge aller Teilmengen von D. Mit der Einschränkung „$\mathcal{P}^{<\infty}$“ beschränken wir uns auf die endlichen Teilmengen.
- Bei Mengen werden wiederholte Nennungen von Elementen ignoriert.
- Wenn wir also $\mathit{insert}(S, x) := S \cup \{x\}$ schreiben, dann hat man im Fall, wo x schon ein Element von S ist, nachher dieselbe Menge wie vorher.
- Das heißt: Die Spezifikation erklärt, was beim erneuten Einfügen eines Elements von x passiert, nämlich nichts.
- Es ist natürlich nicht verboten, dass eine Implementierung in so einem Fall eine Warnung ausgibt. Aber die Operation ist legal und hat ein eindeutiges Ergebnis.
- Das Zeichen „$-$“ steht in dieser Vorlesung (auch) für die Mengendifferenz: $A - B = \{x \in A \mid x \notin B\}$. (In der Mathematik schreibt man dafür oft auch $A \setminus B$.)
- Die Festlegung „$\mathit{delete}(S, x) := S - \{x\}$“ macht eindeutig klar, was passiert, wenn man die Operation *delete* mit einem $x \notin S$ ausführt, nämlich nichts.
- zu *empty* und *member* ist nichts weiter zu sagen.

Datentyp Dynamische Menge: DynSet

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

(C) Einfach verkettete Liste oder Array, **aufsteigend sortiert**

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

(C) Einfach verkettete Liste oder Array, **aufsteigend sortiert**

Nur möglich, wenn es auf D eine totale Ordnung $<$ gibt.

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

(C) Einfach verkettete Liste oder Array, **aufsteigend sortiert**

Nur möglich, wenn es auf D eine totale Ordnung $<$ gibt.

Später:

(D) Suchbäume

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

(C) Einfach verkettete Liste oder Array, **aufsteigend sortiert**

Nur möglich, wenn es auf D eine totale Ordnung $<$ gibt.

Später:

(D) Suchbäume

(E) Hashtabellen

Lesehinweise und Kommentare

- Den Datentyp „dynamische Menge“ kann man auf vielfältige Weise implementieren.
- Eine naheliegende Möglichkeit ist es, die Elemente der Menge einfach in einer **linearen Liste** zu speichern. Hierzu machen wir nur Bemerkungen. Bitte nicht zu sehr ablenken lassen: hier gibt es keinerlei ernsthafte Schwierigkeiten.
- Es genügt eine einfach verkettete Liste ([AuP], Kap. 10 oder in [Saake/Sattler], Kap. 13.2.).
- Leichte Varianten ergeben sich dadurch, dass man es (A) zulässt, dass ein Element von S in der Liste mehrfach vorkommt oder (B) dies verbietet.
- Zu (A): Ist es denn überhaupt erlaubt, dass zum Beispiel die Menge $\{2, 3, 5, 7, 11\}$ durch die Liste mit Einträgen $(5, 7, 11, 2, 5, 3, 7, 5, 2)$ dargestellt wird? Natürlich! Die Implementierung kann tun, was sie will, wenn nur die richtige Mengen-Funktionalität sichergestellt ist.
- Kurz überlegt: *empty* erzeugt einfach eine leere Liste (Kosten $O(1)$). *member*(S, x) wird die Liste durchlaufen und nach einem Eintrag x suchen. Kosten: $O(h)$, wobei $h = O(\#(\text{bisherige Einfügungen}))$ die aktuelle Länge der Liste ist.
- Bei *insert*(S, x) können wir x einfach vorne in die Liste einhängen, z. B. liefert *insert*$(S, 3)$ für die Liste $(5, 7, 11, 2, 5, 3, 7, 5, 2)$ das Ergebnis $(3, 5, 7, 11, 2, 5, 3, 7, 5, 2)$. Kosten: billige $O(1)$.

Lesehinweise und Kommentare

- Bei *delete*(S, x) muss aus der Liste *jedes* Vorkommen von x gelöscht werden, z. B. liefert *delete*$(S, 5)$ für die Liste $(5, 7, 11, 2, 5, 3, 7, 5, 2)$ das Ergebnis $(7, 11, 2, 3, 7, 2)$. Kosten: $O(h) = O(\#(\text{bisherige Einfügungen}))$, wie bei *member*.
- Eine (gar nicht so verrückte) Variante wäre, dass die Operation *delete*(S, x) vorne in die Liste einen Eintrag $(x, \text{deleted})$ einfügt, die Operation *insert*(S, x) nach wie vor den Eintrag x. Dann hätte auch *delete* Kosten $O(1)$. Frage: Wie müsste nun die Operation *member*(S, x) implementiert werden? Kosten: $O(h)$ für aktuelle Listenlänge $h = O(\#\text{bisherige Einfügungen})$.

Lesehinweise und Kommentare

- Anstelle einer Liste kann man auch ein **Array** $\texttt{A}[1..m]$ verwenden.
- Im Prinzip kann man das dann so bedienen wir einen Stack oder ein dynamisches Array (Array $\texttt{A}[1..m]$ plus Pegelvariable $\texttt{p}$, siehe Abschnitt 2.1.
- *empty*(): wie bei Stacks. *insert*(S, x): wie *push* beim Stack. *member*(S, x): Man durchsucht das Teilarray $\texttt{A}[1..\texttt{p}]$.
- *delete*(S, x): Das erfordert Durchsuchen des Arrays und Entfernen aller Kopien von x. Wenn man ganz naiv die Einträge einfach löscht, entstehen Lücken. Was tun?
- Antwort: Es kommt ja auf die Reihenfolge nicht an! Daher: Wenn man $\texttt{A}[i]$ löschen will, setzt man $\texttt{A}[i] \leftarrow \texttt{A}[\texttt{p}]$ und setzt dann den Pegel $\texttt{p}$ um 1 herunter. Dadurch werden Lücken vermieden.
- Wenn man nicht weiß, wie groß die Mengen werden: Verdoppelungsstrategie!
- Die Rechenzeiten/Kosten entsprechen denen bei linearen Listen, bis auf den Unterschied, dass *insert* nur amortisiert, also gemittelt, konstante Kosten hat.

Lesehinweise und Kommentare

- Nun zu Variante (B): Speichern in Listen und in Arrays **ohne Wiederholung**, d. h. dass jedes Element von S genau einmal in der Liste/im Array vorkommt.
- Vorteil: Der von der Datenstruktur benötigte Platz wird nur durch die Größe $|S|$ der Menge bestimmt, nicht durch die Anzahl der Operationen.
- Bei beiden Varianten: *empty*, *member*, *delete* werden implementiert wie bei (A). Bei *insert*(S, x) muss man zuerst in der Liste/im Array nach x suchen. Kosten: $O(|S|)$.
- Wie sinnvoll ist es, Listen oder Arrays zu benutzen, wenn doch die Rechenzeit so hoch ist?
- **Hinweis:** Für sehr kleine Mengen (vielleicht bis zu $h = 20$) ist eine Listen- oder Arrayimplementierung durchaus sinnvoll und effizient.
- In C++: benutze Klasse `std::vector` für Arrays.
- Solche sehr kurzen Listen treten als Teilstrukturen auch von riesigen Datenstrukturen auf, z. B. bei Hashtabellen, s. Kap. 5.

Zusammenfassung

(A) Einfach verkettete Liste oder Array **mit Wiederholung**:

Initialisierung, Einfügen in Zeit $O(1)$.

Platzbedarf, Länge der Liste: $h = O(\#(\text{bisherige Einfügungen}))$.

Suchen, Löschen in Zeit $\Theta(h)$.

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

$n = |S| = \#(\text{Einträge})$.

Platzbedarf: $O(n)$.

Einfügen, Suchen, Löschen in Zeit $\Theta(n)$.

Zusammenfassung

(A) Einfach verkettete Liste oder Array **mit Wiederholung**:

Initialisierung, Einfügen in Zeit $O(1)$.

Platzbedarf, Länge der Liste: $h = O(\#(\text{bisherige Einfügungen}))$.

Suchen, Löschen in Zeit $\Theta(h)$.

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

$n = |S| = \#(\text{Einträge})$.

Platzbedarf: $O(n)$.

Einfügen, Suchen, Löschen in Zeit $\Theta(n)$.

Achtung: Wegen der hohen Suchzeiten sind lineare Listen und Arrays als grundsätzlicher Ansatz zur Implementierung des Mengen-Datentyps DynSet **nicht geeignet** (außer es handelt sich um sehr kleine Strukturen).

Lesehinweise und Kommentare

- Sehr oft gibt es für die Elemente der Menge D eine totale Ordnung $<$.
- In diesem Fall kann man die Suche (*member*-Operation) dramatisch beschleunigen, indem man die Elemente in aufsteigender Reihenfolge in einem Array speichert und einen passenden Algorithmus, nämlich **Binäre Suche**, benutzt.
- **Merke:** Effiziente Datenstrukturen haben oft clevere Algorithmen als Methoden eingebaut.
- Umgekehrt gilt natürlich: Um effiziente, schnelle Algorithmen für Berechnungsprobleme zu bauen, sind oft clevere Datenstrukturen sehr nützlich.
- Wir benutzen die binäre Suche als weiteres Beispiel (nach Insertionsort), um einen Korrektheitsbeweis und eine Zeitanalyse exakt durchzuführen, dieses Mal am Beispiel eines **rekursiven Algorithmus**.
- Der Algorithmus wurde im Prinzip in [AuP], Kapitel 7, vorgestellt.
- Grob weiß man: Wenn es n Einträge gibt, ist die Rechenzeit $\Theta(n)$.

(C) Array, ohne Wiederholung, **aufsteigend sortiert**:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:	2	5	7	10	17	19	26	50	54	59	67	$*$	$*$	$*$	$*$

„Pegelvariable“ `n` enthält $n = \#(\text{Einträge})$, hier $n = 11$.

(C) Array, ohne Wiederholung, **aufsteigend sortiert**:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:	2	5	7	10	17	19	26	50	54	59	67	$*$	$*$	$*$	$*$

„Pegelvariable“ n enthält $n = \#(\text{Einträge})$, hier $n = 11$.

Nachteil:
Bei *insert*, *delete* ist **Verschieben** vieler Elemente nötig, um die Ordnung zu erhalten und dabei das Entstehen von Lücken zu vermeiden.

Zeitaufwand für diese Operationen: $\Theta(n) = \Theta(|S|)$.

(C) Array, ohne Wiederholung, **aufsteigend sortiert**:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:	2	5	7	10	17	19	26	50	54	59	67	$*$	$*$	$*$	$*$

„Pegelvariable“ n enthält $n = \#(\text{Einträge})$, hier $n = 11$.

Nachteil:
Bei *insert*, *delete* ist **Verschieben** vieler Elemente nötig, um die Ordnung zu erhalten und dabei das Entstehen von Lücken zu vermeiden.

Zeitaufwand für diese Operationen: $\Theta(n) = \Theta(|S|)$.

Vorteil:

member: Kann **binäre Suche** benutzen.

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

0. Falls $b < a$, ist x nicht im Teilarray.

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

0. Falls $b < a$, ist x nicht im Teilarray.

1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

0. Falls $b < a$, ist x nicht im Teilarray.
1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.
2. Falls $a < b$, berechne $m = a + \lfloor (b-a)/2 \rfloor$ // Mitte des Teilarrays

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

0. Falls $b < a$, ist x nicht im Teilarray.

1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.

2. Falls $a < b$, berechne $m = a + \lfloor (b-a)/2 \rfloor$ // Mitte des Teilarrays

3 Fälle:

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

0. Falls $b < a$, ist x nicht im Teilarray.

1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.

2. Falls $a < b$, berechne $m = a + \lfloor (b-a)/2 \rfloor$ // Mitte des Teilarrays

3 Fälle:

$x = \texttt{A}[m]$: x gefunden, Antwort $true$.

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

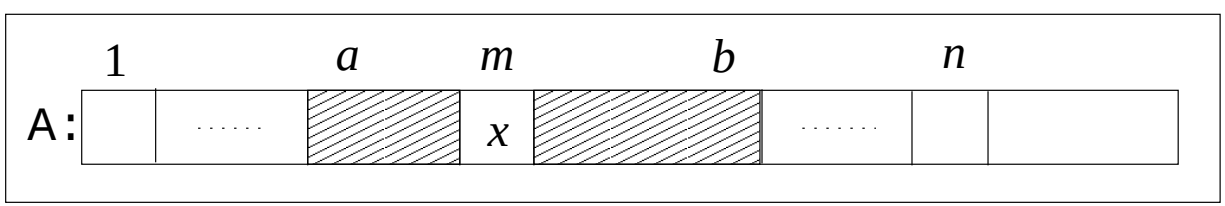
0. Falls $b < a$, ist x nicht im Teilarray.

1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.

2. Falls $a < b$, berechne $m = a + \lfloor (b-a)/2 \rfloor$ // Mitte des Teilarrays

3 Fälle:

$x = \texttt{A}[m]$: x gefunden, Antwort $true$.

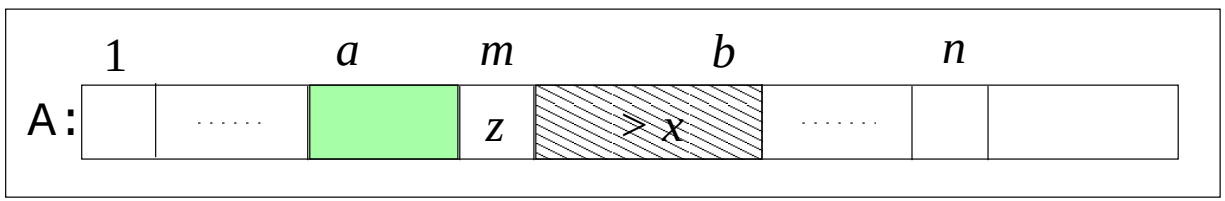


Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.

Binäre Suche, rekursiv

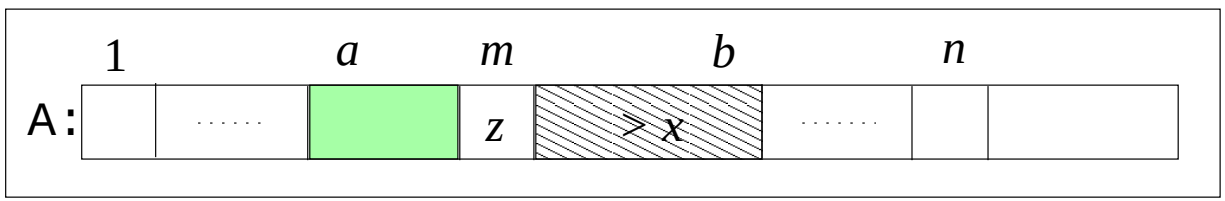
$x <$ `A[`m`]`: Wende **(rekursiv)** binäre Suche auf `A[`a`..`$m-1$`]` an.



$x < z$

Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.

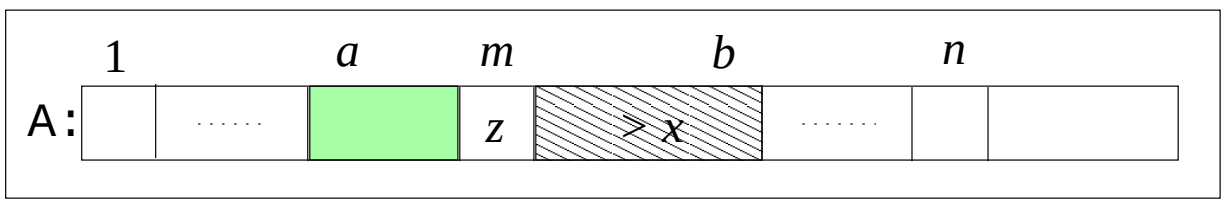


$x < z$

$x > \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[m+1\,..\,b]$ an.

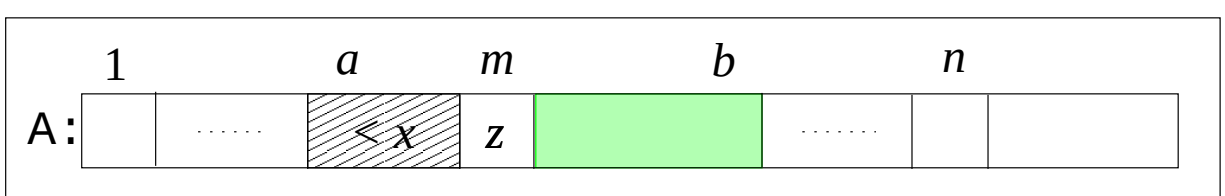
Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.



$x < z$

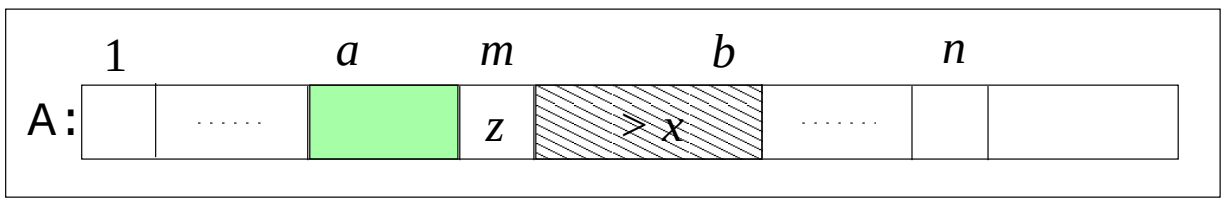
$x > \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[m+1\,..\,b]$ an.



$z < x$

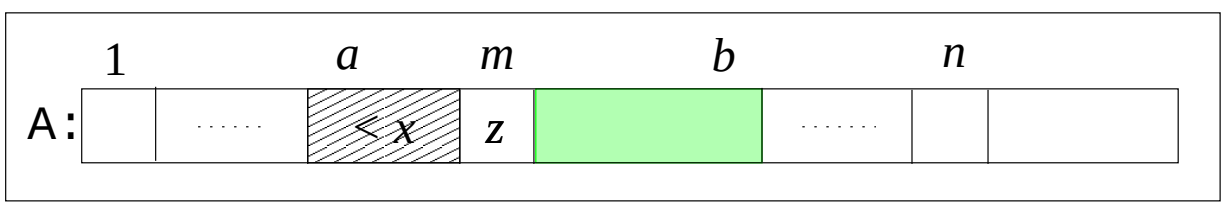
Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.



$x < z$

$x > \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[m+1\,..\,b]$ an.



$z < x$

Um x im Gesamtarray $\texttt{A}[1\,..\,n]$ zu suchen, wenn $n \geq 1$:
Binäre Suche nach x in $\texttt{A}[1\,..\,n]$.

Lesehinweise und Kommentare

- Die Idee der binären Suche ist aus [AuP] bekannt.
- Sie ist immer anwendbar, wenn D eine Totalordnung $<$ hat.
- Wenn a und b beide gerade oder beide ungerade sind, ist m genau $(a+b)/2$. Wenn $a+b$ ungerade ist, ist $m = \lfloor (a+b)/2 \rfloor$, z. B. $4 + \lfloor (7-4)/2 \rfloor = 4+1 = 5 = \lfloor (4+7)/2 \rfloor$.
- Wenn Sie sich fragen, warum hier $m = a + \lfloor (b-a)/2 \rfloor$ gerechnet wird und nicht (wie man erwarten würde, und arithmetisch gleichwertig) $m = \lfloor (a+b)/2 \rfloor$: In Programmiersprachen gibt es MAXINT, die größte darstellbare natürliche Zahl. Falls nun jemand in einem Array mit Indexbereich $1..n$ binäre Suche ausführt, wo $n > \frac{1}{2}$MAXINT ist (nicht wahrscheinlich, aber im Prinzip möglich), dann könnte die Berechnung von $(a+b)/2$ beim Zwischenergebnis $a+b$ zu einem arithmetischen Überlauf führen, wo $a + \lfloor (b-a)/2 \rfloor$ noch problemlos die richtige Zahl berechnet.
- Die Ideen liefern unmittelbar eine rekursive Prozedur in Pseudocode.
- Zur Fallunterscheidung in Zeilen (4)–(6): Viele Programmiersprachen haben für viele angeordnete Mengen D (Zahlen, Strings) einen so genannten **3-Wege-Vergleich**, in dem mit einer Operation festgestellt werden kann, ob „$<$“, „$=$“ oder „$>$“ gilt. Natürlich wird man beim Programmieren eine solche Operation benutzen, wenn sie zur Verfügung steht.

Gegeben: Array `A[1..`n`]`, x: `elements`;

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b)

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$
(1) **if** $b < a$ **then return** $false$;

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$
(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$
(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;
(5) **if** $x <$ `A[m]` **then**

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;
(5) **if** $x <$ `A[m]` **then return RecBinSearch**$(a,$`m`$ - 1)$;

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;
(5) **if** $x <$ `A[m]` **then return RecBinSearch**$(a,$`m`$ - 1)$;
(6) **if** $x >$ `A[m]` **then**

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;
(5) **if** $x <$ `A[m]` **then return RecBinSearch**$(a$,`m` $- 1)$;
(6) **if** $x >$ `A[m]` **then return RecBinSearch**$($`m` $+ 1, b)$.

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

```
(1)   if b < a then return false;
(2)   if a = b then return (x = A[a]);     //boolescher Wert
(3)   m ← a + (b − a) div 2;               //ganzzahlige Division
(4)   if x = A[m] then return true;
(5)   if x < A[m] then return RecBinSearch(a,m − 1);
(6)   if x > A[m] then return RecBinSearch(m + 1,b).
```

Algorithmus Binäre Suche (rekursiv)

Gegeben: Array `A[1..`n`]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

```
(1)   if b < a then return false;
(2)   if a = b then return (x = A[a]);     //boolescher Wert
(3)   m ← a + (b − a) div 2;   //ganzzahlige Division
(4)   if x = A[m] then return true;
(5)   if x < A[m] then return RecBinSearch(a,m − 1);
(6)   if x > A[m] then return RecBinSearch(m + 1,b).
```

Algorithmus Binäre Suche (rekursiv)
Eingabe: Array `A[1..`n`]`; x: `elements`;
mit `A[1]` $< \cdots <$ `A[`n`]`;

```
(1)   return RecBinSearch(1, n).
```

Lesehinweise und Kommentare

- Wir beweisen sorgfältig die **Korrektheit** dieses Verfahrens.
- Obgleich es wohlbekannt und die Idee einfach zu verstehen ist, ist es auch sehr einfach, in eine solchen Implementierung einen Fehler einzubauen.
- Korrektheitsbeweise können helfen, Fehler zu entdecken – sogar ohne Tests.
- Zitat: „Beware of bugs in the above code; I have only proved it correct, not tried it.“ – Donald E. Knuth, Turing Award 1974, schrieb/schreibt „The Art of Computer Programming“, entwickelte TEX, die Basis von LATEX, dem Textsatzprogramm hinter diesen Folien.

Proposition 2.3.1

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray `A[`a`..`b`]` von `A[1..`n`]` sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b)

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray `A[`a`..`b`]` von `A[1..`n`]` sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$,

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt,

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1\,..\,n]$ und x liefert $true$, falls x in $\texttt{A}[1\,..\,n]$ vorkommt,

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1\,..\,n]$ und x liefert $true$, falls x in $\texttt{A}[1\,..\,n]$ vorkommt, $false$, falls nicht.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a..b]$ von $\texttt{A}[1..n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a..b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1..n]$ und x liefert $true$, falls x in $\texttt{A}[1..n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1\,..\,n]$ und x liefert $true$, falls x in $\texttt{A}[1\,..\,n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Wir benutzen **verallgemeinerte Induktion** über i.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1\,..\,n]$ und x liefert $true$, falls x in $\texttt{A}[1\,..\,n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Wir benutzen **verallgemeinerte Induktion** über i.

I.A.: Fall $i = 0$. D. h. $b < a$, d. h. das Teilarray $\texttt{A}[a\,..\,b]$ ist leer.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a\,..\,b]$ von $\texttt{A}[1\,..\,n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a\,..\,b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1\,..\,n]$ und x liefert $true$, falls x in $\texttt{A}[1\,..\,n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Wir benutzen **verallgemeinerte Induktion** über i.

I.A.: Fall $i = 0$. D. h. $b < a$, d. h. das Teilarray $\texttt{A}[a\,..\,b]$ ist leer.
Die Antwort $false$, die in (Zeile (1)) ausgegeben wird, ist also korrekt.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a..b]$ von $\texttt{A}[1..n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a..b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1..n]$ und x liefert $true$, falls x in $\texttt{A}[1..n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Wir benutzen **verallgemeinerte Induktion** über i.

I.A.: Fall $i = 0$. D. h. $b < a$, d. h. das Teilarray $\texttt{A}[a..b]$ ist leer.
Die Antwort $false$, die in (Zeile (1)) ausgegeben wird, ist also korrekt.
Fall $i = 1$: Dann ist $1 \leq a = b \leq n$, und die Ausgabe $(x = \texttt{A}[a])$, die in (Zeile (2)) ausgegeben wird, ist korrekt.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x <$ `A[`m`]`.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x <$ `A[`m`]`. – Analog zu Fall 3.

Fall 3: $x >$ `A[`m`]`. – Beobachte:

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x <$ `A[`m`]`. – Analog zu Fall 3.

Fall 3: $x >$ `A[`m`]`. – Beobachte: x kommt auf keinen Fall in `A[`$a..m$`]` vor, denn für $a \leq i \leq m$ gilt `A[`i`]` $\leq$ `A[`m`]` $< x$, weil das Array sortiert ist.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b - a)/2 \rfloor = \lfloor (a + b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a + b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x <$ `A[`m`]`. – Analog zu Fall 3.

Fall 3: $x >$ `A[`m`]`. – Beobachte: x kommt auf keinen Fall in `A[`$a..m$`]` vor, denn für $a \leq i \leq m$ gilt `A[`i`]` $\leq$ `A[`m`]` $< x$, weil das Array sortiert ist.

Also kommt x in `A[`$a..b$`]` vor genau dann wenn x in `A[`$m + 1..b$`]` vorkommt.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b-a)/2 \rfloor = \lfloor (a+b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a+b)/2 < b$.)

Fall 1: $x = \texttt{A}[m]$. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x < \texttt{A}[m]$. – Analog zu Fall 3.

Fall 3: $x > \texttt{A}[m]$. – Beobachte: x kommt auf keinen Fall in $\texttt{A}[a\,..\,m]$ vor, denn für $a \leq i \leq m$ gilt $\texttt{A}[i] \leq \texttt{A}[m] < x$, weil das Array sortiert ist.

Also kommt x in $\texttt{A}[a\,..\,b]$ vor genau dann wenn x in $\texttt{A}[m+1\,..\,b]$ vorkommt.

Die Länge dieses Abschnittes ist $b - m \overset{(*)}{\leq} b - a < i$. Nach I.V. liefert der rekursive Aufruf **RecBinSearch**$(m+1, b)$ in Zeile (5) das korrekte Resultat für diesen Abschnitt, also auch insgesamt. □

Lesehinweise und Kommentare

- Dieser Beweis ist ein erstes Beispiel für die häufige Situation, dass die Korrektheit von rekursiven Algorithmen durch *verallgemeinerte Induktion* bewiesen wird, wo also die Induktionsvoraussetzung im Induktionsschritt für i ist, dass die Induktionsbehauptung für alle $j < i$ gilt (nicht nur für $j = i - 1$ wie bei der gewöhnlichen Induktion).
- Vielleicht sagt sich der eine oder die andere beim Lesen: So viel „Formelgeklingel“ für so eine Banalität! Man „sieht“ doch, dass der Algorithmus korrekt ist!
- Jedoch macht man beim Programmieren allzu leicht Fehler, weil man sich die Randfälle (hier: Arrays mit Länge 0 und mit Länge 1, die am Ende der Rekursion auftreten) nicht sorgfältig genug anschaut. Hier ist es hilfreich, alle kleinen Ecken des Programms und alle Fälle durchzugehen.
- Es ist ebenso wichtig, sich zu vergewissern, dass für Zugriffe verwendete Indizes nie außerhalb der Arraygrenzen liegen, und dass die Arraylänge in den rekursiven Aufrufen stets strikt sinkt, so dass es keine Endlosschleife geben kann.

Zur **Rechenzeit** von Binärer Suche:

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\texttt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\texttt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Jeder Aufruf von **RecBinSearch** verursacht Kosten $\leq C$ für eine Konstante C, wenn man die Kosten der hiervon ausgelösten rekursiven Aufrufe nicht zählt.

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\texttt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Jeder Aufruf von **RecBinSearch** verursacht Kosten $\leq C$ für eine Konstante C, wenn man die Kosten der hiervon ausgelösten rekursiven Aufrufe nicht zählt.

Gesucht also, für alle i:

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\mathtt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Jeder Aufruf von **RecBinSearch** verursacht Kosten $\leq C$ für eine Konstante C, wenn man die Kosten der hiervon ausgelösten rekursiven Aufrufe nicht zählt.

Gesucht also, für alle i:

$r(i) :=$ maximale Anzahl der Aufrufe (inklusive des ersten Aufrufs), wenn $b - a + 1 = i$.

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\mathtt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Jeder Aufruf von **RecBinSearch** verursacht Kosten $\leq C$ für eine Konstante C, wenn man die Kosten der hiervon ausgelösten rekursiven Aufrufe nicht zählt.

Gesucht also, für alle i:

$r(i) :=$ maximale Anzahl der Aufrufe (inklusive des ersten Aufrufs), wenn $b - a + 1 = i$.

Klar nach Zeilen (1), (2) von **RecBinSearch**: $r(0) = r(1) = 1$.

Lesehinweise und Kommentare

- Wir wenden uns der Rechenzeitanalyse von binärer Suche zu.
- Aus [AuP] weiß man, dass das Ergebnis $\Theta(\log n)$ sein muss.
- Wir nehmen es genau.
- Der Ansatz ist sehr typisch für die Analyse der Rechenzeit von rekursiven Prozeduren. Wenn man keine Ahnung hat, was die Rechenzeit ist, gibt man ihr einfach erst einmal einen Namen, hier $T_{RBS}(i)$, damit man über sie reden kann, auch wenn man sie nicht kennt.
- Diese Zahl hängt nur von der Arraylänge i ab. Abgesehen von i muss man den schlimmstmöglichen Input betrachten, daher die Maximierung über alle Arrays und alle Einträge.
- Konstante Faktoren wollten wir in unseren Analysen ignorieren. Daher sagen wir, dass die Ausführung eines Aufrufs von BinRecSearch konstante Zeit benötigt, wenn man den Aufwand für die ausgelösten rekursiven Aufrufe ignoriert. Dadurch reduziert sich das Problem darauf, die rekursiven Aufrufe zu zählen. Dies ist eine grandiose Vereinfachung (ohne unnötig an Präzision zu verlieren)!

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.

$\Rightarrow$ **„Rekurrenzungleichung“:**

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.

$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.
$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Behauptung: $r(i) \leq 1 + \log i$, für $i \geq 1$. (Wissen sowieso: $r(0) = 1$.)

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.
$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Behauptung: $r(i) \leq 1 + \log i$, für $i \geq 1$. (Wissen sowieso: $r(0) = 1$.)

Beweis durch verallgemeinerte Induktion über $i \geq 1$.

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.
$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Behauptung: $r(i) \leq 1 + \log i$, für $i \geq 1$. (Wissen sowieso: $r(0) = 1$.)

Beweis durch verallgemeinerte Induktion über $i \geq 1$.

I.A.: Beh. stimmt für $i = 1$, weil $r(1) = 1$ und $\log 1 = 0$.

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.
$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Behauptung: $r(i) \leq 1 + \log i$, für $i \geq 1$. (Wissen sowieso: $r(0) = 1$.)

Beweis durch verallgemeinerte Induktion über $i \geq 1$.

I.A.: Beh. stimmt für $i = 1$, weil $r(1) = 1$ und $\log 1 = 0$.

Nun sei $i \geq 2$.

I.V.: Beh. stimmt für alle $j \in \{1, \ldots, i-1\}$.

I.-Schritt: Die Rekurrenzungleichung sagt:

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$r(i) \leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\}$$

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) \end{aligned}$$

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

Das ist die Induktionsbehauptung. □

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

Das ist die Induktionsbehauptung. □

Satz 2.3.3

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

Das ist die Induktionsbehauptung. □

Satz 2.3.3

Der Algorithmus **Binäre Suche** benötigt auf Eingabe `A[1..n]` höchstens $1 + \lfloor \log n \rfloor$ rekursive Aufrufe und Kosten (Rechenzeit) $O(\log n)$.

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

Das ist die Induktionsbehauptung. □

Satz 2.3.3

Der Algorithmus **Binäre Suche** benötigt auf Eingabe $\mathtt{A}[1\,..\,n]$ höchstens $1 + \lfloor \log n \rfloor$ rekursive Aufrufe und Kosten (Rechenzeit) $O(\log n)$.

Beispiel: Binäre Suche in Array der Größe $10\,000\,000$ erfordert nicht mehr als $1 + \lfloor \log_2(10^7) \rfloor = 24$ rekursive Aufrufe. (Extrem wenig im Vergleich zu linearer Zeit in Fällen (A) und (B)!)

Lesehinweise und Kommentare

- Wir werden später viel mit rekursiven Prozeduren und ihren Korrektheitsbeweisen und Zeitanalysen zu tun haben, insbesondere im Kapitel „Divide-and-Conquer-Algorithmen“.
- Der Ansatz mit „Rekurrenzungleichungen“ kommt sehr häufig vor.
- Man gibt zuerst der unbekannten Größe (hier $T_{RBS}(i)$ bzw. $r(i)$) einen Namen . . .
- . . . und stellt dann eine Beziehung zwischen dem Wert für i und irgendwelchen Werten für $j < i$ her.
- Die resultierende Ungleichung analysiert man dann mit mathematischen Mitteln.
- Unser Ansatz hier: Wir raten die Lösung und beweisen sie mittels **verallgemeinerter Induktion**, immer das Mittel der Wahl bei rekursiven Prozeduren.
- Im Beweis: $\lceil (i-1)/2 \rceil \leq i/2$ gilt, weil für gerades i Gleichheit herrscht und für ungerades i die linke Seite gleich $(i-1)/2$ ist.
- Damit hat man $r(\lceil (i-1)/2 \rceil) \leq 1 + \log(\lceil (i-1)/2 \rceil) \leq 1 + \log(i/2)$, mit der I.V. und weil die $\log$-Funktion monoton ist.
- Weil die Rundenzahl immer ganzzahlig ist, darf man die Abschätzung auf $1 + \lfloor \log n \rfloor$ abrunden.
- **Merke:** Logarithmische Rechenzeiten sind drastisch kleiner als lineare!

PAUSE

Iterative binäre Suche in schwach geordneten Arrays

Beispiel: Array `A[1..11]` schwach aufsteigend sortiert, Suche nach x.
Ausgabe $i(x)$: „Position“ von Suchschlüssel x.

	1	2	3	4	5	6	7	8	9	10	11=n
A:	4	4	7	7	7	9	9	10	12	15	15

$i\,(7) = 3$ erstes Vorkommen von x

$i\,(8) = 6$ Position des ersten Eintrags größer als x

$i\,(20) = 12$ x ist größer als alle Arrayeinträge

Allgemein:

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen):

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array `A[1..n]`“ (häufig in Anwendungen): `A[1..n]` enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert,

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen): $\texttt{A}[1\,..\,n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung.

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen): $\texttt{A}[1\,..\,n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen): $\texttt{A}[1\,..\,n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher.

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen): $\texttt{A}[1\,..\,n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher. Zudem definieren wir:

$$i(x) := \min(\{i \mid 1 \leq i \leq n, x \leq \texttt{A}[i]\} \cup \{n+1\}).$$

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1\,..\,n]$“ (häufig in Anwendungen): $\texttt{A}[1\,..\,n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher. Zudem definieren wir:

$$i(x) := \min(\{i \mid 1 \leq i \leq n, x \leq \texttt{A}[i]\} \cup \{n+1\}).$$

Wenn x im Array vorkommt, ist dies die Position des ersten (am weitesten links stehenden) Vorkommens von x.

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1..n]$“ (häufig in Anwendungen): $\texttt{A}[1..n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher. Zudem definieren wir:

$$i(x) := \min(\{i \mid 1 \leq i \leq n, x \leq \texttt{A}[i]\} \cup \{n+1\}).$$

Wenn x im Array vorkommt, ist dies die Position des ersten (am weitesten links stehenden) Vorkommens von x.
Wenn x nicht vorkommt, ist es die Position des ersten Eintrags, der größer als x ist.

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1..n]$“ (häufig in Anwendungen): $\texttt{A}[1..n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher. Zudem definieren wir:

$$i(x) := \min(\{i \mid 1 \leq i \leq n, x \leq \texttt{A}[i]\} \cup \{n+1\}).$$

Wenn x im Array vorkommt, ist dies die Position des ersten (am weitesten links stehenden) Vorkommens von x.
Wenn x nicht vorkommt, ist es die Position des ersten Eintrags, der größer als x ist.
Wenn kein Eintrag im Array größer als x ist, ist $i(x) = n + 1$.

Lesehinweise und Kommentare

- Diese Problemvariante passt nicht recht zum Datentyp „Menge“, weil ja bei Mengen das mehrfache Vorkommen von Elementen ausdrücklich verboten ist.
- Manchmal ist das Konzept einer „Multimenge“ nützlich, in dem gerade dieses erlaubt ist. Unsere Arrays stellen also Multimengen dar.
- Außerdem ist die Problemstellung ein Vorgriff auf Situationen, wo der Suchschlüssel nur ein Attribut eines Datensatzes ist. Dann kann natürlich ein Schlüssel im Prinzip auch mehrfach vorkommen.
- Beispiel: Ein (richtiges, gedrucktes) Wörterbuch, in dem Einträge zu gleichen Wörtern mit unterschiedlicher Bedeutung („Hahn“ als Tier und als Wasserhahn) hintereinander stehen.
- Wenn x im Array vorkommt, findet man den Block der Objekte mit Schlüssel x.
- Außerdem: Wenn x nicht vorkommt, findet man die Position im Array, wo es hingehören würde.

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**
(3) `m` $\leftarrow$ `a + (b-a) div 2` ;

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

```
(1)   a ← 1; b ← n;
(2)   while a < b do
(3)     m ← a + (b-a) div 2 ;
(4)     if x ≤ A[m] then b ← m else a ← m + 1;
```

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..`n`]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[`n`]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**
(3) `m` $\leftarrow$ `a + (b-a) div 2` ;
(4) **if** $x \leq$ `A[m]` **then** `b` $\leftarrow$ `m` **else** `a` $\leftarrow$ `m` $+ 1$;
(5) **if** $x >$ `A[a]` **then return** $(\mathit{false}$, `a` $+ 1)$;

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..n]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[n]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**
(3) `m` $\leftarrow$ `a + (b-a) div 2` ;
(4) **if** $x \leq$ `A[m]` **then** `b` $\leftarrow$ `m` **else** `a` $\leftarrow$ `m` $+ 1$;
(5) **if** $x >$ `A[a]` **then return** $(\mathit{false}$, `a` $+ 1)$;
(6) **if** $x =$ `A[a]` **then return** $(\mathit{true}$, `a`$)$

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..n]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[n]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**
(3) `m` $\leftarrow$ `a + (b-a) div 2` ;
(4) **if** $x \leq$ `A[m]` **then** `b` $\leftarrow$ `m` **else** `a` $\leftarrow$ `m` $+ 1$;
(5) **if** $x >$ `A[a]` **then return** $(\mathit{false}$, `a` $+ 1)$;
(6) **if** $x =$ `A[a]` **then return** $(\mathit{true}$, `a`$)$
(7) **else return** $(\mathit{false}$, `a`$)$.

Satz 2.3.4

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Die Anzahl der Durchläufe durch den Rumpf der Schleife ist höchstens $\lceil \log n \rceil$.

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Die Anzahl der Durchläufe durch den Rumpf der Schleife ist höchstens $\lceil \log n \rceil$.

Die Laufzeit ist $\Theta(\log n)$ im schlechtesten und im besten Fall.

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Die Anzahl der Durchläufe durch den Rumpf der Schleife ist höchstens $\lceil \log n \rceil$.

Die Laufzeit ist $\Theta(\log n)$ im schlechtesten und im besten Fall.

Beweis: Übung.

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Die Anzahl der Durchläufe durch den Rumpf der Schleife ist höchstens $\lceil \log n \rceil$.

Die Laufzeit ist $\Theta(\log n)$ im schlechtesten und im besten Fall.

Beweis: Übung.

Vorsicht! Die Korrektheit ist alles andere als offensichtlich. Sorgfältiges Argumentieren ist nötig.

Lesehinweise und Kommentare

- Für viele Algorithmen gibt es eine rekursive Version und eine iterative Version.
- Meist ist die rekursive Version einfacher zu beschreiben und zu analysieren.
- Iterativ gestaltete Programme sind eventuell in der Programmierung kompakter und in der Ausführung schneller, um einen konstanten Faktor, weil der Organisationsaufwand für die Rekursion wegfällt.
- Im Fall der binären Suche ist der Übergang von rekursiver zu iterativer Programmstruktur besonders einfach, denn: In **BinRecSearch** wird das Ergebnis („*true*“ oder „*false*“) des darin ausgelösten rekursiven Aufrufs einfach „nach außen durchgereicht“ und nicht mehr verarbeitet. So eine Programmstruktur nennt man „Endrekursion“ („tail recursion“), und sie lässt sich fast „durch Hinschauen“ in eine iterative Version umwandeln.

Mehrere Mengen: Datentyp DynSets

Mehrere Mengen: Datentyp DynSets

Intuitive Aufgabe:

Mehrere Mengen: Datentyp DynSets

Intuitive Aufgabe: Speichere **mehrere** Mengen $S_1, \ldots, S_r \subseteq D$, so dass für jede einzelne von ihnen die Operationen des Datentyps **Dynamische Menge** ausführbar sind und zudem

Mehrere Mengen: Datentyp DynSets

Intuitive Aufgabe: Speichere **mehrere** Mengen $S_1, \ldots, S_r \subseteq D$, so dass für jede einzelne von ihnen die Operationen des Datentyps **Dynamische Menge** ausführbar sind und zudem

Vereinigung, **Durchschnitt**, **Differenz**, **symmetrische Differenz**, **Leerheitstest**

Mehrere Mengen: Datentyp DynSets

Intuitive Aufgabe: Speichere **mehrere** Mengen $S_1, \ldots, S_r \subseteq D$, so dass für jede einzelne von ihnen die Operationen des Datentyps **Dynamische Menge** ausführbar sind und zudem

Vereinigung, **Durchschnitt**, **Differenz**, **symmetrische Differenz**, **Leerheitstest**

von beliebigen dieser Mengen.

(Vereinigung: $S \cup S'$; Durchschnitt: $S \cap S'$; Differenz: $S - S'$;
symmetrische Differenz: $S \oplus S' = S \triangle S' = (S - S') \cup (S' - S)$; Leerheitstest: Ist $S = \emptyset$?)

Lesehinweise und Kommentare

- Unser Datentyp Dynset wird „aufgebohrt“. Bisher kann eine Menge verwaltet werden, nun sind es mehrere, beliebig viele.
- Es kommen typische Mengenoperationen hinzu.
- Wir erhalten eine Spezifikation durch Erweiterung des Datentyps Dynset. Das entspricht offensichtlich einem Standardvorgehen bei der objektorientierten Programmierung („Vererbung“, „abgeleitete Klassen“).
- Für den Leerheitstest wird der Datentyp *Boolean* benötigt.
- Die folgende Spezifikation übernimmt für das mathematische Modell einfach die Operationen, wie man sie aus der Mengenlehre kennt.

Mehrere Mengen: Datentyp DynSets

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

$union : Sets \times Sets \rightarrow Sets$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

$union : Sets \times Sets \rightarrow Sets$
$intersection : Sets \times Sets \rightarrow Sets$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

$union : Sets \times Sets \rightarrow Sets$
$intersection : Sets \times Sets \rightarrow Sets$
$diff : Sets \times Sets \rightarrow Sets$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

$union : Sets \times Sets \rightarrow Sets$
$intersection : Sets \times Sets \rightarrow Sets$
$diff : Sets \times Sets \rightarrow Sets$
$symdiff : Sets \times Sets \rightarrow Sets$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

$union : Sets \times Sets \rightarrow Sets$
$intersection : Sets \times Sets \rightarrow Sets$
$diff : Sets \times Sets \rightarrow Sets$
$symdiff : Sets \times Sets \rightarrow Sets$
$isempty : Sets \rightarrow Boolean$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$
intersection$(S_1, S_2) := S_1 \cap S_2$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$
intersection$(S_1, S_2) := S_1 \cap S_2$
diff$(S_1, S_2) := S_1 - S_2$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$
intersection$(S_1, S_2) := S_1 \cap S_2$
diff$(S_1, S_2) := S_1 - S_2$
symdiff$(S_1, S_2) := (S_1 \cup S_2) - (S_1 \cap S_2)$

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$
intersection$(S_1, S_2) := S_1 \cap S_2$
diff$(S_1, S_2) := S_1 - S_2$
symdiff$(S_1, S_2) := (S_1 \cup S_2) - (S_1 \cap S_2)$

$$\textit{isempty}(S) := \begin{cases} true, & \text{falls } S = \emptyset \\ false, & \text{sonst.} \end{cases}$$

Mehrere Mengen: Datentyp DynSets

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Gleichheitstest: Ist $S_1 = S_2$?

$\mathit{equal}(S_1, S_2) = \mathit{isempty}(\mathit{symdiff}(S_1, S_2))$

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Gleichheitstest: Ist $S_1 = S_2$?

$\mathit{equal}(S_1, S_2) = \mathit{isempty}(\mathit{symdiff}(S_1, S_2))$

Teilmengentest: Ist $S_1 \subseteq S_2$?

$\mathit{subset}(S_1, S_2) = \mathit{isempty}(\mathit{diff}(S_1, S_2))$

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Gleichheitstest: Ist $S_1 = S_2$?

$\mathit{equal}(S_1, S_2) = \mathit{isempty}(\mathit{symdiff}(S_1, S_2))$

Teilmengentest: Ist $S_1 \subseteq S_2$?

$\mathit{subset}(S_1, S_2) = \mathit{isempty}(\mathit{diff}(S_1, S_2))$

Disjunktheitstest: Ist $S_1 \cap S_2 = \emptyset$?

$\mathit{disjoint}(S_1, S_2) = \mathit{isempty}(\mathit{intersection}(S_1, S_2))$

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Gleichheitstest: Ist $S_1 = S_2$?

$\mathit{equal}(S_1, S_2) = \mathit{isempty}(\mathit{symdiff}(S_1, S_2))$

Teilmengentest: Ist $S_1 \subseteq S_2$?

$\mathit{subset}(S_1, S_2) = \mathit{isempty}(\mathit{diff}(S_1, S_2))$

Disjunktheitstest: Ist $S_1 \cap S_2 = \emptyset$?

$\mathit{disjoint}(S_1, S_2) = \mathit{isempty}(\mathit{intersection}(S_1, S_2))$

Natürlich könnte man diese Operationen auch in die Spezifikation aufnehmen.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2): **Mit Wiederholung**:

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2): **Mit Wiederholung**:
Für *jedes* Element der Darstellung von S_1 durchsuche Darstellung von S_2.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2): **Mit Wiederholung**:
Für *jedes* Element der Darstellung von S_1 durchsuche Darstellung von S_2.
Kosten:

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2): **Mit Wiederholung**:
Für *jedes* Element der Darstellung von S_1 durchsuche Darstellung von S_2.
Kosten: $\Theta(h_1 \cdot h_2)$.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2): **Mit Wiederholung**:
Für *jedes* Element der Darstellung von S_1 durchsuche Darstellung von S_2.
Kosten: $\Theta(h_1 \cdot h_2)$.

Ohne Wiederholung: Darstellung von S_i hat Länge $n_i = |S_i|$.
Alle Operationen haben **Kosten:** $\Theta(n_1 \cdot n_2)$.

Lesehinweise und Kommentare

- Eine Frage, die sich bei Implementierungen von Mengendatenstrukturen mit mehreren Mengen stellt, wurde bisher unterschlagen.
- Wenn man etwa *union*(S_1, S_2) berechnet, sind dann die Darstellungen von S_1 und S_2 hinterher noch vorhanden oder sind sie verschwunden?
- Typische Möglichkeiten:
  - Man fasst *union*(S_1, S_2) als Methode des Objekts auf, das S_1 darstellt, also als „S_1.*union*(S_2)“. Dieses Objekt wird um die Elemente von S_2 erweitert, und S_2 verschwindet („zerstörende Operation“).
    Diese Vorstellung liegt unserer Diskussion auf der vorigen Folie zugrunde, wo man im Fall (A) zwei lineare Listen in Zeit $O(1)$ aneinanderhängt.
    Analog für *intersection*, *diff*, *symmdiff*.
  - Beide Darstellungen sind noch vorhanden, und die Darstellung von $S_1 \cup S_2$ ist ein völlig neues Objekt, z. B. eine neue Liste mit neuen Einträgen. Dann müssen auch bei *union* beide Mengen kopiert werden („nichtzerstörende Operation“), mit entsprechenden Kosten.
- Welche Möglichkeit gemeint ist, muss bei der Spezifikation des Datentyps festgelegt werden.

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

quasiparallelen Durchlauf durch zwei Arrays/Listen,

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

quasiparallelen Durchlauf durch zwei Arrays/Listen, analog zum Mischen (**Merge**) bei **Mergesort** (Vgl. [AuP], Kapitel 7, oder diese Vorlesung, Kap. 6).

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

quasiparallelen Durchlauf durch zwei Arrays/Listen, analog zum Mischen (**Merge**) bei **Mergesort** (Vgl. [AuP], Kapitel 7, oder diese Vorlesung, Kap. 6).

Beginnend mit den beiden ersten Einträgen der Listen bestimmt man immer den kleineren Eintrag der beiden „aktuellen“ Einträge und überträgt diesen in die Ergebnisliste.
Bei zwei gleichen Einträgen in den beiden Listen wird nur eine Kopie in das Ergebnis übernommen.

Für die anderen Operationen: analog.

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

quasiparallelen Durchlauf durch zwei Arrays/Listen, analog zum Mischen (**Merge**) bei **Mergesort** (Vgl. [AuP], Kapitel 7, oder diese Vorlesung, Kap. 6).

Beginnend mit den beiden ersten Einträgen der Listen bestimmt man immer den kleineren Eintrag der beiden „aktuellen“ Einträge und überträgt diesen in die Ergebnisliste.
Bei zwei gleichen Einträgen in den beiden Listen wird nur eine Kopie in das Ergebnis übernommen.

Für die anderen Operationen: analog. Bei Arrays: Analog.

Implementierung des Datentyps *DynSets*

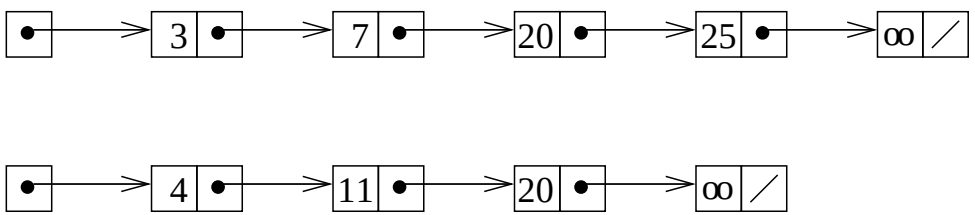
Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein

Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.

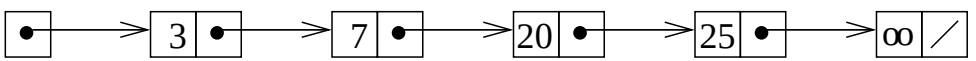
Implementierung des Datentyps *DynSets*

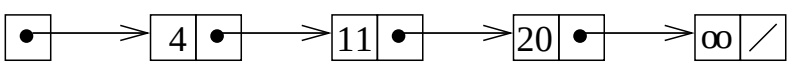
Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.



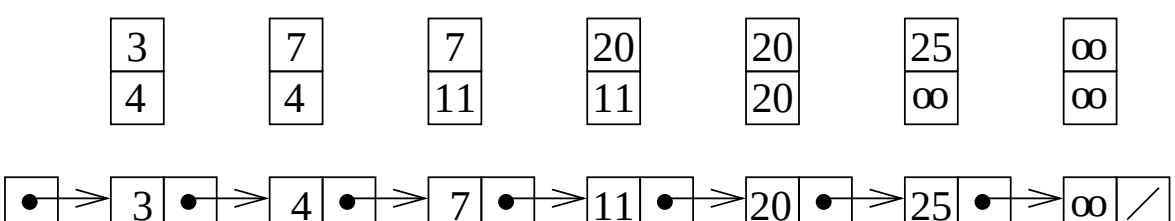
Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.



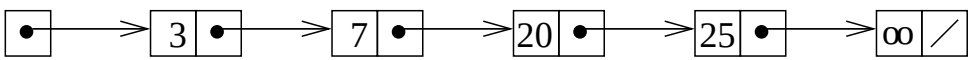


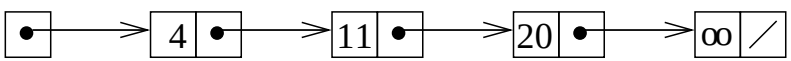
Ausgeführte Vergleiche und Ergebnisliste bei *union*(S_1, S_2):



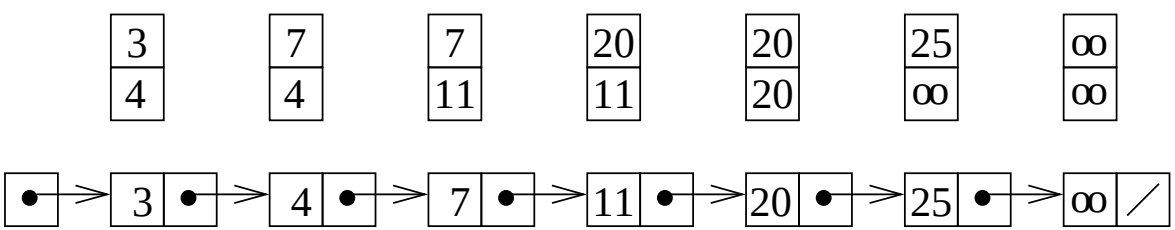
Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.





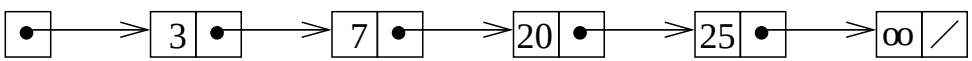
Ausgeführte Vergleiche und Ergebnisliste bei *union*(S_1, S_2):

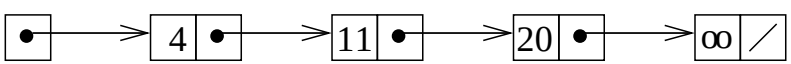


Bei Array-/Listenlängen $|S_1| = n_1$, $|S_2| = n_2$: **Kosten:**

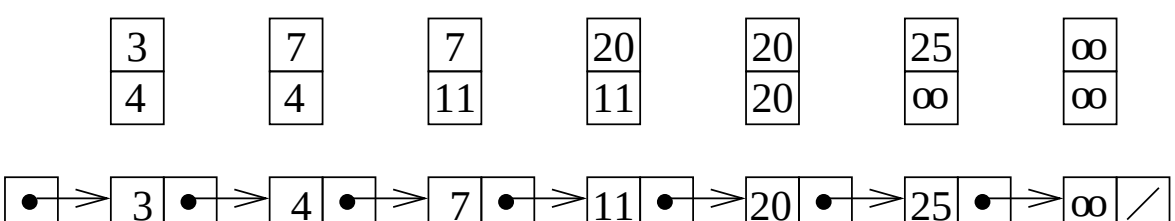
Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.





Ausgeführte Vergleiche und Ergebnisliste bei *union*(S_1, S_2):



Bei Array-/Listenlängen $|S_1| = n_1$, $|S_2| = n_2$: **Kosten:** $\Theta(n_1 + n_2)$.

Lesehinweise und Kommentare

- Wenn mehrere Listen zu verwalten sind und Operationen wie *union* und *intersect* häufig vorkommen, aber keine Veränderungen an einzelnen Mengen, ist die Implementierung mit aufsteigend sortierten Listen bzw. Arrays eine einfache und recht effiziente Möglichkeit.
- Wenn Operationen *insert* und *delete* häufig sind und die Mengen groß werden, sollte man auch für DynSets Suchbäume oder Hashtabellen verwenden, die später betrachtet werden.

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Wir identifizieren x_i mit i, d.h. $D = \{1, \ldots, N\}$.

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Wir identifizieren x_i mit i, d.h. $D = \{1, \ldots, N\}$.

Beispiel: In Pascal:

```
type
Farbe = (rot, gruen, blau, gelb, lila, orange);
Palette = set of Farbe;
```

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Wir identifizieren x_i mit i, d.h. $D = \{1, \ldots, N\}$.

Beispiel: In Pascal:

```
type
Farbe = (rot, gruen, blau, gelb, lila, orange);
Palette = set of Farbe;
```

Hier: $N = 6$.

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Wir identifizieren x_i mit i, d.h. $D = \{1, \ldots, N\}$.

Beispiel: In Pascal:

```
type
Farbe = (rot, gruen, blau, gelb, lila, orange);
Palette = set of Farbe;
```

Hier: $N = 6$.

`rot` $\hat{=}$ 1, `gruen` $\hat{=}$ 2, `blau` $\hat{=}$ 3,
`gelb` $\hat{=}$ 4, `lila` $\hat{=}$ 5, `orange` $\hat{=}$ 6.

Für „kleine“ Mengen: Bitvektor-Darstellung

Repräsentation: Durch Array `A[`$1..N$`] of Boolean`

Für „kleine“ Mengen: Bitvektor-Darstellung

Repräsentation: Durch Array `A[`$1..N$`] of Boolean` wird

$$S = \{i \mid \texttt{A}[i] = 1\}$$

dargestellt.

Wie immer: 0 für $false$, 1 für $true$.

Für „kleine“ Mengen: Bitvektor-Darstellung

Repräsentation: Durch Array `A[1..`N`] of Boolean` wird

$$S = \{i \mid \texttt{A}[i] = 1\}$$

dargestellt.

Wie immer: 0 für $false$, 1 für $true$.

Beispiel: Darstellung der Menge $\{2, 3, 5\} \mathrel{\hat{=}} \{\texttt{gruen}, \texttt{blau}, \texttt{lila}\}$:

0	1	1	0	1	0

Für „kleine“ Mengen: Bitvektor-Darstellung

Repräsentation: Durch Array `A[1..`N`] of Boolean` wird

$$S = \{i \mid \texttt{A}[i] = 1\}$$

dargestellt.

Wie immer: 0 für $false$, 1 für $true$.

Beispiel: Darstellung der Menge $\{2, 3, 5\} \mathrel{\hat{=}} \{\texttt{gruen}, \texttt{blau}, \texttt{lila}\}$:

0	1	1	0	1	0

Platzbedarf:
N Bits, d. h. $\lceil N/8 \rceil$ Bytes oder $\lceil N/64 \rceil$ Speicherworte à 64 Bits.

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$;

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`1`..`N`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$;

Kosten: $\boldsymbol{\Theta(N)}$

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$;

Kosten: $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$;

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $\boldsymbol{O(1)}$

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $\boldsymbol{O(1)}$

delete(S, i):
`A[`i`]` $\leftarrow 0$;

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`;

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:** $O(1)$

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:** $O(1)$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays.

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:** $O(1)$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:** $O(1)$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:** $\Theta(N)$

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $\boldsymbol{O(1)}$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $\boldsymbol{O(1)}$

member(S, i):
return `A[`i`]`; **Kosten:** $\boldsymbol{O(1)}$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:** $\boldsymbol{\Theta(N)}$

isempty(S):
Durchlauf durch ein Array.

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\Theta(N)$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $O(1)$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $O(1)$

member(S, i):
return `A[`i`]`; **Kosten:** $O(1)$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:** $\Theta(N)$

isempty(S):
Durchlauf durch ein Array. **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $\boldsymbol{O(1)}$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $\boldsymbol{O(1)}$

member(S, i):
return `A[`i`]`; **Kosten:** $\boldsymbol{O(1)}$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:** $\boldsymbol{\Theta(N)}$

isempty(S):
Durchlauf durch ein Array. **Kosten:** $\boldsymbol{\Theta(N)}$

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$,

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: $Sets \to Sets$

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\rightarrow$ *Sets* mit der Semantik *compl*$(S) = U - S$ (im math. Modell)

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\to$ *Sets* mit der Semantik *compl*$(S) = U - S$ (im math. Modell) leicht implementieren.

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\rightarrow$ *Sets* mit der Semantik *compl*$(S) = U - S$ (im math. Modell) leicht implementieren. **Kosten:**

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\to$ *Sets* mit der Semantik *compl*$(S) = U - S$ (im math. Modell) leicht implementieren. **Kosten:** $\boldsymbol{\Theta(N)}$

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\to$ *Sets* mit der Semantik $compl(S) = U - S$ (im math. Modell) leicht implementieren. **Kosten:** $\boldsymbol{\Theta(N)}$

Zusatz 3: Eine interessante Variante: Packe 64 Bits in ein Wort (lange vorzeichenlose Zahl). Dann nimmt jedes Element von U nur 1 Bit im Speicher in Anspruch.
(Übung: Programmiere die Operationen.)

Kosten für *DynSets*-Operationen, einfache Implementierungen:

	BV	AmW	LmW	{A,L}oW	A-sort	L-sort
empty	$\Theta(N)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$\boldsymbol{O(1)}$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
delete	$\boldsymbol{O(1)}$	$O(h)$	$O(h)$	$O(n)$	$O(n)$	$O(n)$
member	$\boldsymbol{O(1)}$	$O(h)$	$O(h)$	$O(n)$	$\boldsymbol{O(\log n)}$	$O(n)$
isempty	$\boldsymbol{O(1)}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
union	$\Theta(N)$	$O(h_1+h_2)$	$\boldsymbol{O(1)}$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
intersection	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
diff	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
symdiff	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
compl	$\Theta(N)$	–	–	–	–	–

BV: Bitvektor; A: Array; o: ohne; m: mit; W: Wiederholung; A/L-sort: sortierte(s) Array/Liste

N: Länge des Bitvektors; n, n_1, n_2: Größe von S, S_1, S_2

h, h_1, h_2: Länge der Listen-/Arraydarstellung von S, S_1, S_2

Lesehinweise und Kommentare

- Diese zusammenfassende Tabelle soll dabei helfen, einen Überblick zu gewinnen. Sie ist natürlich nicht zum Auswendiglernen gedacht.
- Gehen Sie die Einträge systematisch durch und überlegen Sie, auf welche Implementierung sich die Einträge beziehen.
- Rot eingetragen sind Werte, die nicht ganz selbstverständlich sind (binäre Suche in sortierten Arrays, Verkettung von zwei Listen bei *union*(S_1, S_2) in Zeit $O(1)$, quasiparalleler Durchlauf bei sortierten Arrays und Listen, etc.
- Wenn Sie die Einträge verstanden haben und erklären können, woher Sie kommen, ohne zurückzublättern, dann hat die Tabelle ihren Zweck erfüllt.

PAUSE

Lesehinweise und Kommentare

- Wir diskutieren jetzt, zum Abschluss von Kapitel 2, den fundamentalen Datentyp „Wörterbuch“.
- Wie wichtig er ist, sieht man an den verschiedenen toll klingenden alternativen Bezeichnungen, u. a. *Map* und *assoziatives Array*, und daran, dass dieser Datentyp praktisch in allen Programmiersprachen fest installiert ist.
- Wir werden zwei wesentlichen Implementierungen dieses Datentyps, nämlich Suchbäumen und Hashtabellen, ganze Kapitel widmen.
- Die Idee ist die: Man möchte so etwas haben wie eine veränderliche mathematische Funktion $f : S \to R$, mit folgendem Verhalten:
  - Wenn man den „Schlüssel“ x in die Datenstruktur hineinruft, soll diese den „Wert“ $f(x)$ liefern.
  - Man möchte neue Schlüssel mit zugehörigen Werten hinzufügen können.
  - Man möchte Schlüssel und zugehörige Werte löschen können.
- S, eine veränderliche Menge, ist immer der Definitionsbereich der Funktion f.
- U, die Menge aller möglichen Schlüssel, ist ein Parameter. Also: $S \subseteq U$, S ist endlich.
- Der Wertebereich R von f ist ebenfalls ein Parameter.

2.4. Der Datentyp Wörterbuch

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

Man möchte:

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

Man möchte:

empty(): ein leeres Wörterbuch erzeugen

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

Man möchte:

empty$()$:	ein leeres Wörterbuch erzeugen
lookup(x):	den dem Schlüssel x zugeordneten Wert $f(x)$ finden

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

Man möchte:

empty$()$:	ein leeres Wörterbuch erzeugen
lookup(x):	den dem Schlüssel x zugeordneten Wert $f(x)$ finden
insert(x, r):	ein neues (Schlüssel, Wert)-Paar (x, r) einbauen

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

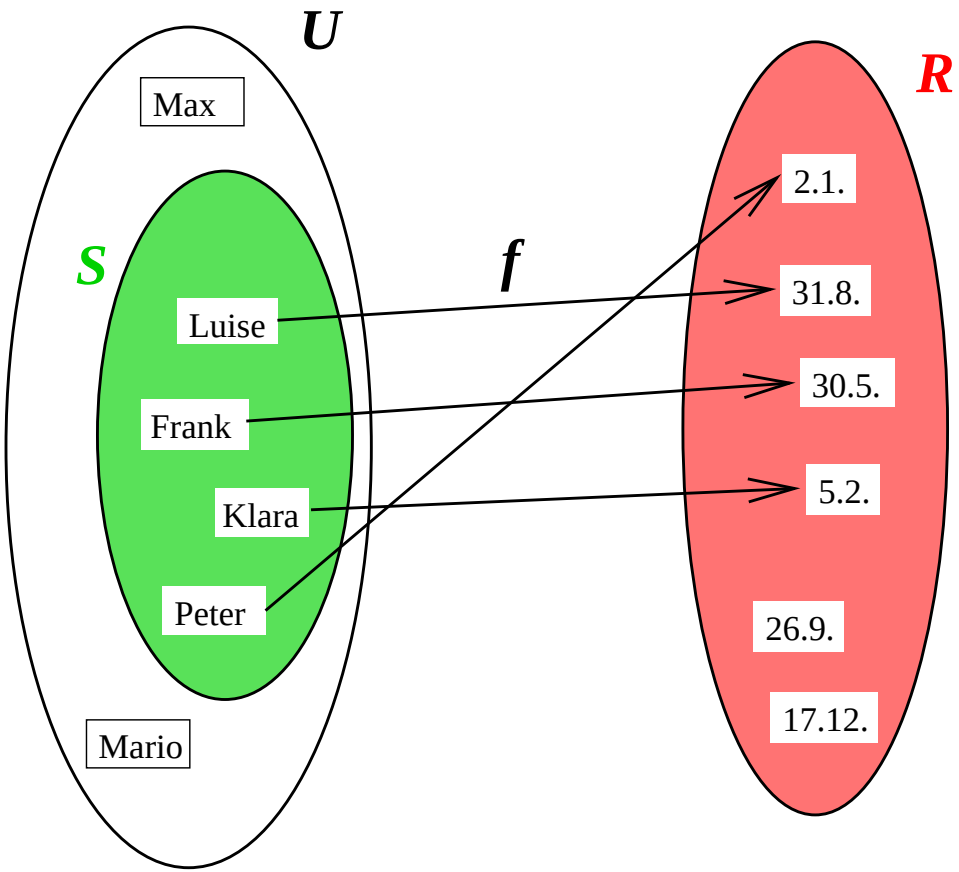
Man möchte:

empty$()$:	ein leeres Wörterbuch erzeugen
lookup(x):	den dem Schlüssel x zugeordneten Wert $f(x)$ finden
insert(x, r):	ein neues (Schlüssel, Wert)-Paar (x, r) einbauen
delete(x):	den Eintrag zu Schlüssel x löschen

Beispiel für eine solche Zuordnung mit $U =$ Menge der endlichen Folgen von Buchstaben A, B, ..., Z, a, b, ..., z und

Beispiel für eine solche Zuordnung mit $U =$ Menge der endlichen Folgen von Buchstaben A, B, ..., Z, a, b, ..., z und $R = \{1, \ldots, 31\} \times \{1, \ldots, 12\}$:

Beispiel für eine solche Zuordnung mit $U =$ Menge der endlichen Folgen von Buchstaben A, B, ..., Z, a, b, ..., z und $R = \{1, \ldots, 31\} \times \{1, \ldots, 12\}$:



Lesehinweise und Kommentare

- Um das folgende Beispiel zu verstehen, erinnere man sich an Folgendes:
- In der Mathematik ist eine Funktion f mit Definitionsbereich $\subseteq U$ und Wertebereich R einfach eine Menge von Paaren $(x, r) \in U \times R$, die „rechtseindeutig“ ist, d. h. zu jedem $x \in U$ gibt es höchstens ein $r \in R$ mit $(x, r) \in f$. Wenn es zu x so ein r gibt, nennen wir es $f(x)$.
- Im Beispiel ist die Funktion immer explizit als eine solche Menge von Paaren dargestellt.
- Wir führen eine Folge $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ von Operationen aus.
- Wie bei Stacks führt dies zu einer Folge von inneren Zuständen (hier: die Menge f der Paare) und zu einem Ein-/Ausgabeverhalten (letzte Spalte).

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“
10	*lookup*(Peter)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“
10	*lookup*(Peter)	” ”	„$\perp$“

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“
10	*lookup*(Peter)	” ”	„$\perp$“
11	*delete*(Mario)		

i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“
10	*lookup*(Peter)	” ”	„$\perp$“
11	*delete*(Mario)	” ”	„!!“

Lesehinweise und Kommentare

- Zeile 0: Der Konstruktor *empty*() erzeugt die *leere Funktion*, dargestellt durch die leere Menge.
- Zeilen 1, 2, 3, und auch Zeile 6: Die *insert*-Operation fügt einen Eintrag hinzu.
- Zeile 4: Ein *lookup*(x) für ein x, das in der Datenstruktur vorkommt, liefert den richtigen Wert.
- Zeile 5: Hoppla. Daran hatten wir noch gar nicht gedacht. Wenn *lookup*(x) für ein nicht vorhandenes x ausgeführt wird, gibt es keinen Fehler, sondern es wird die Ausgabe „$\perp$“ („undefiniert“) ausgegeben. (**Achtung:** Damit legen wir uns darauf fest, dass ein Wörterbuch zwischen $x \in S$ und $x \notin S$ unterscheiden kann.)
- Zeile 7: Wir versuchen ein Paar (x, r) einzufügen, obwohl es den Schlüssel x schon gibt. Wir legen fest, dass dies kein Fehler ist, sondern dass dann der alte Wert $f(x)$ durch den neuen Wert r *ersetzt* werden soll. Das heißt: Ein Wert von f wird geändert, bei gleichem Definitionsbereich.
- Dies nennen wir „*update*“; die *insert*-Operation erledigt es einfach mit (evtl. mit Warnung).
- Zeile 8: *lookup*(x) für den soeben geänderten Eintrag liefert den neuen Wert.
- Zeile 9: *delete*(x) löscht das Paar $(x, f(x))$ aus der Menge f; nun: x nicht mehr im Def.-bereich.
- Zeile 10: Richtige Reaktion auf die Löschung in Zeile 9.
- Zeile 11: Auch *delete*(x) mit einem nicht vorhandenen Schlüssel führt nicht zu einem Fehler, sondern es passiert einfach nichts (außer vielleicht einer Warnung).

Lesehinweise und Kommentare

- Nun spezifizieren wir den Datentyp, nach dem bekannten Schema.
- „*Keys*“ für Schlüssel, „*Values*“ für Werte.
- Die Sorte „*Maps*“ steht für die Wörterbücher oder Abbildungen (englisch: *map*).
- Die Signaturen für die vier Operationen drängen sich geradezu auf.

Datentyp Wörterbuch

Datentyp Wörterbuch

1. Signatur:

Datentyp Wörterbuch

1. Signatur:

Sorten:

Datentyp Wörterbuch

1. Signatur:

Sorten: *Keys*

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps

Operationen:

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps
Operationen:	$empty: \rightarrow Maps$

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps
Operationen:	$empty: \rightarrow Maps$
	$insert: Maps \times Keys \times Values \rightarrow Maps$

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps
Operationen:	$empty: \rightarrow Maps$
	$insert: Maps \times Keys \times Values \rightarrow Maps$
	$delete: Maps \times Keys \rightarrow Maps$

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps
Operationen:	$empty: \rightarrow Maps$
	$insert: Maps \times Keys \times Values \rightarrow Maps$
	$delete: Maps \times Keys \rightarrow Maps$
	$lookup: Maps \times Keys \rightarrow Values$

2. Mathematisches Modell:

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)

Values: Menge R (Menge aller Werte, ein Parameter)

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)

Values: Menge R (Menge aller Werte, ein Parameter)

Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U,\ |S| < \infty\}$

2. Mathematisches Modell:

Sorten:	*Keys*:	Menge U (Menge aller Schlüssel, ein Parameter)
	Values:	Menge R (Menge aller Werte, ein Parameter)
	Maps:	$\{f \mid f\colon S \to R$ für ein $S \subseteq U$, $\lvert S\rvert < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)
Values: Menge R (Menge aller Werte, ein Parameter)
Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U, |S| < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

$f \subseteq S \times R$ und $\forall x \in S\ \exists! r \in R\colon (x, r) \in f$.

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)
Values: Menge R (Menge aller Werte, ein Parameter)
Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U,\ |S| < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

$f \subseteq S \times R$ und $\forall x \in S\ \exists! r \in R\colon (x, r) \in f$.

(„$\exists!$“: „es gibt genau ein“.)

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)
Values: Menge R (Menge aller Werte, ein Parameter)
Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U,\ |S| < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

$f \subseteq S \times R$ und $\forall x \in S\ \exists! r \in R\colon (x, r) \in f$.

(„$\exists!$“: „es gibt genau ein“.)

Für S schreiben wir **Def(f)** (Definitionsbereich)

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)
Values: Menge R (Menge aller Werte, ein Parameter)
Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U,\ |S| < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

$f \subseteq S \times R$ und $\forall x \in S\ \exists! r \in R\colon (x, r) \in f$.

(„$\exists!$“: „es gibt genau ein“.)

Für S schreiben wir $\mathbf{Def}(f)$ (Definitionsbereich)

Falls $x \in \mathrm{Def}(f)$, ist $f(x)$ das eindeutig bestimmte $r \in R$ mit $(x, r) \in f$.

Operationen:

Operationen:

$empty() := \emptyset$.

Operationen:

$\mathit{empty}() := \emptyset$.

$$\mathit{lookup}(f, x) := \begin{cases} f(x), & \text{falls } x \in \mathsf{Def}(f) \\ \text{„}\perp\text{“}, & \text{sonst.} \end{cases}$$

Operationen:

empty$() := \emptyset$.

$$\textit{lookup}(f,x) := \begin{cases} f(x), & \text{falls } x \in \mathsf{Def}(f) \\ \text{„}\perp\text{“}, & \text{sonst.} \end{cases}$$

$$\textit{delete}(f,x) := \begin{cases} f - \{(x, f(x))\}, & \text{falls } x \in \mathsf{Def}(f) \\ f, & \text{sonst.} \end{cases}$$

Operationen:

$\mathit{empty}() := \emptyset$.

$$\mathit{lookup}(f, x) := \begin{cases} f(x), & \text{falls } x \in \mathsf{Def}(f) \\ \text{„}\perp\text{“}, & \text{sonst.} \end{cases}$$

$$\mathit{delete}(f, x) := \begin{cases} f - \{(x, f(x))\}, & \text{falls } x \in \mathsf{Def}(f) \\ f, & \text{sonst.} \end{cases}$$

$$\mathit{insert}(f, x, r) := \begin{cases} (f - \{(x, f(x))\}) \cup \{(x, r)\}, & \text{falls } x \in \mathsf{Def}(f) \\ f \cup \{(x, r)\}, & \text{sonst.} \end{cases}$$

Operationen:

$\mathit{empty}() := \emptyset$.

$$\mathit{lookup}(f, x) := \begin{cases} f(x), & \text{falls } x \in \mathsf{Def}(f) \\ \text{„}\perp\text{“}, & \text{sonst.} \end{cases}$$

$$\mathit{delete}(f, x) := \begin{cases} f - \{(x, f(x))\}, & \text{falls } x \in \mathsf{Def}(f) \\ f, & \text{sonst.} \end{cases}$$

$$\mathit{insert}(f, x, r) := \begin{cases} (f - \{(x, f(x))\}) \cup \{(x, r)\}, & \text{falls } x \in \mathsf{Def}(f) \\ f \cup \{(x, r)\}, & \text{sonst.} \end{cases}$$

Achtung: „$\perp$“ („*undefiniert*“) ist ein reguläres Ergebnis. Natürlich sollte $\perp$ kein Element von R sein.

Lesehinweise und Kommentare

- Diese Definition des mathematischen Modells von Wörterbüchern sei Ihnen zum Studium **sehr empfohlen**.
- Zu *empty*() gibt es wenig zu sagen: Der Konstruktor erzeugt die leere Funktion $\emptyset$.
- *lookup*(f, x) reagiert für $x \in \text{Def}(f)$ mit dem Wert $f(x)$, für $x \notin \text{Def}(f)$ mit der Mitteilung, dass der Schlüssel x nicht bekannt ist.
- Das Wörterbuch schließt also die Funktionalität des Datentyps DynSet für die Menge $S = \text{Def}(f)$ ein.
- *delete*(x) löscht das Paar $\{(x, f(x))\}$, wenn es ein solches gibt, sonst passiert nichts.
- *insert*(f, x, r) kann man sich zweistufig vorstellen: Wenn $x \in \text{Def}(f)$, dann wird der Eintrag für x zuerst gelöscht, genau wie in *delete*. Dann wird auf jeden Fall das Paar (x, r) hinzugenommen.
- Stellen Sie sicher, dass Sie diese Spezifikation des **wichtigsten Datentyps der Vorlesung** auch wiedergeben können, trotz der eventuell verwickelt aussehenden mengentheoretischen Schreibweisen!
- Wir werden Implementierungen dieses Datentyps ausführlich studieren.
- Eine Implementierung heißt korrekt, wenn sie dasselbe Ein-/Ausgabeverhalten hat wie das mathematische Modell.

Implementierungsmöglichkeiten

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$, ohne Wiederholung von Schlüsseln, unsortiert oder sortiert.

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$, ohne Wiederholung von Schlüsseln, unsortiert oder sortiert.
- Wenn $U = \{1, \ldots, N\}$: Array `f[1..N]` mit Werten in $R \cup \{\perp\}$, wobei $\perp \mathrel{\hat{=}}$ „*undefiniert*“.

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$, ohne Wiederholung von Schlüsseln, unsortiert oder sortiert.
- Wenn $U = \{1, \ldots, N\}$: Array `f[1..N]` mit Werten in $R \cup \{\perp\}$, wobei $\perp \mathrel{\hat{=}}$ „*undefiniert*“.
- **Suchbäume** und **Hashtabellen** (später).

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$, ohne Wiederholung von Schlüsseln, unsortiert oder sortiert.
- Wenn $U = \{1, \ldots, N\}$: Array `f[1..N]` mit Werten in $R \cup \{\perp\}$, wobei $\perp \mathrel{\hat{=}}$ „*undefiniert*“.
- **Suchbäume** und **Hashtabellen** (später).

Zeiten für Operationen wie bei Implementierungen des Datentyps *DynSet* (Folie 76).

Lesehinweise und Kommentare

- Man erhält einfache Implementierungen aus unseren Datenstrukturen für Mengen, indem man anstelle von puren Datenobjekten, die gleichzeitig Schlüssel sind, Paare (x, r) speichert.
- Für die Suche verwendet man dann nur den „Schlüsselteil“ x der Paare $(x, f(x))$.
- „Mit Wiederholung“: *insert*(f, x) fügt das Paar (x, r) vorne in die Liste ein, ohne Suche. Damit: Zeit $O(1)$. Bei *lookup*(x) ist der erste gefundene Eintrag mit Schlüssel x relevant.
- „Ohne Wiederholung“: Bei *insert*(f, x) muss man den Schlüssel x zuerst suchen, um zwischen der echten Neueinfügung und der update-Situation zu unterscheiden.
- „Ohne Wiederholung“ + „aufsteigend sortiert“: Die Sortierung bezieht sich natürlich nur auf die Schlüssel. Bei der Arrayimplementierung stehen auch hier alle Varianten der binären Suche zur Verfügung, so dass Suche Zeit $\log n$ benötigt, für $n = |S| = |\mathrm{Def}(f)|$.
- Wenn $U = \{1, \ldots, N\}$ und N nicht zu groß ist, kann man das Bitarray (Folien 72–75) so verallgemeinern, dass man in ein Array `f[1..`N`]` an Stelle $x \in \mathrm{Def}(f)$ direkt den Funktionswert $f(x)$ einträgt, und natürlich $\perp$ für $x \notin \mathrm{Def}(f)$. Dies liefert dann konstante Zugriffszeiten.
- Es ergeben sich dieselben Rechenzeiten wie in der Tabelle auf Folie 76.
- Schlauere Implementierungsmöglichkeiten mit Rechenzeiten $\boldsymbol{O(\log n)}$ oder gar $\boldsymbol{O(1)}$ für alle Operationen kommen später.