Algorithmen und Datenstrukturen

Kapitel 4: Datenstrukturen

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

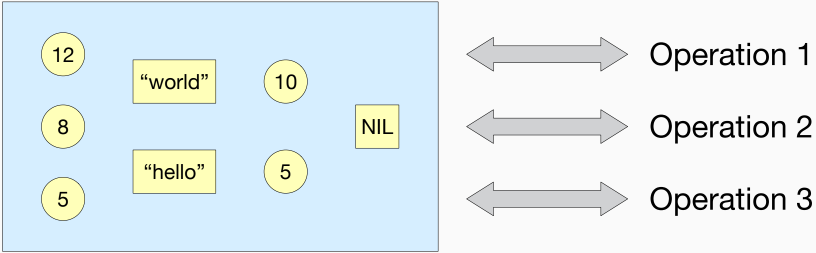
1. Elementare Datenstrukturen
2. Binäre Suchbäume
3. Balancierte Suchbäume
4. Hashing

1) Elementare Datenstrukturen

Was ist eine Datenstruktur?

Definition 4.1

Eine Datenstruktur ist gegeben durch eine Menge von Objekten sowie eine Menge von Operationen auf diesen Objekten.



Definition 4.2

Ein Dictionary ist eine Datenstruktur, welche die Operationen INSERT (Einfügen), REMOVE (Entfernen) sowie SEARCH (Suchen) unterstützt.

Wörterbuch

Definition 4.3

Eine Priority Queue ist eine Datenstruktur, welche die Operationen INSERT (Einfügen), REMOVE (Entfernen) sowie SEARCHMIN (Suchen des Minimums) bzw. SEARCHMAX (Suchen des Maximums) unterstützt.

Prioritätswarteschlange

Ein grundlegendes Datenbankproblem

Speicherung und Verarbeitung von Datensätzen!

Beispiel

Verwalten von Kundendaten wie:

- Name, Adresse, Wohnort
- Kundennummer
- offene Bestellungen oder Rechnungen
- ...

Anforderungen

- schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Zugriff auf Objekte

- Objekte meist durch Schlüssel identifiziert
- Eingabe des Schlüssels liefert gewünschten Datensatz
- über den Schlüsseln gibt es eine totale Ordnung

Ver-
gleich-
barkeit

Beispiel

- Objekt Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: ~~Name~~ Kundennummer
- Totale Ordnung: ~~lexikographische Ordnung~~ „≤“

Typische elementare Operationen

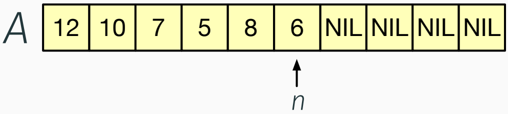
- $\underline{\textsc{Insert}(S, x)}$: Füge Objekt x in S ein.
- $\underline{\textsc{Search}(S, k)}$: Finde Objekt x in S mit Schlüssel k. Falls kein solches Objekt in S existiert, gib NIL zurück.
- $\underline{\textsc{Remove}(S, x)}$: Entferne Objekt x aus S.
- $\underline{\textsc{SearchMin}(S)}$: Finde das Objekt mit minimalem Schlüssel in S. Hierbei muss eine Ordnung auf den Schlüsseln existieren.
- $\underline{\textsc{SearchMax}(S)}$: Finde das Objekt mit maximalem Schlüssel in S. Hierbei muss eine Ordnung auf den Schlüsseln existieren.

Ziele

- Objekte: Zahlen
- Operationen: Einfügen, Suchen, Entfernen

Umsetzung

- beschränke maximale Größe auf max
- speichere Objekte in Array $A[1 \ldots \mathsf{max}]$
- speichere Anzahl aktueller Objekte als n mit $1 \leq n \leq \mathsf{max}$



Eine einfache Datenstruktur: statisches Feld

Ziele

- Objekte: Zahlen
- Operationen: Einfügen, Suchen, Entfernen

Umsetzung

- beschränke maximale Größe auf max
- speichere Objekte in Array $A[1 \dots \text{max}]$
- speichere Anzahl aktueller Objekte als n mit $1 \leq n \leq \text{max}$

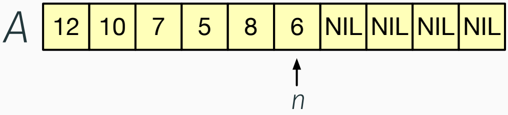
A										
	12	10	7	5	8	6	NIL	NIL	NIL	NIL
						n				

- in diesem Beispiel sind die Schlüssel gleich den Objekten

Implementierung statischer Felder

INSERT(x)

if $n = \text{max}$: **return** „Error: out of space"
$n \leftarrow n + 1$
$A[n] \leftarrow x$



Wie gut sind statische Felder?

Charakteristiken

- Platzbedarf: max
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen $\Theta(1)$

Vorteile

- schnelles Einfügen
- schnelles Löschen

Nachteile

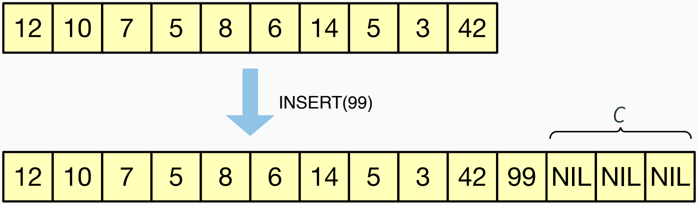
- Speicherbedarf hängt von max ab…
  …und ist nicht vorhersagbar
- hohe Laufzeit für Suche

- aktuell maximale Länge sei ℓ

length

Idee

Wenn Array A zu klein ($n > \ell$), generiere neues Array der Größe $\ell + c$ für feste Konstante c.



Ist das eine gute Implementierung eines dynamischen Feldes?

Überschlagsrechnung

- Zeitaufwand der Erweiterung ist $\Theta(\ell)$
- Zeitaufwand für n INSERT Operationen:
  - Aufwand $\Theta(\ell)$ für je c INSERT Operationen
    - also $\Theta(c)$ für die ersten c INSERT Operationen, ...
    - ...$\Theta(2c)$ für die zweiten c INSERT Operationen, ...
    - ...$\Theta(3c)$ für die dritten c INSERT Operationen, ...
    - ...
  - Gesamtaufwand:

$$\sum_{i=1}^{n/c} i \cdot c = \Theta(n^2)$$

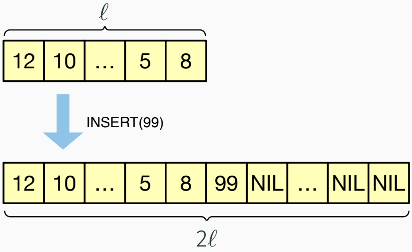
- Also durchschnittliche lineare Laufzeit für INSERT!

Das muss doch besser gehen!?!

Bessere dynamische Felder

Idee

Wenn Array A zu klein ($n > \ell$), generiere neues Array der doppelten Größe 2ℓ.

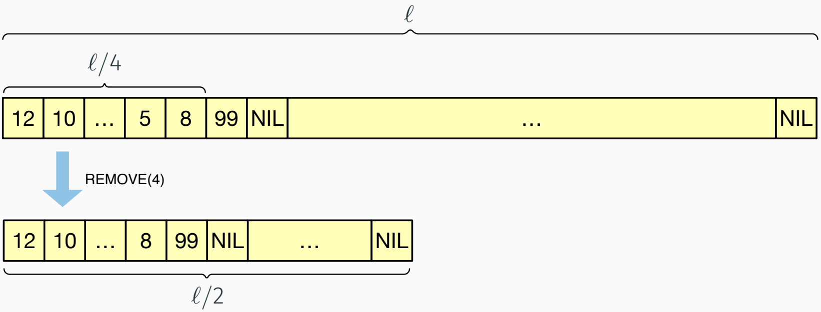


Bessere dynamische Felder

schrumpfen

Idee

Wenn Array A zu groß ($n \leq \ell/4$), generiere neues Array der halben Größe $\ell/2$.



Laufzeit solch dynamischer Felder

Lemma 4.1:

Betrachte ein Anfangs leeres dynamisches Feld A. Jede Folge σ von n INSERT und REMOVE Operationen auf A kann in Zeit $\Theta(m)$ bearbeitet werden.

Beweis später

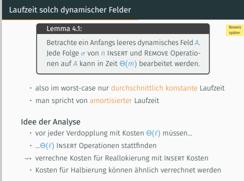
- also im worst-case nur durchschnittlich konstante Laufzeit
- man spricht von amortisierter Laufzeit

Idee der Analyse

- vor jeder Verdopplung mit Kosten $\Theta(\ell)$ müssen...
- ...$\Theta(\ell)$ INSERT Operationen stattfinden

$\rightsquigarrow$ verrechne Kosten für Reallokierung mit INSERT Kosten

- Kosten für Halbierung können ähnlich verrechnet werden



- man spricht hier auch von einem charging argument
- die Kosten der Reallokierungen werden den entsprechenden INSERT Operationen „gecharged“ (zugewiesen)

Wie gut sind dynamische Felder?

Charakteristiken

- Platzbedarf: $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Amortisierte Laufzeit Einfügen/Löschen $\Theta(1)$

Vorteile

- schnelles Einfügen
- schnelles Löschen
- Speicherbedarf linear in n

Nachteile

- hohe Laufzeit für Suche

Mögliche Verbesserungen

- Sortiertes dynamisches Feld?
  - schnellere Suche, aber linearer LZ beim Einfügen/Löschen
- Idee: sortiertes dynamisches Feld mit Lücken
  - geschickt verteilte Lücken erlauben Einfügen/Löschen in amortisierter LZ $\Theta(\log^2 n)$

Wie gut sind dynamische Felder?

Charakteristiken

- Platzbedarf: $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Amortisierte Laufzeit Einfügen/Löschen $\Theta(1)$

Vorteile

- schnelles Einfügen
- schnelles Löschen
- Speicherbedarf linear in n

Nachteile

- hohe Laufzeit für Suche

Mögliche Verbesserungen

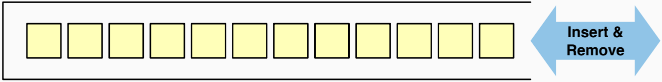
- Sortiertes dynamisches Feld?
  - schnellere Suche, aber linearer LZ beim Einfügen/Löschen
- Idee: sortiertes dynamisches Feld mit Lücken
  - geschickt verteilte Lücken erlauben Einfügen/Löschen in amortisierter LZ $\Theta(\log^2 n)$

- mittels Tricks auch worst-case Laufzeit $\Theta(1)$ (Stichwort „progressives Umkopieren“)
- Grund für lineare LZ: vergleiche mit innerer Schleife von InsertionSort
- Das ist das Prinzip einer Bibliothek! (Bücher alphabetisch sortiert; es gibt immer ein paar Lücken; wenn es eng wird, werden neue Regale angeschafft)
- die Analyse hiervon ist allerdings recht komplex

Stack

Stapel

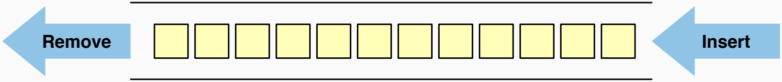
Eine Datenstruktur die das LIFO (last-in-first-out) Prinzip implementiert.



Queue

(Warte-)Schlange

Eine Datenstruktur die das FIFO (first-in-first-out) Prinzip implementiert.



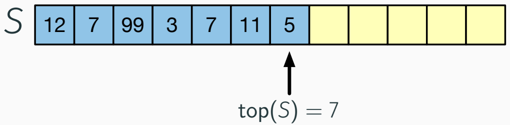
Stacks

Operationen

- PUSH: Einfügen eines Objektes
- POP: Entfernen des zuletzt eingefügten Objektes
- EMPTY: Überprüft ob Stack leer

Implementierung

- Stack mit maximal max Elementen
  - speichere Objekte in Array $S[1 \ldots \mathrm{max}]$
  - $\mathrm{top}(S)$ speichert Index des zuletzt eingefügten Objektes
- maximale Größe nicht bekannt $\rightsquigarrow$ dynamisches Feld



Implementierung der Stack-Operationen

EMPTY(S)

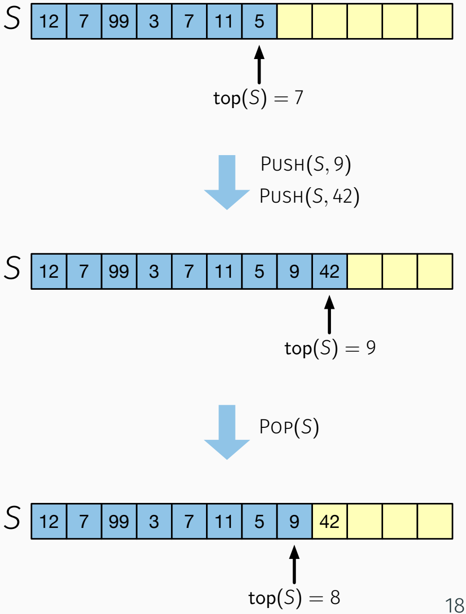
```
if top(S) = 0: return TRUE
else:          return FALSE
```

PUSH(S, x)

```
top(S) ← top(S) + 1
S[top(S)] ← x
```

POP(S)

```
if EMPTY(S)
    return „Error: underflow“
top(S) ← top(S) − 1
return S[top(S) + 1]
```

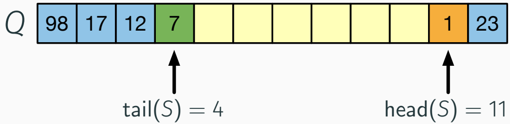


Operationen

- ENQUEUE: Einfügen eines Objektes
- DEQUEUE: Entfernen des ältesten Objektes in der Queue
- EMPTY: Überprüft ob Queue leer

Implementierung

- Queue mit maximal max Elementen
  - speichere Objekte in Array $Q[1 \dots \max +1]$
  - $\text{head}(Q)$ Index des ältesten Objektes in der Queue
  - $\text{tail}(Q)$ „erste“ freie Position
    - interpretieren Array Kreisförmig (auf Position n folgt Position 1)
- maximale Größe nicht bekannt $\rightsquigarrow$ dynamisches Feld

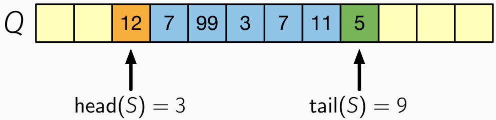


Implementierung der Queue-Operationen

EMPTY(Q)

```
if head(Q) = tail(Q)
    return TRUE
else
    return FALSE
```

ENQUEUE(Q, 42)
ENQUEUE(Q, 3)



DEQUEUE(Q)

```
if EMPTY(Q): return „Error: underflow“
x ← Q[head(Q)]
if head(Q) = length(Q)
    head(Q) ← 1
else
    head(Q) ← tail(Q) + 1
return x
```

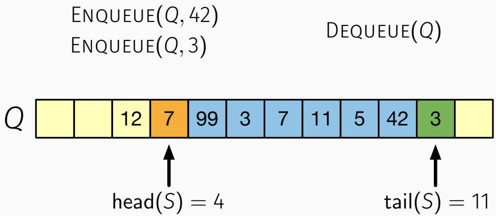
ENQUEUE(Q, x)

```
Q[tail(Q)] ← x
if tail(Q) = length(Q)
    tail(Q) ← 1
else
    tail(Q) ← tail(Q) + 1
```

Implementierung der Queue-Operationen

Empty(Q)

```
if head(Q) = tail(Q)
    return TRUE
else
    return FALSE
```



Dequeue(Q)

```
if EMPTY(Q): return „Error: underflow“
x ← Q[head(Q)]
if head(Q) = length(Q)
    head(Q) ← 1
else
    head(Q) ← tail(Q) + 1
return x
```

Enqueue(Q, x)

```
Q[tail(Q)] ← x
if tail(Q) = length(Q)
    tail(Q) ← 1
else
    tail(Q) ← tail(Q) + 1
```

Effizienz von Stacks & Queues

Theorem 4.1

Die Operationen eines statischen Stacks können mit Laufzeit $\Theta(1)$ implementiert werden.

Theorem 4.2

Die Operationen einer statischen Queue können mit Laufzeit $\Theta(1)$ implementiert werden.

Für dynamische Stacks/Queues:

- dynamische Felder statt Arrays $\rightsquigarrow$ amortisierte Laufzeit $\Theta(1)$
- alternativ: Datenstrukturen mit Zeigern

more to come...

2) Binäre Suchbäume

...

- ...

3) Balancierte Suchbäume

...

- ...

4) Hashing

...

- ...