SS 2021

Algorithmen und Datenstrukturen

2. Kapitel

Fundamentale Datentypen und Datenstrukturen

Martin Dietzfelbinger

Mai 2021

Thema: Datentypen und Datenstrukturen

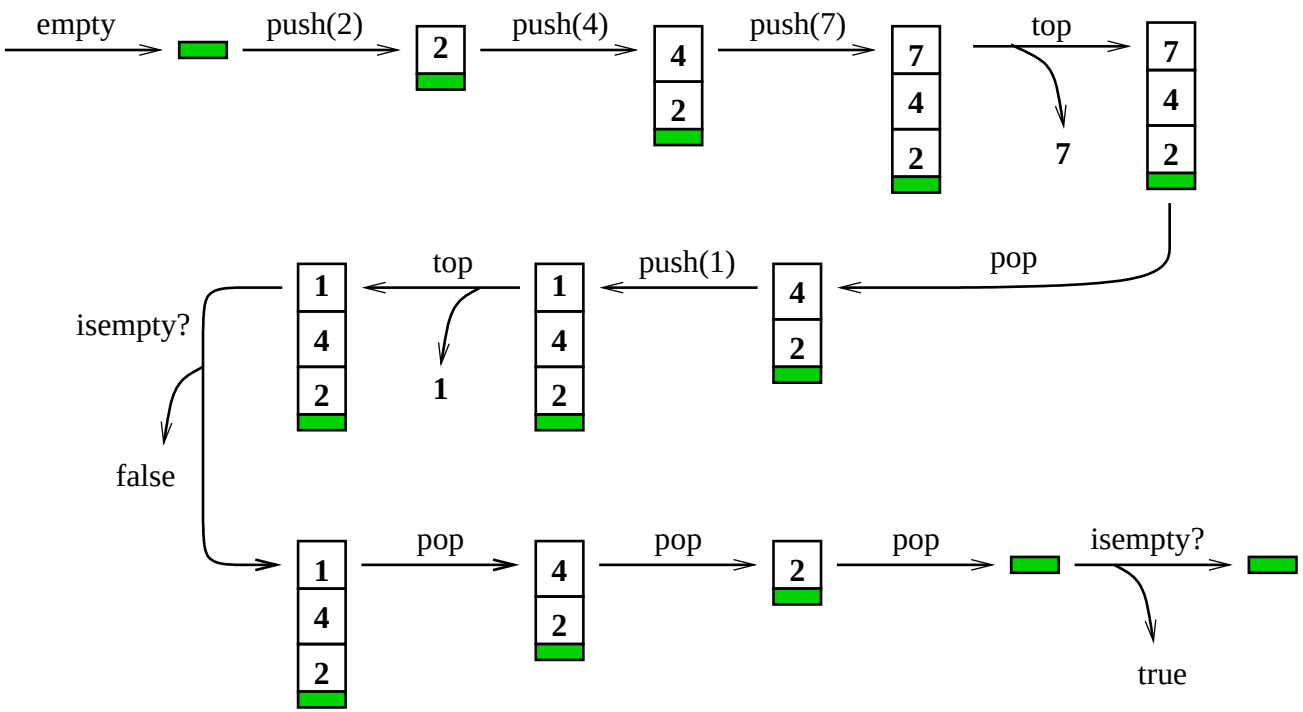
Einfache Datentypen (werden hier diskutiert)

- **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)

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).

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**.

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!

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.)

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

Operationen: *empty*() := ()

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

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

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

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

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.

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

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

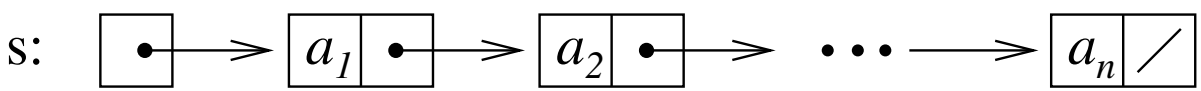
Der Benutzer sieht die Details der Implementierung nicht. Zugriff auf die Objekte der Datenstruktur erfolgt ausschließlich über die (öffentlichen) Methoden der Klassen, entsprechend den Operationen des Datentyps.

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

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

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 $a_1, \ldots, a_n$:



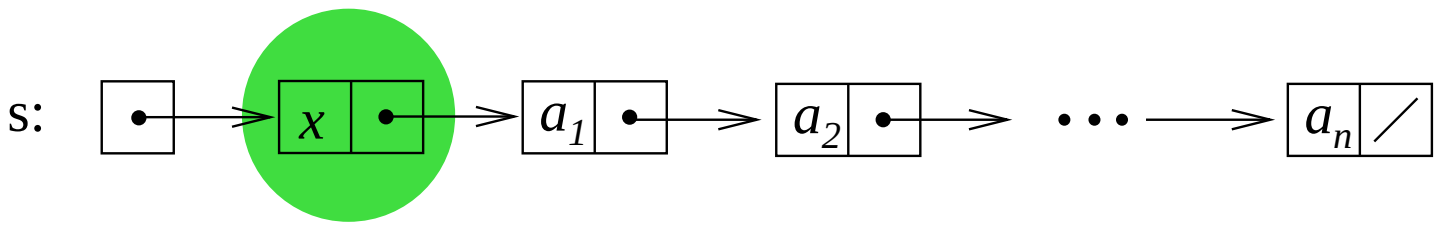
(Das Symbol ⁄ steht für den Nullzeiger/die Nullreferenz)

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

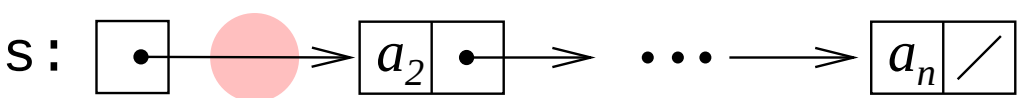
`s.empty`: Erzeugt neue leere Liste.

Listenimplementierung von Stacks

`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



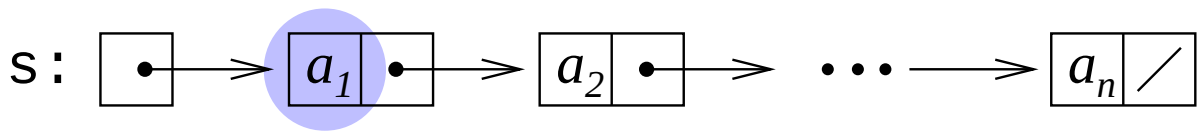
`s.pop`: Entferne ersten Listenknoten (falls vorhanden). Aus der Liste für $(a_1, \ldots, a_n)$ mit $n \geq 1$ wird also (an der Stelle des roten Kreises stand vorher a_1):



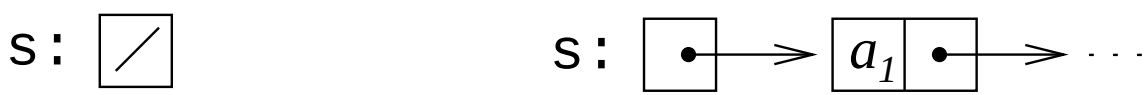
Wenn `s.pop` für eine leere Liste aufgerufen wird, muss eine Fehlermeldung (exception o. ä.) erfolgen.

Listenimplementierung von Stacks

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



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



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

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. $\{true, false\}$).

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

$\mathrm{Op}_0 = \mathit{empty}()$ 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!)

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 $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ 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.

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$_i$).

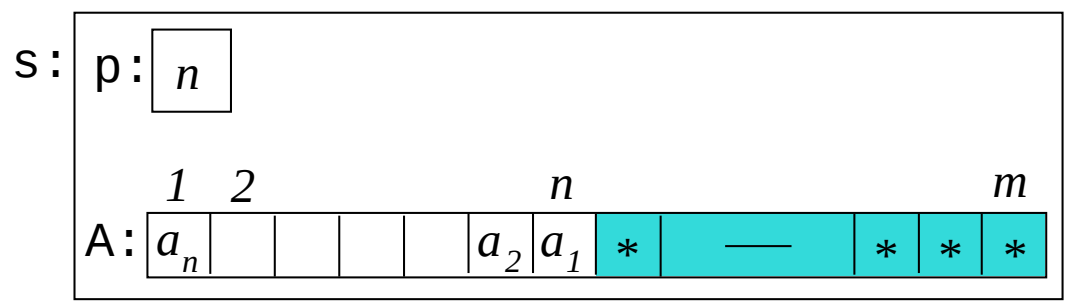
Wenn Op$_i$ = *isempty*, liefert das mathematische Modell $z_i = true$ bzw. $z_i = false$, je nachdem ob $n \geq 1$ oder $n = 0$ gilt. Genau dieselbe Ausgabe wird von der Implementierung erzeugt. Also gilt (IB$_i$).

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

(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

`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** „$false$“;

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 $0, \ldots, N$:

(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.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\texttt{A}[n_0], \ldots, \texttt{A}[1]) = () = s_0$.
(gleichgültig welchen Inhalt $\texttt{A}[1..m]$ hat).
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

Behauptung 2.1.3

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

Das sieht man durch Betrachtung der einzelnen Operationen.

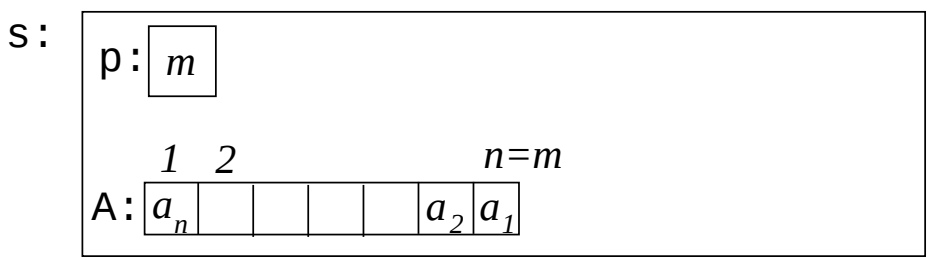
Bei der Initialisierung `empty()` (Aufruf des Konstruktors) muss man aber auf Folgendes achten: Wenn man m, die Arraygröße, vom Benutzer als Parameter angeben lässt (was sehr sinnvoll ist!), dann darf das Array `A[`$1..m$`]` vom Konstruktor nur **alloziert**, nicht initialisiert werden. In C++ ist dies 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.

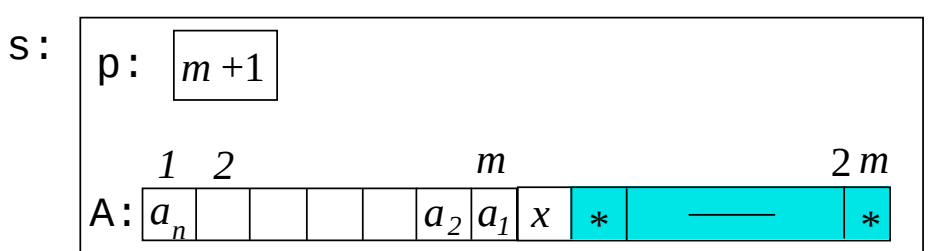
Vermeidung von Platzproblemen: Verdoppelungsstrategie

`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:

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

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 $N \cdot \Theta(1) + O(k) = \Theta(N)$.

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 $\Theta(N)$.
(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),
- 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.

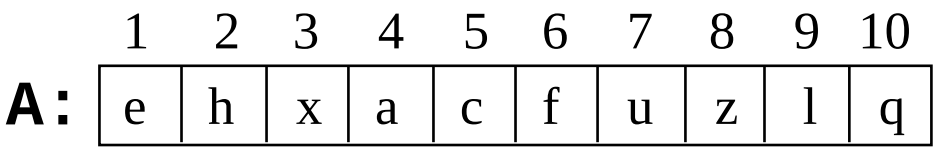
Experimente zeigen, dass die Arrayimplementierung spürbar kleinere Rechenzeiten aufweist.

PAUSE

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:

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

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

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

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

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

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

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

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

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

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.

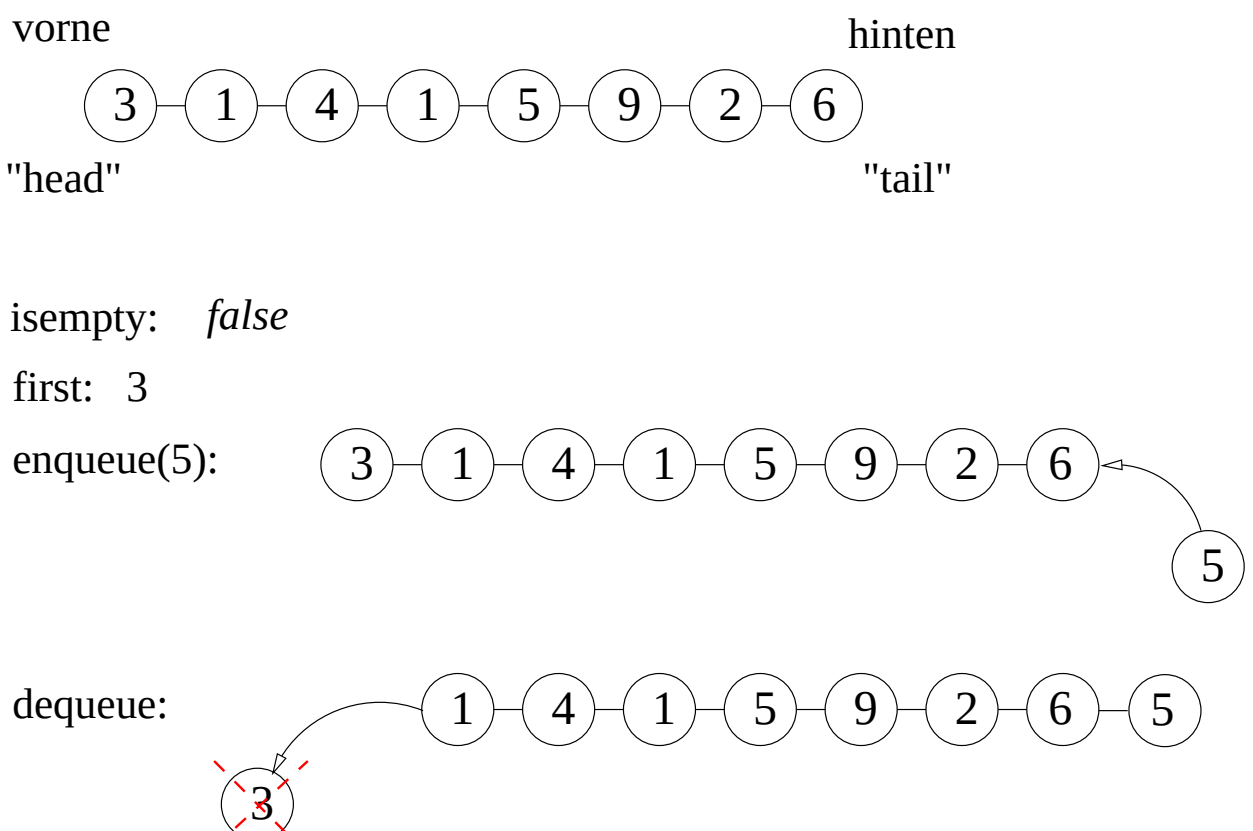
Mitteilung

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“.

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

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



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!

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.)

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

„1. Signatur“: wie oben.

2. Axiome:

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

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

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

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

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

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

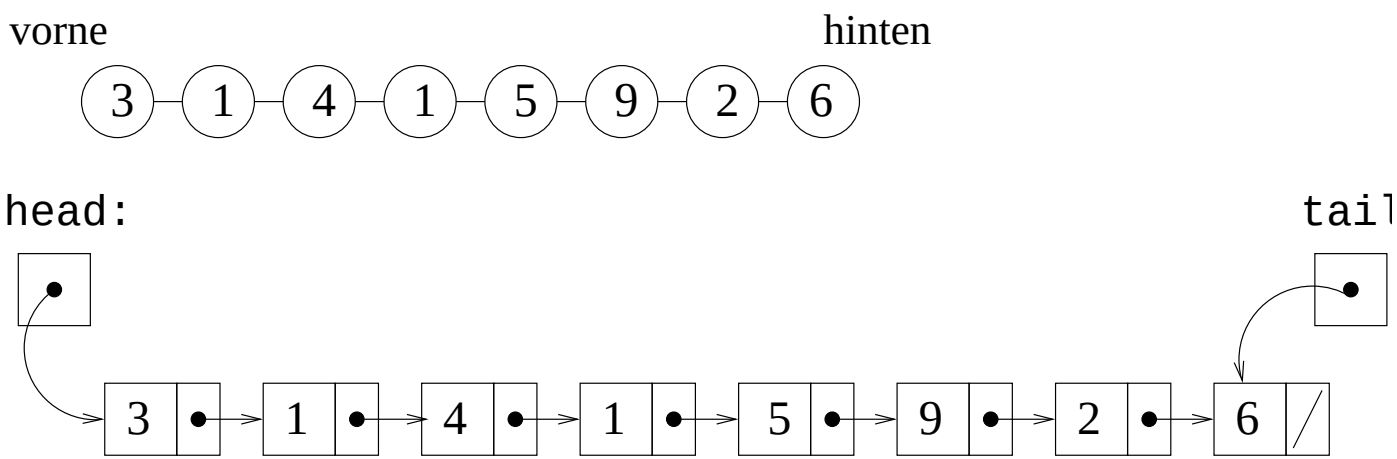
$$\mathit{isempty}(\mathit{enqueue}(q, x)) = \mathit{false}$$

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.

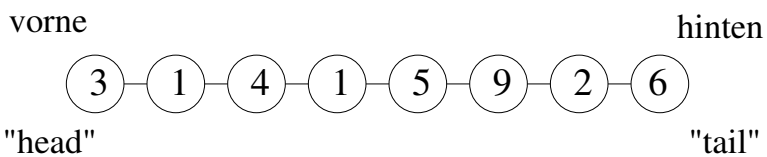
Listenimplementierung von Queues

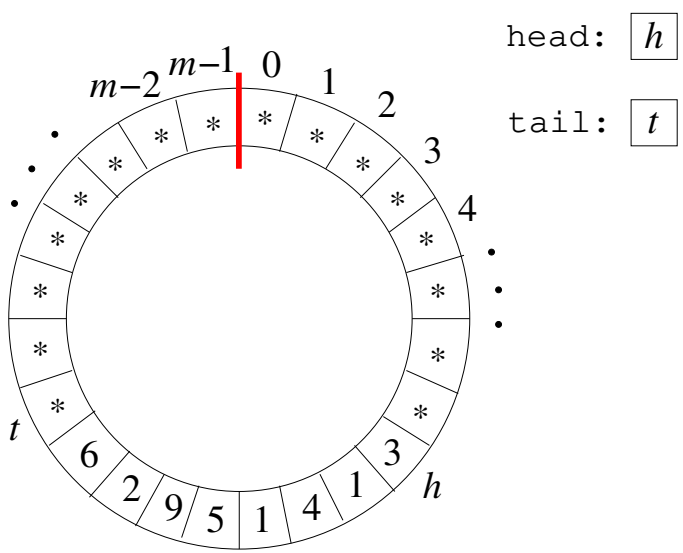
Implementierung mittels einfach verketteter Listen
mit Listenendezeiger/-referenz: Skizze.



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:
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:

`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.

„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 Op_0, Op_1, . . . , 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

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

(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

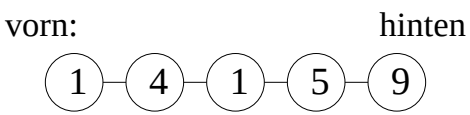
Bei einer Queue mit Arrays und der Verdoppelungsstrategie sind die Gesamtkosten für N Operationen $O(N)$.
Platzbedarf: $O(k)$, wenn k die maximale je erreichte Länge der Queue ist.

Beweise: Analog zu Stacks.

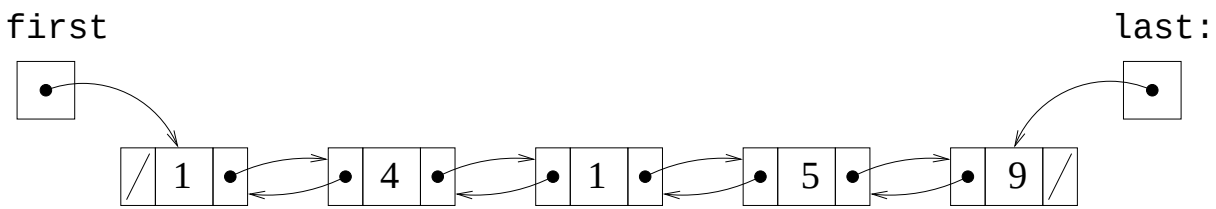
Datentyp Doppelschlange – Deque

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

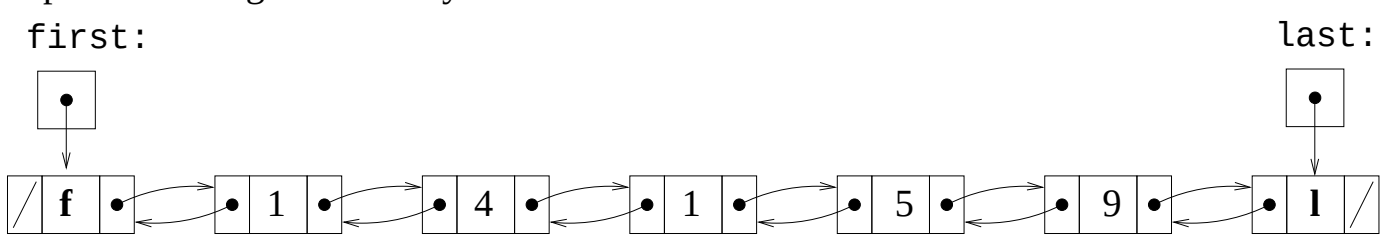
Schema:



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.

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

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

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)**

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

Boolean: $\{true, false\}$

Operationen:

$empty() := \emptyset$

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

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

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

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

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).

(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 $\texttt{A}[a\,..\,b]$

Überlege:

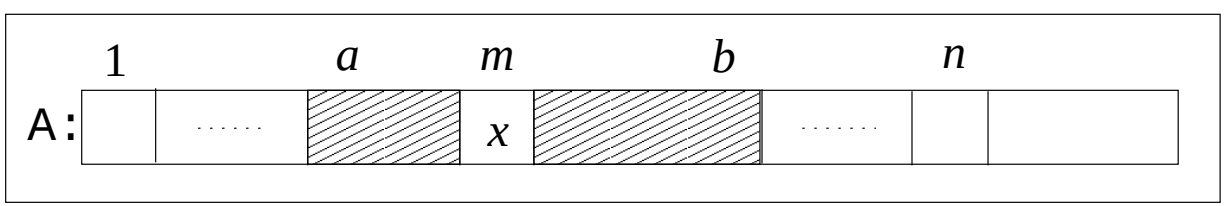
0. Falls $b < a$, ist x nicht im Teilarray.

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

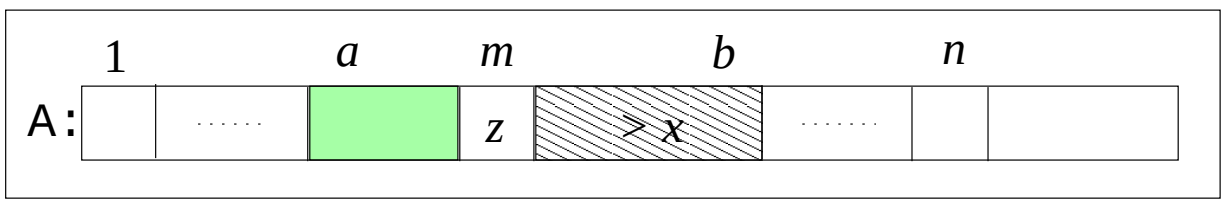
3 Fälle:

$x = \texttt{A}[m]$: x gefunden, Antwort $true$.



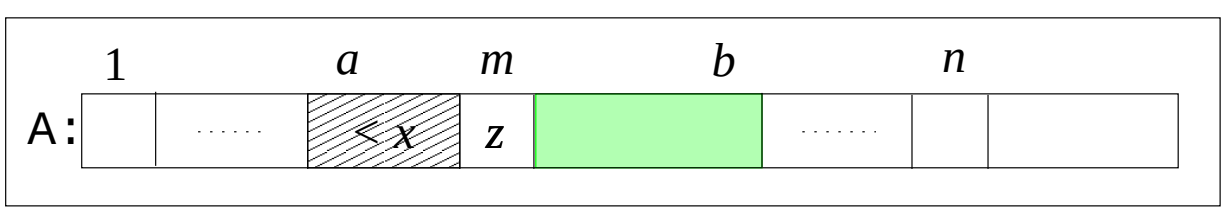
Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.



$x < z$

$x > \texttt{A}[m]$: 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 $\texttt{A}[1\,..\,n]$.

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).
```

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 $d - c + 1 < i$.

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. □

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**: $r(0) = r(1) = 1$.

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:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

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)!)

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): $\texttt{A}[1..n]$ 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 $i(x) = n + 1$.

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

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.

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 $S = \emptyset$?)

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

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 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:** $\Theta(n_1 \cdot n_2)$.

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

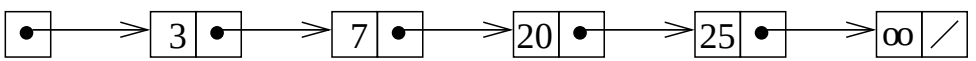
quasiparallelen Durchlauf durch zwei Arrays/Listen, analog zum Mischen (**Merge**) bei **Mergesort** (Vgl. [AuP], Kapitel 7, oder diese Vorlesung, Kap. 6).

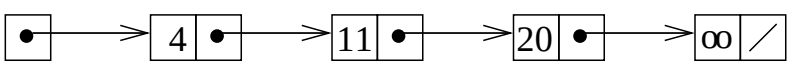
Beginnend mit den beiden ersten Einträgen der Listen bestimmt man immer den kleineren Eintrag der beiden „aktuellen“ Einträge und überträgt diesen in die Ergebnisliste.
Bei zwei gleichen Einträgen in den beiden Listen wird nur eine Kopie in das Ergebnis übernommen.

Für die anderen Operationen: analog. Bei Arrays: Analog.

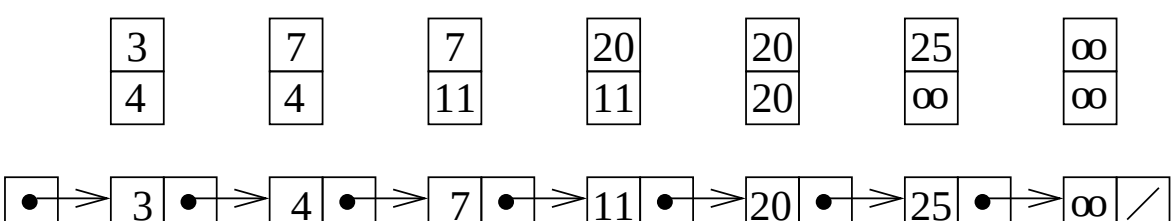
Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein 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:** $\Theta(n_1 + n_2)$.

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` 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]` $\leftarrow 0$; **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:** $\Theta(N)$

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

PAUSE

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*“).

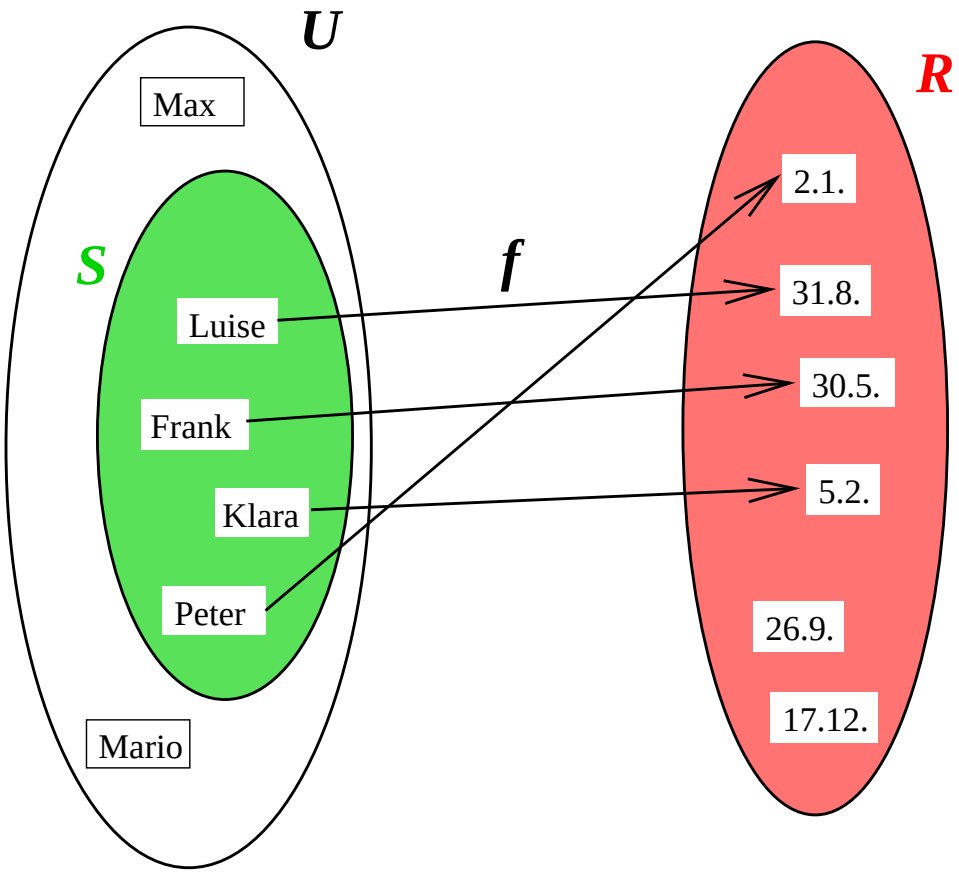
Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

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 $R = \{1, \ldots, 31\} \times \{1, \ldots, 12\}$:



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)	” ”	„$\perp$“
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)	” ”	„!!“

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:

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 $x \in \text{Def}(f)$, ist $f(x)$ das eindeutig bestimmte $r \in R$ mit $(x, r) \in f$.

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.

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 77).