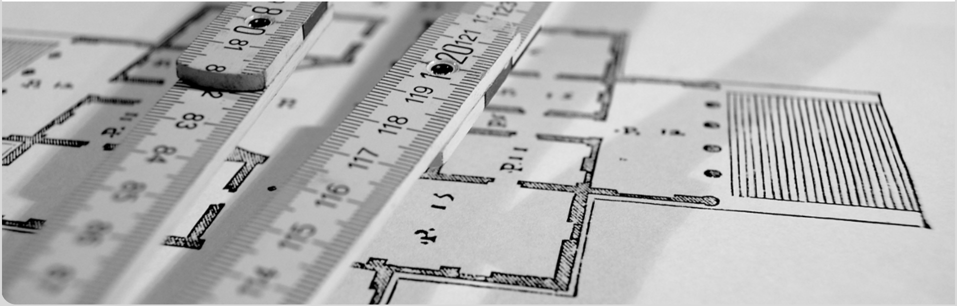
Karlsruher Institut für Technologie

Tutorium 3:
Felder und Hashing

Holger Ebhart | 6. Mai 2015

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS15



Gliederung

Übungsblatt 2

unbounded Arrays

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

unbounded Arrays

Karlsruher Institut für Technologie

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

unbounded Arrays

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A) \rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

unbounded Arrays

- alloziere Array A[1...n]
- pushBack:
  - A voll $\rightarrow$ alloziere $A'[1...2|A|]$ und kopiere A nach A'
  - hänge Element am Ende ein
- popBack:
  - entnehme letztes Element
  - $\frac{1}{4}|A| >= used(A)$ $\rightarrow$ alloziere $A'[1...\frac{1}{2}|A|]$ und kopiere A nach A'
- dynamisch wachsendes Array (verhält sich wie Liste)

Amortisierte Analyse

- pushBack $\rightarrow$ 2 Token ◯ ◯
- popBack $\rightarrow$ 1 Token ◯
- Wann habe ich welche Laufzeit?
- Meistens konstante Laufzeit $\mathcal{O}(1)$
- Bei jeder 2^n-ten pushBack Operation linear $\mathcal{O}(n)$
- Bei jeder $n \cdot \frac{1}{2^n}$-ten popBack Operation linear $\mathcal{O}(n)$

Amortisierte Analyse

- pushBack $\rightarrow$ 2 Token ◯ ◯
- popBack $\rightarrow$ 1 Token ◯
- Wann habe ich welche Laufzeit?
- Meistens konstante Laufzeit $\mathcal{O}(1)$
- Bei jeder 2^n-ten pushBack Operation linear $\mathcal{O}(n)$
- Bei jeder $n \cdot \frac{1}{2^n}$-ten popBack Operation linear $\mathcal{O}(n)$

Amortisierte Analyse

- pushBack $\rightarrow$ 2 Token $\bigcirc$ $\bigcirc$
- popBack $\rightarrow$ 1 Token $\bigcirc$
- Wann habe ich welche Laufzeit?
- Meistens konstante Laufzeit $\mathcal{O}(1)$
- Bei jeder 2^n-ten pushBack Operation linear $\mathcal{O}(n)$
- Bei jeder $n \cdot \frac{1}{2^n}$-ten popBack Operation linear $\mathcal{O}(n)$

cyclic Arrays

Karlsruher Institut für Technologie

```
class cyclicArray
        A:Array[0...n] of Element
        h := 0 :int //head
        t := 0 :int //tail

        function size : int
                return (t-h+n+1) mod (n+1)

        function pushBack(e:Element)
                A[t] := e
                t := (t+1) mod (n+1)
        function pushFront(e:Element)
                if h-- = -1 then h := n
                A[h] := e
```

cyclic Arrays

```
function popBack : Element
    if t-- = -1 then t := n
    return A[t]
function popFront : Element
    h^ := h :int
    h := (h+1) mod (n+1)
    return A[h^]
```

Aufgabe

Gegeben: Arrays $A[1...n_1], B[1...n_2] \subseteq N^*$ aufsteigend sortiert
Gesucht: Array $C[1...n](n := n_1 + n_2)$ aufsteigend sortiert

Wie sollte ein Algorithmus aussehen der das Problem löst?

concat sorted arrays

procedure *merge*(A : Array $[1..n_1]$ of $\mathbb{N}_{\geq 0}$, B : Array $[1..n_2]$ of $\mathbb{N}_{\geq 0}$)
$A[n_1 + 1] := \infty$, $B[n_2 + 1] := \infty$
$n := n_1 + n_2$
$j_A := 1$, $j_B := 1$;
for $i := 1$ **to** n **do**
    $C[i] = \min(A[j_A], B[j_B])$
    if $A[j_A] < B[j_B]$ **then**
        $j_A = j_A + 1$
    else
        $j_B = j_B + 1$
    invariant $C[1..i]$ enthält genau $A[1..j_A - 1], B[1..j_B - 1]$
    invariant $B[k] \leq A[j_A] \quad \forall k \in \{1..j_B - 1\}$,
$A[k] \leq B[j_B] \quad \forall k \in \{1..j_A - 1\}$
    invariant $C[1..i]$ ist sortiert
postcondition $C[i] \leq C[j] \quad \forall i \leq j, i, j \in \{1, ..., n\}$
return C

Aufgabe

- eine Speicherallokation kostet nun $\mathcal{O}(1)$
- Entwickeln sie eine Datenstruktur, die folgendes kann:

  - pushBack in $\mathcal{O}(1)$
  - popBack in $\mathcal{O}(1)$
  - wahlfreier Zugriff in $\mathcal{O}(\log n)$

  - pushBack in $\mathcal{O}(\log n)$
  - popBack in $\mathcal{O}(\log n)$
  - wahlfreier Zugriff in $\mathcal{O}(1)$

Aufgabe

- eine Speicherallokation kostet nun $\mathcal{O}(1)$
- Entwickeln sie eine Datenstruktur, die folgendes kann:
  - pushBack in $\mathcal{O}(1)$
  - popBack in $\mathcal{O}(1)$
  - wahlfreier Zugriff in $\mathcal{O}(\log n)$

  - pushBack in $\mathcal{O}(\log n)$
  - popBack in $\mathcal{O}(\log n)$
  - wahlfreier Zugriff in $\mathcal{O}(1)$

Hashtabellen

Karlsruher Institut für Technologie

- speichert stets *key* $\leftrightarrow$ *value* Paar
- jedem Wert *v* wird ein Schlüssel *key(v)* zugeordnet
- Funktionalität:
  - insert(*value*)
  - remove(*key*)
  - find(*key*)
- Hashing-Invariante: $\forall e \in M : t[h(key(e))] = e$

Hashtabellen

- speichert stets *key* $\leftrightarrow$ *value* Paar
- jedem Wert *v* wird ein Schlüssel *key(v)* zugeordnet
- Funktionalität:
  - insert(*value*)
  - remove(*key*)
  - find(*key*)
- Hashing-Invariante: $\forall e \in M : t[h(key(e))] = e$

Hashtabellen

- speichert stets *key* $\leftrightarrow$ *value* Paar
- jedem Wert *v* wird ein Schlüssel *key(v)* zugeordnet
- Funktionalität:
  - insert(*value*)
  - remove(*key*)
  - find(*key*)
- Hashing-Invariante: $\forall e \in M : t[h(key(e))] = e$

Hashtabellen

- speichert stets *key* $\leftrightarrow$ *value* Paar
- jedem Wert *v* wird ein Schlüssel *key(v)* zugeordnet
- Funktionalität:
  - insert(*value*)
  - remove(*key*)
  - find(*key*)
- Hashing-Invariante: $\forall e \in M : t[h(key(e))] = e$

Hashtabellen

- speichert stets *key* $\leftrightarrow$ *value* Paar
- jedem Wert *v* wird ein Schlüssel *key(v)* zugeordnet
- Funktionalität:
  - insert(*value*)
  - remove(*key*)
  - find(*key*)
- Hashing-Invariante: $\forall e \in M : t[h(key(e))] = e$

Beispiel

- Hashfunktion: $h(v) = v \mod 7$
- Elemente: $v \in \{1, 2, \ldots, 11\}$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$ und verketteten Listen
- Was ist mit der Invariante: $\forall e \in M : t[h(key(e))] = e$?
- Neue Invariante: $\forall e \in M : e \in t[h(key(e))]$
- stets versuchen die 1.Invariante anzustreben, dies ist aber praktisch unmöglich

Beispiel

- Hashfunktion: $h(v) = v \mod 7$
- Elemente: $v \in \{1, 2, \ldots, 11\}$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$ und verketteten Listen
- Was ist mit der Invariante: $\forall e \in M : t[h(key(e))] = e$?
- Neue Invariante: $\forall e \in M : e \in t[h(key(e))]$
- stets versuchen die 1.Invariante anzustreben, dies ist aber praktisch unmöglich

Beispiel

- Hashfunktion: $h(v) = v \mod 7$
- Elemente: $v \in \{1, 2, \dots, 11\}$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$ und verketteten Listen
- Was ist mit der Invariante: $\forall e \in M : t[h(key(e))] = e$?
- Neue Invariante: $\forall e \in M : e \in t[h(key(e))]$
- stets versuchen die 1.Invariante anzustreben, dies ist aber praktisch unmöglich

Beispiel

- Hashfunktion: $h(v) = v \mod 7$
- Elemente: $v \in \{1, 2, \ldots, 11\}$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$
- Hashing mit Array $A[0, 1, 2, 3, 4, 5, 6]$ und verketteten Listen
- Was ist mit der Invariante: $\forall e \in M : t[h(key(e))] = e$?
- Neue Invariante: $\forall e \in M : e \in t[h(key(e))]$
- stets versuchen die 1.Invariante anzustreben, dies ist aber praktisch unmöglich

Aufgabe

Gegeben sei ein unsortiertes Array $A = A[1 \cdots n] \subseteq \mathbb{N}^n$. Finden sie für ein $x \in \mathbb{N}$ ein Paar $(A[i], A[j]), 1 \leq i, j \leq n$ für das gilt: $A[i] + A[j] = x$.

- Es sei $A = (7, 15, 21, 14, 18, 3, 9)$ und $x = 33$
- finden sie einen Algorithmus der das Problem in erwarteter Zeit $\mathcal{O}(n)$ löst und bei Erfolg ein Paar $(A[i], A[j])$ ausgibt, ansonsten `NIL`.

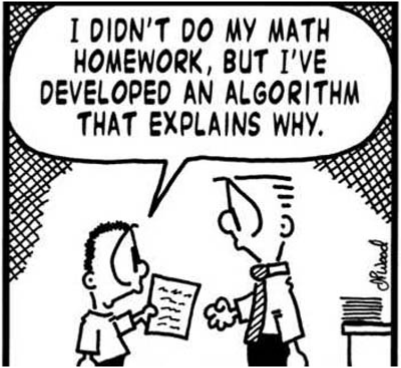
Rekursionen lösen

Löse folgende Rekursionen:

- $f(1) = 1; f(n) = 1 + f(\frac{n}{2})$
- $g(1) = a, (a \in \mathbb{R}^+); g(n) = n + 3 \cdot g(\frac{n}{4})$

Benutze **nicht** das Mastertheorem!

Vielen Dank für eure Aufmerksamkeit!
Bis zum nächsten Mal.



you-can-be-funny.com