

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

Groß-O-Notation

Motivation

Die O-Notation ist eine abstrakte Möglichkeit, die das Wachstum der Rechenzeit in Abhängigkeit von der Größe der Eingabe berechnet

- Soll von konstanten Faktoren/ unwesentlichen Termen abstrahieren
- Man definiert ein k , ab dem dies gilt und einen Faktor c für den alte Aussage gilt: $f(x) \leq c \cdot g(x) \quad x \geq k$

Master-Theorem

Erreichkrk Methode zur Bestimmung einer Rekurrenzgleichung

Bedingungen:

- $\alpha, \beta \in \mathbb{N}, \beta \geq 2, \delta \in \mathbb{R}, \delta \geq 0$
- Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}_+$ erfüllt:

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

- Fälle
- $\alpha < \beta^\delta \rightarrow f(n) \in O(n^\delta)$
 - $\alpha = \beta^\delta \rightarrow f(n) \in O(\log_\beta(n) \cdot n^\delta)$
 - $\alpha > \beta^\delta \rightarrow f(n) \in O(n^{\log_\beta(\alpha)})$

Arten der O-Notation

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

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

$$\Theta(g) := O(g) \cap \Omega(g)$$

Propositions

- Reflexivität
- Addition
- Transitiv
- Grenzwert
- Multipaktivität

Rekurrenzgleichungen

- Den Homogenen Teil lösen
 $x_n = \lambda^n$
- Inhomogenen Teil lösen
 $x_n = y$
- Einsetzen in die allgemeine Lösung

$$x_n = \alpha \cdot \lambda_1^n + \beta \cdot \lambda_2^n + y$$

! Ausnahmen !

- $\alpha + b = 1 \rightarrow x_n = y \cdot n$
- $\lambda_1 = \lambda_2 \rightarrow x_n = \alpha \cdot \lambda^n + \beta \cdot \lambda^n \cdot n + y$

Sortieren

Insertion Sort

Durchläuft Schritt für Schritt die Liste und entnimmt aus der unsortierten Eingabefolge und fügt es an der richtigen Stelle ein.

```
sort([]) = []
sort([x] + R) = insert(x, sort(R))
insert(x, []) = [x]
x ≤ y → insert(x, [y] + R) = [x] + [y] + R
¬(x ≤ y) → insert(x, [y] + R) = [y] + insert(x, R)
```

Komplexität

Bereits sortierte Liste
 $O(n)$

worst case (falsch herum sortiert)
 $\frac{1}{2} \cdot n^2 + O(n)$

average
 $\frac{1}{4} \cdot n^2 + O(n)$

Selection Sort

Es wird das Minimum aus der Liste genommen und vor die Restliste ohne das Element selbst gesetzt.

```
sort([]) = []
L ≠ [] ∧ x := min(L) → sort(L) = [x] + sort(delete(x, L))
```

Komplexität

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

Counting Sort

Counting Sort zählt die Häufigkeiten der einzelnen Elemente und fügt diese in Sublisten ein, damit ist die Liste dann sortiert.
Counting Sort ist stabil!

1. Counting-Stage
2. Indexing-Stage
3. Distribution-Stage

Komplexität:

$O(n)$ linear, da keine Vergleiche stattfinden.

Merge Sort

Merge Sort arbeitet nach dem Teile und Herrsche Prinzip. Es werden immer zwei gleich große Listen gebildet und sortiert zusammengefügt.

```
n < 2 → sort(L) = L
n ≥ 2 → sort(L) = merge(sort(L[:n//2]), sort(L[n//2:]))
merge([], L2) = L2
merge(L1, []) = L1
x ≤ y → merge([x|R1], [y|R2]) = [x] + merge(R1, [y|R2])
¬x ≤ y → merge([x|R1], [y|R2]) = [y] + merge([x|R1], R2)
```

Komplexität

Immer:

$$f(n) \in O(n \cdot \log_2(n))$$

In der Schreibweise des Master Theorems.

$$f(n) = 2 \cdot f(n/2) \cdot O(n)$$

Sortierproblem

Annahme ist, dass eine Liste L gegeben ist. Alle Elemente aus L stammen aus S . S verfügt über eine binäre Relation \leq , welche reflexiv, transitiv und anti-symmetrisch ist.

- | | |
|----------------------|--|
| Partielle
Ordnung | <ol style="list-style-type: none"> 1. $\forall x \in S : x \leq x$ 2. $\forall x, y \in S : (x \leq y \wedge y \leq x \rightarrow x = y)$ |
| Lineare
Ordnung | <ol style="list-style-type: none"> 3. $\forall x, y, z \in S : (x \leq y \wedge y \leq z \rightarrow x \leq z)$ 4. $\forall x, y \in S : (x \leq y \vee y \leq x)$ |

Quasiordnung vs. Totale Quasiordnung
• transitiv
• reflexiv
+ • linear

geg.: (M, \leq) ist in totaler-Quasiordnung und L ist eine Liste aus M .

- ges.: 1. Die Liste S ist aufsteigend sortiert:
 $\forall i \in \{0, \dots, (n(S)-2)\} : S[i] \leq S[i+1]$
2. Doppelte Elemente aus L kommen auch doppelt in S vor

Quick Sort

Man wählt ein Pivotelement und sortiert alle kleineren Elemente in die Liste S und alle größeren in die Liste R . Dies wird rekursiv wiederholt.

```
sort([]) = []
sort([x] + R) = sort([y ∈ R | y < x]) + [x] + sort([y ∈ R | y ≥ x])
```

Komplexität:

Worst-Case: $\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n \in O(n^2)$

Average Case: $2 \cdot (n/2) \cdot n \cdot (\log_2(n) + O(n)) \text{ also } O(n \cdot \log_2(n))$

Insertion Sort

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

Merge Sort

$n := \text{len}(L)$
 $n < 2 \rightarrow \text{sort}(L) = L$
 $n \geq 2 \rightarrow \text{sort}(L) = \text{merge}(L_1[:n/2], L_2[n/2:1])$
 $\text{merge}(L_1, \text{[}]) = L_1$
 $\text{merge}(\text{[}], L_2) = L_2$
 $x \leq y \rightarrow \text{merge}([x|R_1], [y|R_2]) = [x] + \text{merge}(R_1, R_2)$
 $x > y \rightarrow \text{merge}([x|R_1], [y|R_2]) = [y] + \text{merge}(R_1, R_2)$

Sortier-Algorithmen

Selection Sort

$\text{sort}(\text{[}] = \text{[}]$
 $L \neq \text{[}] \wedge x := \min(L)$
 $\rightarrow \text{sort}(L) = [x] + \text{sort}(\text{delete}(x, L))$
 $\text{delete}(x, \text{[}]) = \text{[}]$
 $\text{delete}(x, [x] + R) = R$
 $x \neq y \rightarrow \text{delete}(x, [y] + R) = \text{delete}(x, R)$

Quick Sort

$\text{sort}(\text{[}] = \text{[}]$
 $\text{sort}([x] + R) = \text{sort}([y \in R \mid y < x] + [x] + [y \in R \mid y > x])$

Map Geordnete Binärbäume

1. Nil ist ein geordneter Binärbaum
2. $\text{Node}(k, v, l, r) \in B$ g.d.w.
 - a. k ist ein Schlüssel der Menge Key
 - b. v ist ein Wert der Menge Value
 - c. l ist ein geordneter Binärbaum
 - d. r ist ein geordneter Binärbaum
 - e. l ist der linke Unterbaum von $\text{Node}(k, v, l, r)$
 - f. r ist der rechte Unterbaum von $\text{Node}(k, v, l, r)$
 - g. alle Keys in l sind kleiner k Ordnungsbedingung
 - h. alle Keys in r sind größer k
3. $\text{map}() = \text{Nil}$
4. $\text{find}: B \times \text{key} \rightarrow \text{Value} \cup \{\Omega\}$
5. $\text{insert}: B \times \text{key} \times \text{value} \rightarrow B$
6. $\text{delete}: B \times \text{key} \rightarrow B$
7. $\text{delMin}: B \rightarrow B \times \text{key} \times \text{value}$
8. FIND
 - a. $\text{Nil}.find(k) = \Omega$
 - b. $\text{Node}(k, v, l, r).find(k) = v$
 - c. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).find(k_2) = l.find(k_2)$
 - d. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).find(k_2) = r.find(k_2)$
 - e. $\text{Nil}.insert(k, v) = \text{Node}(k, v, \text{Nil}, \text{Nil})$
 - f. $\text{Node}(k_1, v_1, l, r).insert(k_2, v_2) = \text{Node}(k_1, v_1, l, \text{insert}(k_2, v_2, \text{Nil}))$
 - g. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).insert(k_2, v_2) = \text{Node}(k_2, v_2, l, \text{insert}(k_1, v_1, r))$
 - h. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).insert(k_2, v_2) = \text{Node}(k_2, v_2, l, r.insert(k_1, v_1))$
9. DELMIN
 - i. $\text{Node}(k, v, l, r).delMin() = (r, k, v)$
 - j. $l \neq \text{Nil} \wedge l.delMin() = (l', k_{min}, v_{min}) \rightarrow \text{Node}(k, v, l, r).delMin() = (\text{Node}(k, v, c', r), k_{min}, v_{min})$
 - k. $\text{Nil}.delete(k) = \text{Nil}$
 - l. $\text{Node}(k, v, l, r).remove(k) = r$
 - m. $\text{Node}(k, v, l, \text{Nil}).remove(k) = l$
 - n. $l \neq \text{Nil} \wedge r \neq \text{Nil} \wedge l.delMin() = (r', k_{min}, v_{min}) \rightarrow \text{Node}(k, v, l, r).remove(k) = \text{Node}(k_{min}, v_{min}, l, r')$
 - o. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).remove(k_2) = \text{Node}(k_2, v_2, l, \text{Node}(k_1, v_1, r))$
 - p. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).remove(k_2) = \text{Node}(k_2, v_2, l, r, \text{Node}(k_1, v_1, r))$

Komplexität
 Average Case: Logarithmisch $\mathcal{O}(\ln(n))$
 Worst Case: Linear $\mathcal{O}(n)$

ADT's / Set / Maps

ADT Stack

1. $N = \text{"Stack"}$
2. $P = \{\text{Element}\}$
3. $Fs = \{\text{Stack, push, pop, top, isEmpty}\}$
4. $Ts:$
 - a. $\text{Stack} : \text{Stack}$
 - b. $\text{push} : \text{Stack} \times \text{Element} \rightarrow \text{Stack}$
 - c. $\text{pop} : \text{Stack} \rightarrow \text{Stack}$
 - d. $\text{top} : \text{Stack} \rightarrow \text{Element}$
 - e. $\text{isEmpty} : \text{Stack} \rightarrow \mathbb{B}$
5. $\text{Ax}:$
 - a. $\text{Stack}.top() = \Omega$
 - b. $\text{S.push}(x).top() = x$
 - c. $\text{Stack}.pop() = \Omega$
 - d. $\text{S.push}(x).pop() = S$
 - e. $\text{Stack.isEmpty}() = \text{true}$
 - f. $\text{S.push}(x).\text{isEmpy} = \text{false}$

ADT Set and Maps

1. $N = \text{Map}$
2. $P = \{\text{Key, Value}\}$
3. $Fs = \{\text{map, find, insert, delete}\}$
4. $Ts:$
 - a. $\text{map} : \text{Map}$
 - b. $\text{Map} \times \text{key} \rightarrow \text{Value} \cup \{\Omega\}$
 - c. $\text{insert} : \text{Map} \times \text{key} \times \text{value} \rightarrow \text{Map}$
 - d. $\text{delete} : \text{Map} \times \text{key} \rightarrow \text{Map}$
5. $\text{Ax}:$
 - a. $\text{map}().find(k) = \Omega$
 - b. $m.insert(k, v).find(k) = v$
 - c. $k_1 \neq k_2 \rightarrow m.insert(k_1, v).find(k_2) = m.find(k_2)$
 - d. $m.delete(k).find(k) = \Omega$
 - e. $k_1 \neq k_2 \rightarrow m.delete(k_1).find(k_2) = m.find(k_2)$

Formale Definition von ADT's

$$D = \langle N, P, Fs, Ts, Ax \rangle$$

$N = \text{Name des ADT's}$
 $P = \text{Menge der Typparameter}$
 $Fs = \text{Menge der Funktionszeichen}$
 $Ts = \text{Menge der Typspezifikationen}$
 $Ax = \text{mathematische Formeln zum Spezifizieren von ADT's}$

Vorteile von ADT's

1. Austauschbar
2. Wiederverwendbar
3. Trennt die Implementierung von ADT's vom Anwendungsprogramm

Map Tries

Notation

- Σ : Endliche Menge der Buchstaben (Alphabet)
 Σ^* : Die Menge der Wörter die in mit Σ gebildet werden können.
 \mathcal{C} : leeres Wort $\mathcal{C} = \text{"}$
 Value : Menge der Werte, die den Schlüssel zugeordnet werden kann.

\mathbb{T} : Menge der Tries
 $\text{Node}: \text{Value} \times \text{List}(\Sigma) \times \text{List}(\mathbb{T}) \rightarrow \mathbb{T}$
 $\text{Node}(k, v, l, r) \in \mathbb{T}$, wenn gilt:
 a) $v \in \text{Value} \cup \{\Omega\}$
 b) $C_s = [c_1, \dots, c_n] \in \text{List}(\Sigma)$ (Eine Liste verschiedener Charaktere der Länge n)
 c) $T_s = [t_1, \dots, t_n] \in \text{List}(\mathbb{T})$ (Eine Liste von Tries derselben Länge)

$\text{find}: \mathbb{T} \times \Sigma^* \rightarrow \text{Value} \cup \{\Omega\}$
 $\text{insert}: \mathbb{T} \times \Sigma^* \times \text{Value} \rightarrow \mathbb{T}$
 $\text{isEmpty}: \mathbb{T} \rightarrow \mathbb{B}$
 $\text{delete}: \mathbb{T} \times \Sigma^* \rightarrow \mathbb{T}$

FIND
 $\text{Node}(v, cs, ts).find(c) = v$
 $\text{Node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n]).find(cr) = t_i, \text{find}(r)$
 $c \neq Cs \rightarrow \text{Node}(v, ls, Ts).find(cr) = \Omega$

INSERT
 $\text{Node}(v_1, l, T).insert(c, v_2) = \text{Node}(v_2, l, T)$
 $\text{Node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n]).insert(c, v_2) = \text{Node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n, t])$
 $c \neq Cs \rightarrow \text{Node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n, Node(\Omega, [], [])].insert(r, v_2)) = \Omega$

ISMPTY
 $\text{Node}(v, cs, Ts).isEmpty() = \text{true} \Leftrightarrow c = \Omega \wedge cs = [] \wedge Ts = []$

DELETE
 $\text{Node}(v, cs, Ts).delete(c) = \text{Node}(\Omega, Cs, Ts)$
 $t_i, \text{delete}(r).isEmpty() \rightarrow \text{Node}(v, [c_1, \dots, c_n], [t_1, \dots, t_p, \dots, C_p], [t_1, \dots, t_p, \dots, t_n])$
 $= \text{Node}(v, [c_1, \dots, c_{i-1}, c_{i+1}, \dots, C_p], [t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n])$
 $\neg t_i, \text{delete}(r).isEmpty() \rightarrow \text{Node}(v, [c_1, \dots, c_n], [t_1, \dots, t_p, \dots, t_n]).delete(c, r)$
 $= \text{Node}(v, [c_1, \dots, c_n, c], [t_1, \dots, t_p, t_i, \text{delete}(r), \dots, t_n])$
 $c \neq Cs \rightarrow \text{Node}(v, Cs, Ts).delete(cr) = \text{Node}(v, Cs, Ts)$

ADT Priority Queues

Die Prioritäten werden durch Zahlen dargestellt. Eine kleinere Zahl hat eine höhere Priorität!

1. $N = \text{ProQueue}$
2. $P = [\text{priority, value}]$
3. $Fs = \{\text{proQueue, insert, remove, top, isEmpty}\}$
4.
 - a. proQueue
 - b. $\text{insert}: \text{ProQueue} \times \text{Priority} \times \text{Value} \rightarrow \text{ProPriorityQueue}$
 - c. $\text{remove}: \text{ProPriorityQueue} \rightarrow \text{ProPriorityQueue}$
 - d. $\text{top}: \text{ProPriorityQueue} \rightarrow (\text{ProPriorityQueue} \times \text{value}) \cup \{\Omega\}$
 - e. $\text{isEmpty}: \text{ProPriorityQueue} \rightarrow \mathbb{B}$
5.
 - a. $\text{priorQueue} : \text{ProPriorityQueue}$
 - b. $\text{insert}: \text{ProPriorityQueue} \times \text{Priority} \times \text{Value} \rightarrow \text{ProPriorityQueue}$
 - c. $\text{remove}: \text{ProPriorityQueue} \rightarrow \text{ProPriorityQueue}$
 - d. $\text{top}: \text{ProPriorityQueue} \rightarrow (\text{ProPriorityQueue} \times \text{value}) \cup \{\Omega\}$
 - e. $\text{isEmpty}: \text{ProPriorityQueue} \rightarrow \mathbb{B}$

6.
 - a. TO LIST
 - a. $\text{ProQueue} \rightarrow \text{List} < \text{Priority, Value} >$
 - b. $\text{Q.isEmpty}() \rightarrow \text{Q.toList}() = []$
 - c. $\neg Q.isEmpty() \rightarrow Q.toList() = [Q.top()] + Q.remove().toList()$
 - b. INSERLIST
 - d. $\text{priority} \times \text{value} \times \text{List} < \text{Priority, Value} > \rightarrow \text{List} < \text{Priority, Value} >$
 - e. $\text{insertList}(p, v, []) = [p, v]$
 - f. $p_1 \leq p_2 \rightarrow \text{insertList}(p_1, v_1, [p_2, v_2] + R) = [p_1, v_1] + [p_2, v_2] + R$
 - g. $p_1 > p_2 \rightarrow \text{insertList}(p_1, v_1, [p_2, v_2] + R) = [p_2, v_2] + [p_1, v_1] + R$

Heap

1. $\text{Nil} \in H$
2. $\text{Node}(p, v, l, r) \in H$
 - a. $p \leq l \wedge p \leq r$
 - b. $|l.count() - r.count()| \leq 1$
 - c. $l \in H \wedge r \in H$
3. $\text{count}: H \rightarrow \mathbb{N}$
 - COUNT
 - a. $\text{Nil}.top() = 0$
 - b. $\text{Node}(p, v, l, r).count() = 1 + l.count() + r.count()$
 - TOP
 - c. $\text{Nil}.top() = \Omega$
 - d. $\text{Node}(p, v, l, r).top() = < p, v >$
 - ISEMPTY
 - e. $\text{Nil}.isEmpty() = \text{true}$
 - f. $\text{Node}(p, v, l, r).isEmpty() = \text{false}$
4.
 - INSERT
 - g. $p_{top} \leq p \wedge l.count() \leq r.count()$
 $\rightarrow \text{Node}(p_{top}, v_{top}, l, r).insert(p, v) = \text{Node}(p_{top}, v_{top}, l, \text{insert}(p, v), r)$
 - h. $p_{top} \leq p \wedge l.count() > r.count()$
 $\rightarrow \text{Node}(p_{top}, v_{top}, l, r).insert(p, v) = \text{Node}(p_{top}, v_{top}, l, r, \text{insert}(p, v))$
 - i. $p_{top} < p \wedge l.count() \leq r.count()$
 $\rightarrow \text{Node}(p_{top}, v_{top}, l, r).insert(p, v) = \text{Node}(p, v, l, \text{insert}(p_{top}, v_{top}), r)$
 - j. $p_{top} > p \wedge l.count() > r.count()$
 $\rightarrow \text{Node}(p_{top}, v_{top}, l, r).insert(p, v) = \text{Node}(p, v, l, r, \text{insert}(p_{top}, v_{top}))$

5. REMOVE
 - k. $\text{Nil}.remove() = \text{Nil}$
 - l. $\text{Node}(p, v, l, r).remove() = l$
 - m. $\text{Node}(p, v, l, \text{Nil}).remove() = l$
 - n. $l \neq \text{Nil} \wedge r \neq \text{Nil} \wedge (r', k_{min}, v_{min}) := r, \text{delMin}()$
 $\rightarrow \text{Node}(p, v, l, r).remove() = \text{Node}(p, v, l, \text{delMin}(), k_{min}, v_{min})$
 - o. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).remove() = \text{Node}(k_2, v_2, l, \text{insert}(k_1, v_1, r)).remove()$
 - p. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).remove() = \text{Node}(k_2, v_2, l, r, \text{insert}(k_1, v_1)).remove()$

AVL-Bäume

- Höhenbedingung
 1. $\text{Nil}.height() = 0$
 2. $\text{Node}(k, v, l, r).height() = \max(l.height(), r.height()) + 1$
- Menge A der AVL Bäume
 1. $\text{Nil} \in A$
 2. $\text{Node}(k, v, l, r) \in A$ genau dann, wenn:
 - a. $\text{Node}(k, v, l, r) \in B$
 - b. $l, r \in A$
 - c. $|l.height() - r.height()| \leq 1$
- a. $\text{restore}: B \rightarrow A$
 - b. $\text{insert}: B \times \text{key} \times \text{value} \rightarrow B$
 - c. $\text{find}: B \times \text{key} \rightarrow \text{value} \cup \{\Omega\}$
 - d. $\text{delete}: B \times \text{key} \rightarrow B$
 - e. $\text{delMin}: B \rightarrow B \times \text{key} \times \text{value}$
- FIND
 - a. $\text{Nil}.find(k) = \Omega$
 - b. $\text{Node}(k, v, l, r).find(k) = v$
 - c. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).find(k_2) = l.find(k_2)$
 - d. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).find(k_2) = r.find(k_2)$
 - e. $\text{Nil}.insert(k, v) = \text{Node}(k, v, \text{Nil}, \text{Nil})$
 - f. $\text{Node}(k, v, l, r).insert(k, v) = \text{Node}(k, v_1, l, r)$
 - g. $k_1 < k_2 \rightarrow \text{Node}(k_1, v_1, l, r).insert(k_2, v_2) = \text{Node}(k_2, v_2, l, \text{insert}(k_1, v_1, r)).restore()$
 - h. $k_1 > k_2 \rightarrow \text{Node}(k_1, v_1, l, r).insert(k_2, v_2) = \text{Node}(k_2, v_2, l, r, \text{insert}(k_1, v_1)).restore()$
 - i. $\text{Node}(k, v, Nil, r).delMin() = (r, k, v)$
 - j. $l \neq \text{Nil} \wedge r \neq \text{Nil} \wedge (r', k_{min}, v_{min}) := l, \text{delMin}()$
 $\rightarrow \text{Node}(k, v, l, r).delMin() = (\text{Node}(k, v, l, r).delMin(), k_{min}, v_{min})$
 - DELETE
 - k. $\text{Nil}.delete(k) = \text{Nil}$
 - l. $\text{Node}(k, v, l, r).delete(k) = r$
 - m. $\text{Node}(k, v, l, \text{Nil}).delete(k) = l$
 - n. $l \neq \text{Nil} \wedge r \neq \text{Nil} \wedge (r', k_{$