

Kapitel 2: Groß-O-Notation

Dient zur Laufzeitanalyse eines Algorithmus.

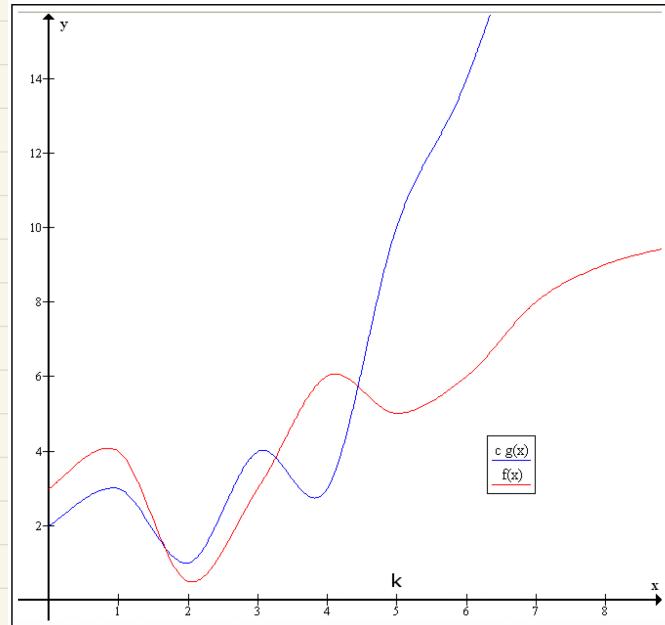
$$O(g) := \{ f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq k \rightarrow f(n) \leq c \cdot g(n)) \}$$

Transitivität der O-Notation

$$f \in O(g)$$

Beweis: $k := 0$ $c := 1$

$$\forall n \in \mathbb{N} : f(n) \leq g(n) \quad \checkmark$$



Multiplikativität der O-Notation

$$f, g : \mathbb{N} \rightarrow \mathbb{R}_+, \quad d \in \mathbb{R}_+$$

$$g \in O(f) \rightarrow d \cdot g \in O(f)$$

Beweis

$g \in O(f)$ impliziert, dass es die Konstanten $c' \in \mathbb{R}_+$ und $k' \in \mathbb{N}$, so dass

$$\forall n \in \mathbb{N} : (n \geq k' \rightarrow g(n) \leq c' \cdot f(n))$$

Multiplizieren von $g(n)$ mit d

$$\forall n \in \mathbb{N} : (n \geq k' \rightarrow d \cdot g(n) \leq d \cdot c' \cdot f(n))$$

Definieren von $k := k'$ und $c := d \cdot c'$

$$\forall n \in \mathbb{N} : (n \geq k \rightarrow d \cdot g(n) \leq c \cdot f(n))$$

Dies impliziert nach Definition

$$d \cdot g \in O(f)$$

□

Addition der \mathcal{O} -Notation

$f, g, h : \mathbb{N} \rightarrow \mathbb{R}_+$

$$f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h) \rightarrow f+g \in \mathcal{O}(h)$$

Beweis

$f \in \mathcal{O}(h)$ und $g \in \mathcal{O}(h)$ impliziert $k_1, k_2 \in \mathbb{N}$ und $c_1, c_2 \in \mathbb{R}$

$\forall n \in \mathbb{N} : (n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n))$ and

$\forall n \in \mathbb{N} : (n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n))$

Definiere von $k := \max(k_1, k_2)$ und $c := c_1 + c_2$ für $n \in \mathbb{N}$

$$f(n) \leq c_1 \cdot h(n) \text{ und } g(n) \leq c_2 \cdot h(n)$$

Addieren der Ungleichungen

$$f(n) + g(n) \leq (c_1 + c_2) \cdot h(n) = c \cdot h(n)$$

□

Transitivität der \mathcal{O} -Notation

$f, g, h : \mathbb{N} \rightarrow \mathbb{R}_+$

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \rightarrow f \in \mathcal{O}(h)$$

Beweis

$f \in \mathcal{O}(g)$ impliziert $k_1 \in \mathbb{N}$ $c_1 \in \mathbb{R}$

$\forall n \in \mathbb{N} : (n \geq k_1 \rightarrow f(n) \leq c_1 \cdot g(n))$

$g \in \mathcal{O}(h)$ impliziert $k_2 \in \mathbb{N}$ $c_2 \in \mathbb{R}$

$\forall n \in \mathbb{N} : (n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n))$

Definiere $k := \max(k_1, k_2)$ $c := c_1 \cdot c_2$ $n \in \mathbb{N}$

$$f(n) \leq c_1 \cdot g(n) \text{ and } g(n) \leq c_2 \cdot h(n)$$

Multiplizieren der zweiten Gleichung mit c_1

$$f(n) \leq c_1 \cdot g(n) \text{ und } c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Das führt zu

$$f(n) \leq c \cdot h(n) \quad \text{für } n \geq k$$
$$\Rightarrow f \in O(h)$$

Grenzwertsatz der O -Notation

$$f, g : \mathbb{N} \rightarrow \mathbb{R}_+$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \quad \text{daraus folgt } f \in O(g)$$

Beweis

$$\lambda := \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

Nach dem Grenzwertsatz:

$$\forall \varepsilon \in \mathbb{R}_+ : \exists k \in \mathbb{R} : \exists n \in \mathbb{N} : (n \geq k \rightarrow \left| \frac{f(n)}{g(n)} - \lambda \right| < \varepsilon)$$

$$\varepsilon := 1$$

$$\left| \frac{f(n)}{g(n)} - \lambda \right| \leq 1$$

Multiplizieren mit $g(n)$

$$|f(n) - \lambda \cdot g(n)| \leq g(n)$$

Durch die Dreiecksungleichung

$$|a+b| \leq |a| + |b|$$

$$f(n) = |f(n)| = |f(n) - \lambda \cdot g(n) + \lambda \cdot g(n)| \leq |f(n) - \lambda \cdot g(n)| + \lambda \cdot g(n)$$

Man definiert $C: 1+\lambda$

und zeigt somit, dass $f(n) \leq C \cdot g(n)$ gilt für $n \geq h$

Second Order Linear Inhomogeneous Recurrence Relation

Die Form: $X_{n+2} = a \cdot X_{n+1} + b \cdot X_n + c$

$$X_0 = d \quad X_1 = e$$

3 Schritte zur Lösung:

1. Homogenen Teil lösen:

$$X_{n+2} = a \cdot X_{n+1} + b \cdot X_n$$

$$X_n = \lambda^n$$

$$\lambda^{n+2} = a \cdot \lambda^{n+1} + b \cdot \lambda^n$$

λ_1 & λ_2 berechnen

1. Fall $\lambda_1 \neq \lambda_2$:

$$X_n = \alpha \cdot \lambda_1^n + \beta \cdot \lambda_2^n + f$$

2. Fall $\lambda_1 = \lambda_2$:

$$X_n = \alpha \cdot \lambda^n + \beta \cdot n \cdot \lambda^n + f$$

2. Inhomogener Teil lösen:

$$X_{n+2} = a \cdot X_{n+1} + b \cdot X_n + c$$

$$X_n = f \quad (\text{wenn } a+b \neq 1) \quad \text{sonst } X_n = f \cdot n$$

$$\hookrightarrow f = a \cdot f + b \cdot f + c$$

f bestimmen

3. Generelle Lösung bestimmen

$$X_n = \alpha \cdot \lambda_1^n + \beta \cdot \lambda_2^n + f$$

α & β bestimmen unter Berücksichtigung der Anfangsbedingungen.

Insertion Sort

Kapitel 3: Sorting

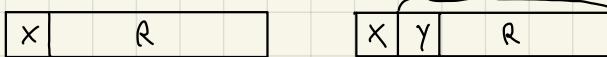
1. Wenn eine leere Liste sortiert werden soll, dann wird diese einfach zurückgegeben.

2. Ansonsten wird die Liste L aufgeteilt in X und R . X ist das erste Element aus L und R alles ab dem ersten.

$$\hookrightarrow X = L[0], R = L[1:]$$

$$\hookrightarrow X + R = L$$

3. Jetzt wird $\text{insert}(x, L)$ mit $(x, \text{sort}(R))$ aufgerufen.



Aus der Restliste R wird nun das erste Element in y hinzugefügt und die Restliste R zugezogen.

4. x und y verglichen:

a) $x \leq y$

$$\text{return } [x] + L$$

andererseits:

$$\text{return } [y] + \text{insert}(x, R)$$

Komplexität

Worst-Case : Falschrum sortierte Liste:

$$O(n^2)$$

Best-Case : Korrekt sortierte Liste:

$$O(n)$$

```
1 def sort(L):
2     if L == []:
3         return []
4     x, R = L[0], L[1:]
5     return insert(x, sort(R))
6
7 def insert(x, L):
8     if L == []:
9         return [x]
10    y, R = L[0], L[1:]
11    if x <= y:
12        return [x] + L
13    else:
14        return [y] + insert(x, R)
```

Bei teilweise sortierten Listen
aber sehr effektiv

Selection Sort

1. Wenn die leere Liste sortiert werden

soll, dann wird diese zurück gegeben.

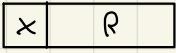
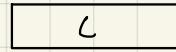
2. Wenn L nicht leer ist, dann wird

$x = \min(L)$ ausgewählt und aus

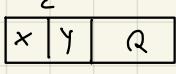
L gelöscht. Dann wird dieses Element vor die sortierte Liste gesetzt (rekursiv) $\rightarrow [x] + \text{sort}(\text{delete}(x, L))$

3. Das Minimum vor der Rekursiv sortierte Liste setzen.

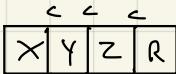
In der Rekursion wird immer wieder das Minimum aus der Restliste gezogen und entfernt, bis



x min von L



y min R



z min R

Komplexität von Selection Sort

$$\frac{1}{2} \cdot n \cdot (n-1) + O(n^2) \quad \text{Vergleiche}$$

```

1 def sort(L):
2     if L == []:
3         return []
4     x = min(L)
5     return [x] + sort(delete(x, L))
6
7 def delete(x, L):
8     if L == []:
9         assert L != [], f'delete({x}, [])'
10    if L[0] == x:
11        return L[1:]
12    return [L[0]] + delete(x, L[1:])

```

Merge Sort

1. Wenn nur ein Element in der Liste ist, dann wird die Liste einfach zurückgegeben

2. Sonst wird die Liste in zwei gleichgroße Teillisten L_1, L_2 aufgeteilt. Diese werden dann rekursiv in $\text{merge}(\text{sort}(L_1), \text{sort}(L_2))$ aufgerufen.

3. Wenn L_1 oder L_2 leer sind, dann wird die jeweils andere Liste zurückgegeben.

4. L_1 und L_2 werden aufgeteilt in jeweils das erste Element von L_1/L_2 und den Rest.

5. Durch den Vergleich von x_1 und x_2 werden die beiden Listen L_1/L_2 'zusammengefertigt' und so zu einer sortierten Liste ausgegeben.

```
1 def sort(L):
2     n = len(L)
3     if n < 2:
4         return L
5     L1, L2 = L[:n//2], L[n//2:]
6     return merge(sort(L1), sort(L2))
7
8 def merge(L1, L2):
9     if L1 == []:
10        return L2
11    if L2 == []:
12        return L1
13    x1, R1 = L1[0], L1[1:]
14    x2, R2 = L2[0], L2[1:]
15    if x1 <= x2:
16        return [x1] + merge(R1, L2)
17    else:
18        return [x2] + merge(L1, R2)
```

Komplexität von Merge Sort

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

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

Quicksort

```
1 def sort(L):
2     if L == []:
3         return L
4     x, R = L[0], L[1:]
5     S = [y for y in R if y <= x]
6     B = [y for y in R if y > x]
7     return sort(S) + [x] + sort(B)
```

1. Wenn L eine leere Liste

ist, dann wird die leere Liste zurück gegeben.

2. L wird in x und R aufgeteilt.

x ist das erste Element der Liste und R der Rest
 $x = L[0]$ | $R = L[1:]$

3. Nun werden die Listen S (small) und B (big) gebildet.

Die Elemente in R werden einzeln mit x verglichen.

Alle Elemente kleiner/gleich x werden in S hinzugefügt und alle Elemente größer als x werden in B hinzugefügt.

4. Die Schritte werden rekursiv für S und B ausgeführt.

Komplexität von Quicksort

Worst case :

$$O(n^2)$$

ist 39% langsamer als Mergesort

best / average Case:

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

Counting Sort

Counting Sort vergleicht alle Elemente nach, daher ist seine Komplexität linear mit $O(n)$

Counting Sort durchläuft drei Schritte, um eine Sortierung zu erreichen.

```
1 def countingSort(Names, Grades):
2     assert len(Names) == len(Grades)
3     maxGrade      = 256
4     Counts        = [0] * maxGrade
5     Indices       = [None] * maxGrade
6     SortedNames   = [None] * len(Names)
7     SortedGrades  = [None] * len(Names)
8
9     # Phase 1: Counting
10    for g in Grades:
11        Counts[g] += 1
12
13    # Phase 2: Indexing
14    Indices[0] = 0
15
16    for g in range(1, maxGrade):
17        Indices[g] = Indices[g-1] + Counts[g-1]
18
19    # Phase 3: Distribution
20
21    for i in range(len(Names)):
22        grade          = Grades[i]
23        idx            = Indices[grade]
24        SortedNames[idx] = Names[i]
25        SortedGrades[idx] = Grades[i]
26        Indices[grade] += 1
27
28    return SortedNames, SortedGrades
```

1) Counting-Stage:

Die Häufigkeit von den vorkommenden Elementen wird bestimmt.

Bspw. Element $x : 10$ (kommt 10x vor)

2) Indexing-Stage:

Durch die Erkenntnis, wie häufig ein Element vorkommt, werden Sublisten in Abhängigkeit der Häufigkeiten erstellt, welche dann verkettet werden und so eine sortierte CSK ergeben.

3) Distribution-Stage:

Man iteriert über die List und fügt die Elemente in den passenden Sublisten ein.

Radix Sort

Man will eine Liste von 32-Bit Zahlen sortieren.

Eine 32 bit Zahl lässt sich wie folgt darstellen:

$$x = b_4 \cdot 256^3 + b_3 \cdot 256^2 + b_2 \cdot 256^1 + b_1 \cdot 256^0$$

Radix Sort verwendet counting sort und besteht aus vier Schritten:

1. Sortieren der Liste nach dem untersten Byte b_1
2. Sortieren nach b_2
3. Sortieren nach b_3
4. Sortieren nach b_4

Da counting sort stabiles ist, kann man so vorgehen.

```
1 def extractByte(n, k):
2     return n >> (8 * (k-1)) & 255
3
4 def radixSort(L):
5     for k in range(1, 4+1):
6         Grades = [extractByte(n, k) for n in L]
7         L      = countingSort(L, Grades)[0]
8
9     return L
```

3. Check your understanding

1) Lineare Ordnung Definition:

Das Paar $\langle S, \leq \rangle$ mit $\leq \subseteq S \times S$ und

1. $\forall x \in S : x \leq x$ (Reflexiv)

2. $\forall x, y \in S : (x \leq y \wedge y \leq x \rightarrow x = y)$ (Anti-Symmetrie)

3. $\forall x, y, z \in S : (x \leq y \wedge y \leq z \rightarrow x \leq z)$ (Transitiv)

4. $\forall x, y \in S : (x \leq y \vee y \leq x)$ (Linearität)

} Partielle
Ordnung
} Lineare
Ordnung

2) Welche Ordnung benötigen wir, um eine Liste zu sortieren?

Eine totale Quasiordnung wird benötigt.

1. $\forall x \in S : (x \leq x)$ (Reflexiv)

2. $\forall x, y, z \in S : (x \leq y \wedge y \leq z \rightarrow x \leq z)$ (Transitiv)

3. $\forall x, y \in S : (x \leq y \vee y \leq x)$ (Linearität)

} Quasiordnung

} Totale
Quasiordnung

3) Insertion Sort auf einem abstrakten mathematischen Level:

1. Fall: $\text{sort}([I]) = [I]$

2. Fall: $\text{sort}([x] + R)$

a) $\text{insert}(x, [I]) = [x] + I$

b) $\text{insert}(x, [y] + R) \stackrel{x \leq y}{=} [x] + [y] + R$

$$\begin{aligned} &= [y] + \text{insert}(x, R) \\ &\quad \vdots(x \leq y) \end{aligned}$$

4) Die Komplexität von Insertion Sort:

$a_n := \#\text{Vgl. für } L \text{ mit } \text{len}(L) = n$

\rightarrow Best Case: L sortiert $\Rightarrow a_n := n - 1 \Rightarrow$ Linear kompl. $O(n)$

\rightarrow Schlechtester Fall: L absteigend sort. $\Rightarrow a_n := \frac{1}{2}n^2 + O(n) \Rightarrow$ exponentiell $O(n^2)$

5) Gibt es einen Fall, in dem die Komplexität von Insertion Sort linear ist?

Ja, wenn die gegebene Liste bereits sortiert ist.

6) Abstrakte mathematische Beschreibung von Selection Sort:

$$1. \text{sort}(\emptyset) = \emptyset$$

$$2. L \neq \emptyset \wedge x := \min(L) \rightarrow \text{sort}(L) = [x] + \text{sort}(L \setminus \{x\})$$

$$\cdot \min(\emptyset) = \infty \quad (\because L \neq \emptyset)$$

$$\cdot \min([x] + R) = \min(x, \min(R)) \Rightarrow \min(x, y) := \begin{cases} x & \text{wenn } x \leq y \\ y & \text{sonst} \end{cases}$$

$$\cdot \text{delete}(x, \emptyset) = \emptyset$$

$$\cdot \text{delete}(x, [x] + R) = R$$

$$\cdot x \neq y \rightarrow \text{delete}(x, [y] + R) = [y] + \text{delete}(x, R)$$

Kapitel 4: Abstrakte Datentypen (ADT's)

Die Notation eines ADT abstrahiert von den tatsächlichen Implementierungsdetails einer konkreten Datenstruktur.

ADT

Die formale Definition eines ADT D wird als 5-Tupel dargestellt.

$$D = \langle N, P, F_S, T_S, A_X \rangle$$

1. N ist ein String. N ist der Name des ADT's

2. P ist Menge der Typ-Parameter.

3. F_S ist Menge der Funktionszeichen

4. T_S ist Menge der Typspezifikationen

für jedes $f \in F_S$ gibt es in T_S genau eine Typspezifikation.

$$f: T_1 \times \dots \times T_n \rightarrow S$$

5. A_X ist Menge von mathematischen Formeln. Diese spezifizieren den ADT

ADT Stack

1. $\mathcal{V} = \text{"Stack"}$
2. $\mathcal{P} = \{\text{Element}\}$
3. $\mathcal{F}_S = \{\text{Stack, push, pop, top, isEmpty}\}$
4. $\mathcal{T}_S :$

$\text{Stack} : \text{Stack}$

$\text{push} : \text{Stack} \times \text{Element} \rightarrow \text{Stack}$

$\text{pop} : \text{Stack} \rightarrow \text{Stack}$

$\text{top} : \text{Stack} \rightarrow \text{Element}$

$\text{isEmpty} : \text{Stack} \rightarrow \mathbb{B}$

5. $\mathcal{A}_x :$

1. $\text{Stack}(). \text{top}() = \varnothing$
2. $S. \text{push}(x). \text{top}() = x$
3. $\text{Stack}(). \text{pop}() = \varnothing$
4. $S. \text{push}(x). \text{pop}() = S$
5. $\text{Stack}(). \text{isEmpty}() = \text{true}$
6. $S. \text{push}(x). \text{isEmpty} = \text{false}$

Generatoren sind die Funktionszeichen, mit denen man jeden möglichen Zustand erreichen kann (bei Stack: Stack und push)

Stack implementiert

```
1  class Stack:
2      def __init__(self):
3          self.mStackElements = []
4
5      def push(self, e):
6          self.mStackElements.append(e)
7
8      def pop(self):
9          assert len(self.mStackElements) > 0, "popping empty stack"
10         self.mStackElements = self.mStackElements[:-1]
11
12     def top(self):
13         assert len(self.mStackElements) > 0, "top of empty stack"
14         return self.mStackElements[-1]
15
16     def isEmpty(self):
17         return self.mStackElements == []
18
19     def copy(self):
20         C = Stack()
21         C.mStackElements = self.mStackElements[:]
22         return C
23
24     def __str__(self):
25         C = self.copy()
26         result = C._convert()
27         dashes = "-" * len(result)
28         return '\n'.join([dashes, result, dashes])
29
30     def _convert(self):
31         if self.isEmpty():
32             return '|'
33             t = self.top()
34             self.pop()
35             return self._convert() + ' ' + str(t) + ' |'
36
37     def createStack(L):
38         S = Stack()
39         n = len(L)
40         for i in range(n):
41             S.push(L[i])
42             print(S)
43         return S
44
45     createStack(range(10))
```

Vorteile von ADT's :

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

The Shunting-Yard - Algorithmus

Eine Formel wird rückwärts in eine Liste der einzelnen Element umgewandelt

$$4 - 3 * 2$$

$$\Rightarrow [2, '*', 3, '-', 4]$$

Präzedenz	Operator
1	+, -
2	*, /
3	**

Die Funktion eval : String \rightarrow IN soll einen arithmetischen Ausdruck auswerten.

Shunting-Yard verwackt drei Stacks:

TS: Token Stack, enthält die Eingabesymbole

$$"1 + 2 * 3" \quad TS = [3, '*', 2, '+', 1]$$

AS: Argumentstack, enthält Zahlen, anfangs leer

Zahlen aus dem TS auf den AS.

OS: Operator-Stack, enthält Operatoren, als zehn

'('

Vorgang

1. Lese phase:

Die Tokens werden vom Tokenstack entweder auf den Argumentstack oder Operatorstack aufgeteilt. Es können bereits Operatoren evaluiert werden während dieser Phase.

2. Evaluierungs Phase.

Die Operatoren werden evaluiert.

- Element vom OP Stack nehmen
- Element vom AS Stack nehmen
- Das Ergebnis bestimmen
- Das Ergebnis auf den AS legen

Implementierung von Shunting-Yard

```
1 def evaluate(self):
2     while not self.mTokens.isEmpty():
3         nextOp = self.mTokens.top(); self.mTokens.pop()
4         if isinstance(nextOp, int):
5             self.mArguments.push(nextOp)
6             continue
7         if self.mOperators.isEmpty():
8             self.mOperators.push(nextOp)
9             continue
10        if nextOp == "(":
11            self.mOperators.push(nextOp)
12            continue
13        stackOp = self.mOperators.top()
14        if stackOp == "(" and nextOp == ")":
15            self.mOperators.pop()
16            continue
17        if nextOp == ")":
18            self.popAndEvaluate()
19            self.mTokens.push(nextOp)
20            continue
21        if stackOp == '(':
22            self.mOperators.push(nextOp)
23            continue
24        if evalBefore(stackOp, nextOp):
25            self.popAndEvaluate()
26            self.mTokens.push(nextOp)
27        else:
28            self.mOperators.push(nextOp)
29    while not self.mOperators.isEmpty():
30        self.popAndEvaluate()
31    return self.mArguments.top()
```

```
1 def popAndEvaluate(self):
2     rhs = self.mArguments.top(); self.mArguments.pop()
3     lhs = self.mArguments.top(); self.mArguments.pop()
4     op = self.mOperators.top(); self.mOperators.pop()
5     result = None
6     if op == '+':
7         result = lhs + rhs
8     if op == '-':
9         result = lhs - rhs
10    if op == '*':
11        result = lhs * rhs
12    if op == '/':
13        result = lhs // rhs
14    if op == '%':
15        result = lhs % rhs
16    if op == '**':
17        result = lhs ** rhs
18    self.mArguments.push(result)
```

Kapitel 5: Set and Maps

Funktionsale Relationen sind Maps. In Python Dictionaries.

Der abstrakte Datentyp Map

$$D = \langle N, P, F_S, T_S, A_T \rangle$$

1. $N = \text{Map}$

2. $P = \{ \text{Key}, \text{Value} \}$

3. $F_S = \{ \text{map}, \text{find}, \text{insert}, \text{delete} \}$

4. $T_S :$

a) $\text{Map} : \text{Map}$

b) $\text{find} : \text{Map} \times \text{Key} \rightarrow \text{Value} \cup \{\emptyset\}$

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. $A_T :$

a) $\text{map().find}(k) = \emptyset$

b) $m.\text{insert}(k, v).\text{find}(k) = v$

c) $k_1 \neq k_2 \rightarrow m.\text{insert}(k_1, v).\text{find}(k_2) = m.\text{find}(k_2)$

d) $m.\text{delete}(k).\text{find}(k) = \emptyset$

e) $k_1 \neq k_2 \rightarrow m.\text{delete}(k_1).\text{find}(k_2) = m.\text{find}(k_2)$

```

1 class Map:
2     def __init__(self):
3         self.mDictionary = {}
4
5     def find(self, k):
6         return self.mDictionary[k]
7
8     def insert(self, k, v):
9         self.mDictionary[k] = v
10
11    def delete(self, k):
12        del self.mDictionary[k]
13
14    def __str__(self):
15        return str(self.mDictionary)

```

Geordnete Binär Bäume

\mathcal{B} Menge der geordneten Binärbäume

1. Nil ist ein geordneter Binärbaum

2. Node $(k, v, \ell, r) \in \mathcal{B}$ 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
 - ℓ ist das linke Kinderbaum von Node (k, v, ℓ, r)
 - d) r ist ein geordneter Binärbaum
 - r ist das rechte Kinderbaum von Node (k, v, ℓ, r)
 - e) alle Keys in ℓ sind kleiner k
 - f) alle Keys in r sind größer k
- } Ordnungsbeziehung

Komplexität

Average case:

Logarithmisch $O(\log(n))$

Worst case

Linear $O(n)$

- 3.
- a) $\text{map}() = \text{Nil}$
 - b) $\text{find} : \mathcal{B} \times \text{Key} \rightarrow \text{Value} \cup \{\text{err}\}$
 - c) $\text{insert} : \mathcal{B} \times \text{Key} \times \text{Value} \rightarrow \mathcal{B}$
 - d) $\text{delete} : \mathcal{B} \times \text{Key} \rightarrow \mathcal{B}$
 - e) $\text{deMin} : \mathcal{B} \rightarrow \mathcal{B} \times \text{Key} \times \text{Value}$

- 4.
- a) $\text{Nil}. \text{find}(k) = \text{err}$
 - b) $\text{Node}(k_1, v_1, \ell_1, r_1). \text{find}(k) = v$
 - c) $k_1 < k_2 \rightarrow \text{Node}(k_2, v_2, \ell_2, r_2). \text{find}(k_1) = \ell. \text{find}(k_1)$
 - d) $k_1 > k_2 \rightarrow \text{Node}(k_2, v_2, \ell_2, r_2). \text{find}(k_1) = r. \text{find}(k_1)$
 - e) $\text{Nil}. \text{insert}(k, v) = \text{Node}(k, v, \text{Nil}, \text{Nil})$
 - f) $\text{Node}(k_2, v_2, \ell_2, r_2). \text{insert}(k_1, v_1) = \text{Node}(k_1, v_1, \ell_2, r_2)$
 - g) $k_1 < k_2 \rightarrow \text{Node}(k_2, v_2, \ell_2, r_2). \text{insert}(k_1, v_1) = \text{Node}(k_2, v_2, \ell. \text{insert}(k_1, v_1), r)$
 - h) $k_1 > k_2 \rightarrow \text{Node}(k_2, v_2, \ell_2, r_2). \text{insert}(k_1, v_1) = \text{Node}(k_2, v_2, \ell, r. \text{insert}(k_1, v_1))$
 - i) $\text{Node}(k_1, v_1, \text{Nil}, \text{Nil}). \text{deMin}() = \langle r, k, v \rangle$
 - j) $\ell \neq \text{Nil} \wedge \ell. \text{deMin}() = \langle \ell', k_{\min}, v_{\min} \rangle \rightarrow$
 $\text{Node}(k_1, v_1, \ell, r). \text{deMin}() = \langle \text{Node}(k_1, v_1, \ell', r), k_{\min}, v_{\min} \rangle$

l) $\text{Nil}. \text{delete}(k) = \text{Nil}$

m) $\text{Node}(k_1, v_1, \text{Nil}, r). \text{remove}(k) = r$

n) $\text{Node}(k_1, v_1, \ell, \text{Nil}). \text{remove}(k) = \ell$

o) $\ell \neq \text{Nil} \wedge r \neq \text{Nil} \wedge \ell. \text{deMin}() = \langle r', k_{\min}, v_{\min} \rangle \rightarrow$
 $\text{Node}(k_1, v_1, \ell, r). \text{delete}(k) = \text{Node}(k_{\min}, v_{\min}, \ell, r')$

p) $k_1 < k_2 \rightarrow \text{Node}(k_2, v_2, \ell, r). \text{delete}(k_1) = \text{Node}(k_2, v_2, \ell. \text{delete}(k_1), r)$

AUC Bäume

Die Komplexität ist immer $O(n(n))$

Höhenbeleingeneq:

1. $\text{Nil}. \text{height}() = 0$
2. $\text{Node}(u, v, l, r). \text{height}() = \max(l. \text{height}(), r. \text{height}()) + 1$ (maximale Differenz darf 1 sein)

Menge A der AUC Bäume

1. $\text{Nil} \in A$
2. $\text{Node}(u, v, l, r) \in A$ gelten.
 - a) $\text{Node}(u, v, l, r) \in B$ (Binary Trees)
 - b) $l, r \in A$
 - c) $|l. \text{height}() - r. \text{height}()| \leq 1$ (Balancierungs Beleingeneq)

3. $\text{restore} : B \rightarrow A$
 - $\text{insert} : B \times \text{Key} \times \text{Value} \rightarrow B$
 - $\text{find} : B \times \text{Key} \rightarrow \text{Value} \cup \{\text{?}\}$
 - $\text{delele} : B \times \text{Key} \rightarrow B$
 - $\text{delelin} : B \rightarrow B \times \text{Key} \times \text{Value}$

Restore wird nicht spezifiziert, da wir es auch in der Vorschlag nicht gemacht haben.

4.
 - a) $\text{Nil}. \text{find}(u) = ?$
 - b) $\text{Node}(u, v, l, r). \text{find}(u) = v$
 - c) $u_1 < u_2 \rightarrow \text{Node}(u_2, v, l, r). \text{find}(u_1) = l. \text{find}(u_1)$
 - d) $u_1 > u_2 \rightarrow \text{Node}(u_2, v, l, r). \text{find}(u_1) = r. \text{find}(u_1)$

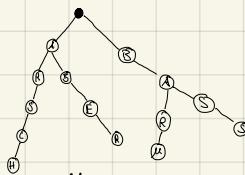
Vorbesseungssideen:

- Rot-Schwarz-Bäume
- 2-3-Bäume
- 2-3-4 Bäume

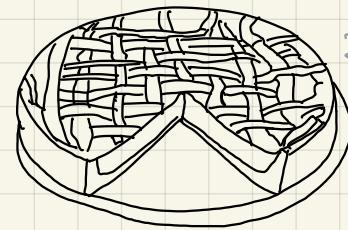
- e) $\text{Nil}. \text{insert}(u, v) = \text{Node}(u, v, \text{Nil}, \text{Nil})$
- f) $\text{Node}(u, v_2, l, r). \text{insert}(u, v_1) = \text{Node}(u, v_1, l, r)$
- g) $u_1 < u_2 \rightarrow \text{Node}(u_2, v_2, l, r). \text{insert}(u_1, v_1) = \text{Node}(u_2, v_2, l. \text{insert}(u_1, v_1), r). \text{restore}()$
- h) $u_1 > u_2 \rightarrow \text{Node}(u_2, v_2, l, r). \text{insert}(u_1, v_1) = \text{Node}(u_2, v_2, l, r. \text{insert}(u_1, v_1)). \text{restore}()$
- i) $\text{Node}(u, v, \text{Nil}, r). \text{delelin}() = \langle r, u, v \rangle$ c) Linker Teilbaum ohne u_{\min}, v_{\min} in dem Teilbaum
- j) $l \neq \text{Nil} \wedge \langle l', u_{\min}, v_{\min} \rangle := l. \text{delelin}() \rightarrow$
 $\text{Node}(u, v, l, r). \text{delelin}() = \langle \text{Node}(u, v, l; r). \text{restore}(), u_{\min}, v_{\min} \rangle$

- k) $\text{Nil}. \text{delele}(u) = \text{Nil}$
- l) $\text{Node}(u, v, \text{Nil}, r). \text{delele}(u) = r$
- m) $\text{Node}(u, v, l, \text{Nil}). \text{delele}(u) = l$
- n) $l \neq \text{Nil} \wedge r \neq \text{Nil} \wedge \langle r', u_{\min}, v_{\min} \rangle := r. \text{delelin}() \rightarrow$
 $\text{Node}(u, v, l, r). \text{delele}(u) = \text{Node}(u_{\min}, v_{\min}, l, r'). \text{restore}()$
- o) $u_1 < u_2 \rightarrow \text{Node}(u_2, v_2, l, r). \text{delele}(u_1) = \text{Node}(u_2, v_2, l. \text{delele}(u_1), r). \text{restore}()$
- p) $u_1 > u_2 \rightarrow \text{Node}(u_2, v_2, l, r). \text{delele}(u_1) = \text{Node}(u_2, v_2, l, r. \text{delele}(u_1)). \text{restore}()$

Tries



In einem Trie kann jeder Knoten SO VIELE Kinder haben, wie Buchstaben im Alphabet zur Verfügung stehen



Don't forget: pronounce
trie so that it
rhymes with pie!

Notation:

Σ : Endliche Menge der Buchstaben, Alphabet

Σ^* : Die Menge der Wörter die mit Σ gebildet werden können.

ϵ : leeres Wort $\epsilon = ''$

Value : Menge aller Wörter, die den Schlüsseln zugeordnet werden kann

Π : Menge der Tries

Node : Value \times List(Σ) \times List(Π) $\rightarrow \Pi$

Node(v, cs, ts) $\in \Pi$ g.d.w. :

(a) $v \in \text{Value} \cup \{\varnothing\}$

(b) $cs = [c_1, \dots, c_n] \in \text{List}(\Sigma)$ (Eine Liste verschiedener charaktere der Länge n)

(c) $ts = [t_1, \dots, t_n] \in \text{List}(\Pi)$ (Eine List von Tries der selben Länge)

find : $\Pi \times \Sigma^* \rightarrow \text{Value} \cup \{\varnothing\}$

insert : $\Pi \times \Sigma^* \times \text{Value} \rightarrow \Pi$

isEmpty : $\Pi \rightarrow \mathbb{B}$

delete : $\Pi \times \Sigma^* \rightarrow \Pi$

Node(v, cs, ts). find(ϵ) = v

Node($v, [c_1, \dots, c_n], [t_1, \dots, t_n]$). find(c_r) = t_i . find(r)

$c \notin cs \rightarrow \text{Node}(v, cs, ts). \text{find}(cr) = \varnothing$

Node(v_1, l, T). insert(E, v_2) = Node(v_2, l, T)

Node($v_1, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i, \dots, t_n]$). insert(r, v_2) = $[t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n]$

= Node($v_1, [c_1, \dots, c_{i-1}, c_{i+1}, c_n], [t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n]$)

$c \notin \{c_1, \dots, c_n\} \rightarrow \text{Node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n]). \text{insert}(cr, v_2)$

= Node($v_1, [c_1, \dots, c_n, c], [t_1, \dots, t_n, \text{Node}(\varnothing, [l], t)]$). insert(r, v_2)

Node(v, cs, ts). isEmpty() = $v = \varnothing \wedge cs = [] \wedge ts = []$

Node(v, cs, ts). delete(ϵ) = Node(\varnothing, cs, ts)

t_i . delete(r). isEmpty() $\rightarrow \text{Node}(v, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n])$

= Node($v, [c_1, \dots, c_{i-1}, c_{i+1}, c_n], [t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n]$)

t_i . delete(r). isEmpty() $\rightarrow \text{Node}(v, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_{i-1}, t_i, \dots, t_n]). \text{delete}(c, r)$

= Node($v, [c_1, \dots, c_{i-1}, c_i, \dots, c_n], [t_1, \dots, t_{i-1}, t_i. \text{delete}(r), \dots, t_n]$)

$c \notin cs \rightarrow \text{Node}(v, cs, ts). \text{delete}(cr) = \text{Node}(v, cs, ts)$

Komplexität

String s $len(s) = k$

$O(k)$

Tries unterstützen

String-Vervollständigungen.

Binary Tries

Idee $\Sigma = \{0, 1\}$

$\text{BT} = \text{Menge der Binary Tries}$

1. $\text{Nil} \in \text{BT}$

2. $\text{Bin}(v, l, r) \in \text{BT}$ g.d.w.:

(a) $v \in \text{Value} \cup \{\Omega\}$

(b) $l \in \text{BT}$

(c) $r \in \text{BT}$

$\text{find} : \text{BT} \times \mathbb{N} \rightarrow \text{Value} \cup \{\Omega\}$

$\text{insert} : \text{BT} \times \mathbb{N} \times \text{Value} \rightarrow \text{BT}$

$\text{delete} : \text{BT} \times \text{Key} \rightarrow \text{BT}$

a) $\text{Nil}. \text{insert}(0, v) = \text{Bin}(v, \text{Nil}, \text{Nil})$

b) $n \neq 0 \rightarrow \text{Nil}. \text{insert}(2n, v) = \text{Bin}(\Omega, \text{Nil}. \text{insert}(n, v), \text{Nil})$

c) $\text{Nil}. \text{insert}(2n+1, v) = \text{Bin}(\Omega, \text{Nil}, \text{Nil}. \text{insert}(n, v))$

d) $\text{Bin}(v_1, l_1, r_1). \text{insert}(0, v) = \text{Bin}(v_1, l_1, r)$

e) $n \neq 0 \rightarrow \text{Bin}(v, l, r). \text{insert}(2n, v) = \text{Bin}(v, l. \text{insert}(n, v), r)$

f) $\text{Bin}(v, l, r). \text{insert}(2n+1, v) = \text{Bin}(v, l, r. \text{insert}(n, v))$

g) $\text{Nil}. \text{simplify}() = \text{Nil}$

h) $\text{Bin}(\Omega, \text{Nil}, \text{Nil}). \text{simplify}() = \text{Nil}$

i) $l. \text{simplify}() = \text{Nil} \wedge r. \text{simplify}() = \text{Nil}$

$\rightarrow \text{Bin}(\Omega, l, r) = \text{Nil}$

j) $l. \text{simplify}() \neq \text{Nil} \vee r. \text{simplify}() \neq \text{Nil}$

$\rightarrow \text{Bin}(v, l, r). \text{simplify}()$
 $= \text{Bin}(v, l. \text{simplify}(), r. \text{simplify}())$

k) $\text{Nil}. \text{delete}(n) = \text{Nil}$

l) $\text{Bin}(v, l, r). \text{delete}(0) = \text{Bin}(\Omega, l, r)$

m) $n \neq 0 \rightarrow \text{Bin}(v, l, r). \text{delete}(2n) = \text{Bin}(v, l. \text{delete}(n), r). \text{simplify}()$

n) $n \neq 0 \rightarrow \text{Bin}(v, l, r). \text{delete}(2n+1) = \text{Bin}(v, l, r. \text{delete}(n)). \text{simplify}()$

Kapitel 11.6: Priority Queues

Priority Queue: (ADT) $\mathcal{D} = \langle N, P, F_S, T_S, A_x \rangle$

1. $N = \text{prioQueue}$

kleinere Zahl = höhere Priorität

2. $P = \{ \text{Priority}, \text{Value} \}$

3. $F_S = \{ \text{prioQueue}, \text{insert}, \text{remove}, \text{top}, \text{isEmpty} \}$

4. T_S :

- a) $\text{prioQueue} : \text{PrioQueue}$
- b) $\text{insert} : \text{Priority} \times \text{Priority} \times \text{Value} \rightarrow \text{PriorityQueue}$
- b) $\text{remove} : \text{PrioQueue} \rightarrow \text{PrioQueue}$
- c) $\text{top} : \text{PrioQueue} \rightarrow (\text{PrioQueue} \times \text{Value}) \cup \{\varnothing\}$
- d) $\text{isEmpty} : \text{PrioQueue} \rightarrow \mathbb{B}$

5. A_x :

- a) $\text{prioQueue}(), \text{toList}() = []$
- b) $Q.\text{insert}(p, v). \text{toList}() = \text{insertList}(p, v, Q.\text{toList}())$
- c) $Q.\text{isEmpty}() = (Q.\text{toList}() = [])$
- d) $Q.\text{toList}() = [] \rightarrow Q.\text{top}() = \varnothing$
- e) $Q.\text{toList}() \neq [] \rightarrow Q.\text{top}() = Q.\text{toList}()[0]$
- f) $Q.\text{toList}() = [] \rightarrow Q.\text{remove}() = Q$
- g) $Q.\text{toList}() \neq [] \rightarrow Q.\text{remove}(). \text{toList}() = Q.\text{toList}[1:]$

Hilfsfunktionen:

toList(): PrioQueue \rightarrow List<Priority, Value>

$Q.\text{isEmpty}() \rightarrow Q.\text{toList}() = []$

$\neg [Q.\text{isEmpty}()] \rightarrow Q.\text{toList}() = [Q.\text{top}()] + Q.\text{remove}(). \text{toList}()$

insertList: Priority \times Value \times List<Priority, Value> \rightarrow List<Priority, Value>

$\text{insertList}(p, v, []) = [p, v]$

$p_1 \leq p_2 \rightarrow \text{insertList}(p_1, v_1, [p_2, v_2] + R) = [p_1, v_1], [p_2, v_2] + R$

$p_1 > p_2 \rightarrow \text{insertList}(p_1, v_1, [p_2, v_2] + R) = [p_2, v_2] + \text{insertList}(p_1, v_1, R)$

Wichtig!: Prioritäten werden durch Zahlen dargestellt

Der Wert der Priorität verhält sich ungefähr
anti-proportional zu seinem Zahlenwert!

Heap

$H :=$ Menge der Heaps

1. $\text{Nil} \in H$

2. $\text{Node}(p, v, l, r) \in H$ gelte

- a) $p \leq l \wedge p \leq r$ (Heapbedingung p hat höhere prio als l und r)
- b) $|l.\text{count}() - r.\text{count}()| \leq 1$ (Balancierungsbedingung)
- c) $l \in H \wedge r \in H$

3. $\text{Coent}: H \rightarrow \mathbb{N}$

a) $\text{Nil}.\text{Coent}() = 0$

b) $\text{Node}(p, v, l, r).\text{Coent}() = 1 + l.\text{Coent}() + r.\text{Coent}()$

c) $\text{Nil}.\text{top}() = \varnothing$

d) $\text{Node}(p, v, l, r).\text{top}() = \langle p, v \rangle$

e) $\text{Nil}.\text{isEmpty}() = \text{true}$

f) $\text{Node}(p, v, l, r).\text{isEmpty}() = \text{false}$

g) $\text{Nil}.\text{insert}(p, v) = \text{Node}(p, v, \text{Nil}, \text{Nil})$

h) $p_{\text{top}} \leq p \wedge l.\text{count}() \leq r.\text{count}()$

$\rightarrow \text{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(p, v) = \text{Node}(p_{\text{top}}, v_{\text{top}}, l.\text{insert}(p, v), r)$

i) $p_{\text{top}} \leq p \wedge l.\text{count}() > r.\text{count}()$

$\rightarrow \text{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(p, v) = \text{Node}(p_{\text{top}}, v_{\text{top}}, l, r.\text{insert}(p, v))$

j) $p_{\text{top}} > p \wedge l.\text{count}() \leq r.\text{count}()$

$\rightarrow \text{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(p, v) = \text{Node}(p, v, l.\text{insert}(p_{\text{top}}, v_{\text{top}}), r)$

k) $p_{\text{top}} > p \wedge l.\text{count} > r.\text{count}()$

$\rightarrow \text{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(p, v) = \text{Node}(p, v, l, r.\text{insert}(p_{\text{top}}, v_{\text{top}}))$

l) $\text{Nil}.\text{remove}() = \text{Nil}$

m) $\text{Node}(p, v, \text{Nil}, \text{r}) = r$

n) $\text{Node}(p, v, l, \text{Nil}) = l$

o) $l = \text{Node}(p_1, v_1, l_1, r_1) \wedge r = \text{Node}(p_2, v_2, l_2, r_2) \wedge p_1 \leq p_2$

$\rightarrow \text{Node}(p, v, l, r).\text{remove}() = \text{Node}(p_1, v_1, l.\text{remove}(), r)$

p) $l = \text{Node}(p_1, v_1, l_1, r_1) \wedge r = \text{Node}(p_2, v_2, l_2, r_2) \wedge p_1 > p_2$

$\rightarrow \text{Node}(p, v, l, r).\text{remove}() = \text{Node}(p_2, v_2, l, r.\text{remove}())$

Heap Sort

Hat ebenso wie Mergesort die Komplexität von $O(n \cdot \log_2(n))$, benötigt aber nicht so viel Speicherplatz.

Heapsort nach Williams

A ist ein Array von Keys

1. Die Elemente von A werden in einen Heap H eingefügt.

2. Man legt das kleinste Element oben an der Wurzel an.
Wenn man nun einzelne Elemente, eins nach dem Anderen löscht, dann erhält man die Zahlen in aufsteigender Reihenfolge.

```

1 def heap_sort(L):
2     H = Nil()
3     for p in L:
4         H = H.insert(p, None)
5     S = []
6     while isinstance(H, Node):
7         p, _ = H.top()
8         S.append(p)
9         H = H.remove()
10    return S

```

toHeap : List(\mathbb{N}) \rightarrow H

toList : H \rightarrow List(\mathbb{N})

$$a) \text{toHeap}(\emptyset) := \text{Nil}$$

$$b) \text{toHeap}([x] + R) := \text{toHeap}(R).insert(x, x)$$

$$c) \text{Nil}.toList() = []$$

$$d) h \neq \text{Nil} \wedge \langle p, _ \rangle = h.\text{top}() \\ \rightarrow h.toList() = [p] + h.\text{remove()}.toList()$$