Alle Definitionen zusammen

!Nicht "Alle" nur die, die ich lerne!

O-Notation

$\mathcal{O}(g) := \{ f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq k \rightarrow f(n) \leq c \cdot g(n) \}$

$\Omega(g) := \{ f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \quad \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} (n \geq k \rightarrow f(n) \geq c \cdot g(n) \}$

$\Theta(g) := \mathcal{O}(g) \cap \Omega(g)$

Master-Theorem

$\alpha, \beta \in \mathbb{N}$ und $\beta \geq 2$
$\delta \in \mathbb{R}$ und $\delta \geq 0$
$f : \mathbb{N} \rightarrow \mathbb{R}_+$ mit der Form

$$f(n) = \alpha \cdot f(n // \beta) + \mathcal{O}(n^\delta)$$

1. Fall: $\alpha < \beta^\delta \rightarrow f(n) \in \mathcal{O}(n^\delta)$
2. Fall: $\alpha = \beta^\delta \rightarrow f(n) \in \mathcal{O}(n^\delta \cdot \log_\beta(n))$
3. Fall: $\alpha > \beta^\delta \rightarrow f(n) \in \mathcal{O}(n^{\log_\beta(\alpha)})$

Sortierproblem

Partielle Ordnung

1. $\forall x \in S : x \leq x$ (Reflexiv)
2. $\forall x, y \in S : (x \leq y \land y \leq x \rightarrow x = y)$ (anti-Stimmetrie)
3. $\forall x, y, z \in S : (x \leq y \land y \leq z \rightarrow x \leq z)$ (transitiv)

Quasi Ordnung

1. $\forall x \in S : x \leq x$ (Reflexiv)
2. $\forall x, y, z \in S : (x \leq y \land y \leq z \rightarrow x \leq z)$ (transitiv)

Lineare Ordnung

1. Partiell geordnet
2. $\forall x, y \in S : (x \leq y \lor y \leq x)$ (Linear)

Totale Quasiordnung

1. Quasiordnung
2. $\forall x, y \in S : (x \leq y \lor y \leq x)$ (Linear)

Sortierproblem

gegeben sei $\langle M, \leq \rangle$ in totaler Quasiordnung.
L ist Liste aus M
gesucht ist S mit folgenden Bedingungen

1. $\forall i \in \{0, \ldots, len(S) - 2\} : S[i] \leq S[i+1]$
2. $\forall x \in M : count(x, L) = count(x, S)$

Sortier-Algorithmen:

Insertion Sort:

sort([]) = []
sort([x]+R) = insert(x, sort(R))
insert(x, []) = [x]
$x \leq y \rightarrow$ insert([x], [y]+R) = [x]+[y]+R
$x > y \rightarrow$ insert([x], [y]+R) = [y] + insert(x, R)

best case: $\mathcal{O}(n)$
worst case: $\frac{1}{2} \cdot n^2 + \mathcal{O}(n)$
average case: $\frac{1}{4} \cdot n^2 + \mathcal{O}(n)$

Selection Sort

sort([]) = []
x := min(L)
sort(L) = [x] + sort(delete(x, L))
delete(x, []) = []
delete(x, [x]+R) = R
$x \neq y \rightarrow$ delete(x, [y]+R) = delete(x, R)
min([]) = ∞
min([x]+R) = min(x, min(R))
$\min(x, y) \begin{cases} x & \text{falls } x<y \\ y & \text{sonst} \end{cases}$

immer $\frac{1}{2} \cdot n^2 + \mathcal{O}(n)$

Merge Sort

n := len(L)
$n < 2 \rightarrow$ sort(L) = L
$n \geq 2 \rightarrow$ sort(L) = merge(sort(L[:n//2]), sort(L[n//2:]))
merge(L_1, []) = L_1
merge([], L_2) = L_2
$L_1 \neq [] \land L_2 \neq [] \land x_1 \leq x_2$
$\rightarrow$ merge([x_1|R_1], [x_2|R_2]) = [x_1] + merge(R_1, [x_2|R_2])
$L_1 \neq [] \land L_2 \neq [] \land \neg(x_1 \leq x_2)$
$\rightarrow$ merge([x_1|R_1], [x_2|R_2]) = [x_2] + merge([x_1|R_1], R_2)

immer $\mathcal{O}(n \cdot \log_2(n))$

oder $2 \cdot f(n//2) \cdot \mathcal{O}(n)$

Quicksort

sort([]) = []
sort([x]+R) = sort([y ∈ R | y ≤ x]) + [x] + sort([y ∈ R | y > x])

worst case $\mathcal{O}(n^2)$
aver. case $\underbrace{2 \cdot \ln(2)}_{1,39} \cdot n + \mathcal{O}(n \cdot \log_2(n))$

Counting sort

1. Zählphase
2. Indixing-Phase
3. Distribution-Phase

immer $\mathcal{O}(n)$

Heapsort

Nil.toList() = []
$h \neq$ Nil $\land$ <p, _> = h.top()
$\rightarrow$ h.toList() = [p] + h.remove().toList()

immer $\mathcal{O}(n \cdot \log_2(n))$

Heaps

Menge H der Heaps

Nil ∈ H
Node(p, v, l, r) ∈ H gdw.
  $p \leq l \land p \leq r$
  | l.count() − r.count | ≤ 1
  l, r ∈ H

count H → $\mathbb{N}$
Top, remove, isEmpty, Insert,

Top

Nil.top() = Nil
Node(p, v, l, r).top() = <p, v>

Count

Nil.count() = 0
Node(p, v, l, r).count() = r.count() + l.count + 1

isEmpty

Nil.isEmpty() = true
Node(p, v, l, r).isEmpty() = false

INSERT

$p_{top} \leq p \land$ l.count() ≤ r.count()
$\rightarrow$ Node(p_{top}, v_{top}, l, r).insert(p, v) = Node(p_{top}, v_{top}, l.insert(p, v), r)
$p_{top} \leq p \land$ l.count() > r.count
$\rightarrow$ Node(p_{top}, v_{top}, l, r).insert(p, v) = Node(p_{top}, v_{top}, l, r.insert(p, v))
$p_{top} > p \land$ l.count() ≤ r.count
$\rightarrow$ Node(p_{top}, v_{top}, l, r).insert(p, v) = Node(p, v, l.insert(p_{top}, v_{top}), r)
$p_{top} > p \land$ l.count() > r.count
$\rightarrow$ Node(p_{top}, v_{top}, l, r).insert(p, v) = Node(p, v, l, r.insert(p_{top}, v_{top}))

REMOVE

Nil.remove(h) = Nil
Node(p, v, Nil, r) = r
Node(p, v, l, Nil) = l
l = Node(p_1, v_1, l_1, r_1) $\land$ r = (p_2, v_2, l_2, r_2) $\land$ $p_1 \leq p_2$
$\rightarrow$ Node(p, v, l, r).remove() = Node(p_1, v_1, l.remove(), r)
l = Node(p_1, v_1, l_1, r_1) $\land$ r = (p_2, v_2, l_2, r_2) $\land$ $p_1 > p_2$
$\rightarrow$ Node(p, v, l, r).remove() = Node(p_2, v_2, l, r.remove())

Abstrakt Datentypen (ADT)

Formale Definition von ADT's

$D := \langle N, P, F_s, T_s, Ax \rangle$

N = Name des ADT's als String
P = Typparameter als Menge
F_s = Funktionsspezifikationen als Menge (für
T_s = Typspezifikationen als Menge
Ax = Axiome

Vorteile von ADT's:
1) wiederverwendbar
2) Austauschbar
3) Abstrahieren von der Implementierung

Stack

N = "stack"
P = {Element}
F_s = {stack, push, pop, top, isEmpty}
T_s:
  stack : Stack
  push : Stack × Element → Stack
  pop : Stack → Stack ∪ {Ω}
  top : Stack → Element ∪ {Ω}
  isEmpty : Stack → $\mathbb{B}$

Ax:
  S.push(x).isEmpty() = false
  S.push(x).top() = x
  S.push(x).pop() = S
  Stack().isEmpty() = true
  Stack().top() = Ω
  Stack().pop() = Ω

Map

N = "Map"
P = {Key, Value}
F_s = {map, insert, delete, find}
T_s:
  map : Nil
  insert : Map × Key × Value → Map
  delete : Map × Key → Map
  find : Map × Key → Value ∪ {Ω}

Ax:
  Map().find(k) = Ω
  Map().insert(k, v).find(k) = v
  $k_1 \neq k_2 \rightarrow$ Map().insert(k_1, v).find(k_2) = m.find(k_2)
  Map().delete(k).find(k) = Ω
  $k_1 \neq k_2 \rightarrow$ Map.delete(k_1).find(k_2) = m.delete(k_2)

geordnete Binärbäume

Menge der B (Binärbäume)

Nil ∈ B
Node(k, v, l, r) ∈ B g.d.w.
  k ∈ Keys
  v ∈ Values
  l, r ∈ B
  l, r sind unterbäume von Node(___)
  Alle Element aus l < k
  " " " r > k

  map : Nil
  insert : B × k × v → B
  delete : B × k → B
  find : B × k → v ∪ {Ω}
  delMin: B → B × k × v

INSERT

Nil.insert(k, v) = Node(k, v, Nil, Nil)
Node(k_1, v_1, l, r).insert(k_1, v_2, l, r) = Node(k_1, v_2, l, r)
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).insert(k_2, v_2, l, r) = r.insert(k_2, v_2)
$k_1 > k_2 \rightarrow$ l.insert(k_2, v_2)

DELETE

Nil.delete(k) = Nil
Node(k, v, Nil, r).delete(k) = r
Node(k, v, l, Nil).delete(k) = r
l, r ≠ Nil $\land$ r.delMin() < r', k_{min}, v_{min}>
Node(k, v, l, r).delete(k) = Node(k_{min}, v_{min}, l, r')
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).delete(k_2) = Node(k_1, v_1, l, r.delete(k_2))
$k_1 > k_2 \rightarrow$ " " " = " (k_1, v_1, l.delete(k_2), r)

DELMIN

l.delMin() = <l', k_{min}, v_{min}>
Node(k, v, Nil, r).delMin() = <r, k, v>
Node(k, v, l, r).delMin = <Node(k, v, l', r), k_{min}, v_{min}>

FIND

Nil.find(k) = Nil
Node(k, v, l, r).find(k) = v
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).find(k_2) = r.find(k_2)
$k_1 > k_2 \rightarrow$ " " = l.find(k_2)

AVL-Bäume

Menge A der AVL Bäume

Nil ∈ A
Node(k, v, l, r) ∈ A gdw.

1. Node(k, v, l, r) ∈ B
2. l, r ∈ A
3. | l.height() − r.height() | ≤ 1

insert : A × k × v → A
delete : A × k → A
find : A × k → V ∪ {Ω}
delMin: A → A × k × v
Restore: B → A

INSERT

Nil.insert(k, v) = Node(k, v, Nil, Nil)
Node(k_1, v_1, l, r).insert(k_1, v_2, l, r) = Node(k_1, v_2, l, r)
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).insert(k_2, v_2) = Node(k_1, v_1, l, r.insert(k_2, v_2)).restore()
$k_1 > k_2 \rightarrow$ " " = Node(k_1, v_1, l.insert(k_2, v_2), r).restore()

FIND

Nil.find(k) = Nil
Node(k, v, l, r).find(k) = v
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).find(k_2) = r.find(k_2)
$k_1 > k_2 \rightarrow$ " " = l.find(k_2)

DELETE

Nil.delete(k) = Nil
r.delMin() := <r', k_{min}, v_{min}> $\land$ l, r ≠ Nil
Node(k, v, l, r).delete(k) = Node(k_{min}, v_{min}, l, r').restore()
Node(k, v, Nil, r).delete(k) = r
Node(k, v, l, Nil).delete(k) = l
$k_1 < k_2 \rightarrow$ Node(k_1, v_1, l, r).delete(k_2) = Node(k_1, v_1, l, r.delete(k_2)).restore()
= Node(k_1, v_1, l.delete(k_2), r).restore()

DELMIN

Node(k, v, Nil, r).delMin() = <r, k, v>
l.delMin() := <l', k_{min}, v_{min}> $\land$ l, r ≠ Nil
$\rightarrow$ Node(k, v, l, r).delMin() = <Node(k, v, l', r).restore(), k_{min}, v_{min}>