Algorithmen und Komplexität

Semester Nr. 2 Algorithmen und Komplexität bei Karl Stroetmann

Inhaltsverzeichnis

Überblick

1. Komplexität von Algorithmen
   - Groß O-Notation
2. Rekurrenz Gleichungen
   - Master Theorem
3. Sortier-Algorithmen:
   - Sortieren durch einfügen (Insertion Sort)
   - Sortieren durch Auswahl (Selection Sort)
   - Sortieren durch Mischen (Merge Sort)
   - Quicksort
   - Radix Sort
   - Heapsort
4. Abstrakte Datentypen
5. Dictionaries (und Mengen)
   - Binäre Bäume
   - AVL – Bäume + 2-3-Bäume
   - Hash Tabellen
   - Tries (Spezialfall Strings)
6. Prioritäts Warteschlangen
7. Graphentheoretische Algorithmen

Man muss nun zwischen einem Algorithmus und einem Programm unterscheiden. Ein Algorithmus ist eine abstrakte Darstellung, wie ein gegebenes Problem gelöst werden kann. Ein Programm hingegen ist die konkrete Implementierung dessen.

Des Weiteren sollte ein Algorithmus drei Kriterien erfüllen:

1. Ein Algorithmus muss korrekt sein
2. Ein Algorithmus sollte effizient sein in Bezug auf die Rechenzeit und den Speicherverbrauch
3. Ein Algorithmus sollte einfach sein.

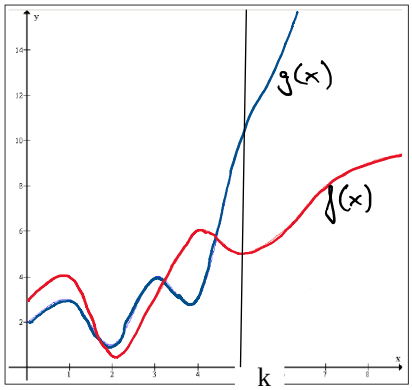
Groß-O-Notation

Motivation

Wie berechnen Rechner die Zeiten eines Algorithmus?

1. Implementierung in Programmiersprache
2. Zählen von arithmetischen Operationen und Speicherzugriffen
3. Nachschlagen der Zeit der Operationen im Prozessorhandbuch
4. Berechnung der Rechenzeit

Die Groß-O-Notation ist eine abstrakte Möglichkeit, die das Wachstum der Rechenzeit in Abhängigkeit von der Größe der Eingabe beschreiben. Die O-Notation soll von konstanten Faktoren und unwesentlichen Termen abstrahieren.
Man definiert einen X-Wert (k), ab dem die Funktion $f(x)$ (rot) immer unter $c \cdot g(x)$ liegt. Das c muss ebenfalls definiert werden und gibt einen Faktor der Funktion g(x) (blau) an, für welche dann das Wachstum von $f(x)$ ab k immer unterhalb von $c \cdot g(x)$ verläuft.



Definition der Groß-O-Notation

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

Reflexivität der Groß-O-Notation

Proposition 2 (Reflexivity of Big $\mathcal{O}$ Notation) For all functions $f : \mathbb{N} \rightarrow \mathbb{R}_+$ we have that

$f \in \mathcal{O}(f)$ holds.

Proof: Let us define $k := 0$ and $c := 1$. Then our claim follows immediately from the inequality

$\forall n \in \mathbb{N}: f(n) \leq f(n)$. □

Multiplikation von Konstanten

Assume that we have functions $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ and a number $d \in \mathbb{R}_+$. Then we have

$g \in \mathcal{O}(f) \rightarrow d \cdot g \in \mathcal{O}(f)$.

Proof: The premiss $g \in \mathcal{O}(f)$ implies that there are constants $c' \in \mathbb{R}_+$ and $k' \in \mathbb{N}$ such that

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

holds. If we multiply the inequality involving $g(n)$ with d, we get

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

Let us therefore define $k := k'$ and $c := d \cdot c'$. Then we have

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

and by definition this implies $d \cdot g \in \mathcal{O}(f)$. □

Remark: The previous proposition shows that the big $\mathcal{O}$ notation does indeed abstract from constant factors. ◇

Addition

Proposition 4 (Addition) Assume that $f, g, h \colon \mathbb{N} \to \mathbb{R}_+$. Then we have

$$f \in \mathcal{O}(h) \land g \in \mathcal{O}(h) \to f + g \in \mathcal{O}(h).$$

Proof: The preconditions $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$ imply that there are constants $k_1, k_2 \in \mathbb{N}$ and $c_1, c_2 \in \mathbb{R}$ such that both

$$\forall n \in \mathbb{N} \colon \big(n \geq k_1 \to f(n) \leq c_1 \cdot h(n)\big) \quad \text{and}$$

$$\forall n \in \mathbb{N} \colon \big(n \geq k_2 \to g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 + c_2$. For all $n \in \mathbb{N}$ such that $n \geq k$ it then follows that both

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

holds. Adding these inequalities we conclude that

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

holds for all $n \geq k$. □

Exercise 2: Assume that $f_1, f_2, h_1, h_2 \colon \mathbb{N} \to \mathbb{R}_+$. Prove that

$$f_1 \in \mathcal{O}(h_1) \land f_2 \in \mathcal{O}(h_2) \to f_1 \cdot f_2 \in \mathcal{O}(h_1 \cdot h_2) \quad \text{holds.}$$ ◇

Exercise 3: Assume that $f_1, f_2, h_1, h_2 \colon \mathbb{N} \to \mathbb{R}_+$. Prove or refute the claim that

$$f_1 \in \mathcal{O}(h_1) \land f_2 \in \mathcal{O}(h_2) \to f_1 / f_2 \in \mathcal{O}(h_1 / h_2) \quad \text{holds.}$$ ◇

Transitivität der Groß-O-Notation

Proposition 5 (Transitivity of Big $\mathcal{O}$ Notation) Assume $f, g, h \colon \mathbb{N} \to \mathbb{R}_+$. Then we have

$$f \in \mathcal{O}(g) \land g \in \mathcal{O}(h) \to f \in \mathcal{O}(h).$$

Proof: The precondition $f \in \mathcal{O}(g)$ implies that there exists a $k_1 \in \mathbb{N}$ and a number $c_1 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N} \colon \big(n \geq k_1 \to f(n) \leq c_1 \cdot g(n)\big)$$

holds, while the precondition $g \in \mathcal{O}(h)$ implies the existence of $k_2 \in \mathbb{N}$ and $c_2 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N} \colon \big(n \geq k_2 \to g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 \cdot c_2$. Then for all $n \in \mathbb{N}$ such that $n \geq k$ we have the following:

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

Let us multiply the second of these inequalities with c_1. Keeping the first inequality this yields

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

The transitivity of the relation $\leq$ immediately implies $f(n) \leq c \cdot h(n)$ for $n \geq k$. □

Grenzwert der Groß-O-Notation

Proposition 6 (Limit Proposition) Assume that $f, g: \mathbb{N} \to \mathbb{R}_+$. Furthermore, assume that the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists. Then we have $f \in \mathcal{O}(g)$.

Proof: Define

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

Since the limit exists by our assumption, we know that

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

Since this is valid for all positive values of ε, let us define $\varepsilon := 1$. Then there exists a number $k \in \mathbb{N}$ such that for all $n \in \mathbb{N}$ satisfying $n \geq k$ the inequality

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

holds. Let us multiply this inequality with $g(n)$. As $g(n)$ is positive, this yields

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

The triangle inequality $|a+b| \leq |a| + |b|$ for real numbers tells us that

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

holds. Combining the previous two inequalities yields

$$f(n) \leq \left|f(n) - \lambda \cdot g(n)\right| + \lambda \cdot g(n) \leq g(n) + \lambda \cdot g(n) = (1+\lambda) \cdot g(n).$$

Therefore, we define

$$c := 1 + \lambda$$

and have shown that $f(n) \leq c \cdot g(n)$ holds for all $n \geq k$. □

Lösen einer Rekurrenzgleichung

1. Homogenen Teil lösen ($x_n = \lambda^n$)
    - Die Rekurrenzgleichung ohne +1 am Ende
2. Inhomogenen Teil lösen ($x_n = \gamma^n$)
    - Die gesamte Rekurrenzgleichung
3. Einsetzen in die allgemeine Lösung
    - $x_n = \alpha \cdot \lambda_1^n + \beta \cdot \lambda_2^n + \gamma$
    - 0000

Arten der Groß-O-Notation

1. Die O-Notation:
    - $\mathcal{O}(g) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq k \to f(n) \leq c \cdot g(n))\}$
2. Die Ω-Notation:
    - $\Omega(g) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq k \to f(n) \geq c \cdot g(n))\}$
3. Die Θ-Notation:
    - $\Theta(g) := \mathcal{O}(g) \cap \Omega(g)$

Master-Theorem

Mit dem Master-Theorem kann man einfacher die Komplexität einer rekursiven Funktion bestimmen.

Bedingungen für das Master-Theorem

a. $\alpha, \beta \in \mathbb{N}$, so dass $\beta \geq 2, \delta \in \mathbb{R}$, so dass $\delta \geq 0$ und
b. die Funktion $f : \mathbb{N} \to \mathbb{R}_+$ erfüllt die Rekurrenzrelation

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

Anwendung des Master-Theorems

Man kann durch das Master-Theorem alle benötigten Variablen schnell ablesen und so die Komplexität des Algorithmus in einen der folgenden Fälle einordnen.

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

Sortieren

Das Sortierproblem

Es wird immer davon ausgegangen, dass eine Liste L gegeben ist. Alle Elemente aus L stammen aus der Menge S. S verfügt über eine binäre Relation ≤, welche reflexiv, anti-symmetrisch und transitiv ist.

1. $\forall x \in S : x \leq x$ (≤ ist reflexiv)
2. $\forall x, y \in S : (x \leq y \wedge y \leq x \rightarrow x = y)$ (≤ ist anti-symmetrisch)
3. $\forall x, y, z \in S : (x \leq y \wedge y \leq z \rightarrow x \leq y)$ (≤ ist transitiv)

Definition Lineare Ordnung

Ein Paar $\langle S, \leq \rangle$ in dem S ein Set ist und $\leq \subseteq S \times S$ eine Relation auf S, welche reflexiv, anti-symmetrisch und transitiv ist, wird partiell geordnet genannt. Wenn zudem noch

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

gilt, dann ist das Paar $\langle S, \leq \rangle$ in der totalen Ordnung und die Relation ≤ wird als lineare Ordnung bezeichnet.

Definition Quasiordnung

Ein Paar $\langle S, \leq \rangle$ ist in Quasiordnung, genau dann, wenn:

1. $\leq \subseteq S \times S$
2. $\forall x \in S : x \leq x$ (≤ ist reflexiv)
3. $\forall x, y, z \in S : (x \leq y \wedge y \leq z \rightarrow x \leq y)$ (≤ ist transitiv)

Wenn eine Quasiordnung vorliegt und dazu noch

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

gilt, dann liegt eine Totale-Quasi-Ordnung vor.

Definition Sortier-Problem

Gegeben: $\langle M, \leq \rangle$ ist in Totaler-Quasi-Ordnung. L ist eine Liste aus M.

Gesucht: Liste aus S mit zwei Eigenschaften:

1. Liste S ist aufsteigend sortiert
   $\forall i \in \{0,...,len(S) - 2\} : S[i] \leq S[i + 1]$
2. Liste S enthält dieselben Elemente wie L und doppelte Elemente in S kommen auch doppelt in L vor.

Insertion Sort

Er durchläuft Schritt für Schritt eine Liste und entnimmt dabei aus der unsortierten Eingabefolge ein Element und setzt es dann an der entsprechend richtigen Stelle wieder ein.

Formale Definition

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

Implementierung

```
def sort(L):
    if L == []:
        return []
    x, R = L[0], L[1:]
    return insert(x, sort(R))
```

```
def insert(x, L):
    if L == []:
        return [x]
    y, R = L[0], L[1:]
    if x <= y:
        return [x] + L
    else:
        return [y] + insert(x, R)
```

Komplexität

Insertion Sort ist gerade bei teilweise sortierten Listen sehr effizient.

Bester Fall

Liste ist bereits sortiert

$$\mathcal{O}(n)$$

Worst-case

Liste ist falsch herum sortiert

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

Durchschnittlicher Fall

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

Selection Sort

Das Minimum aus der Liste L wird entnommen und dort gelöscht. Das Minimum wird vor die Restliste gesetzt und dies geschieht rekursiv.

Formale Definition

$sort([]) = []$

$L \neq [] \wedge x := min(L) \rightarrow sort(L) = [x] + sort(delete(x, L))$

Implementierung

```
def sort(L):
    if L == []:
        return []
    x = min(L)
    return [x] + sort(delete(x, L))
```

```
def delete(x, L):
    if L == []:
        assert L != [], f'delete({x}, [])'
    if L[0] == x:
        return L[1:]
    return [L[0]] + delete(x, L[1:])
```

Komplexität

Immer

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

Merge Sort

Merge Sort arbeitet nach dem Teilen-und-Herrsche-Prinzip. Es werden immer zwei geleichgroße Teillisten gebildet und sortiert und im Anschluss zusammengemerged.

Formale Definition

$n < 2 \rightarrow sort(L) = L$
$n \geq 2 \rightarrow sort(L) = merge(sort(L[: n//2]), sort(L[n//2 :]))$
$merge([], L_2) = L_2$
$merge([], L_1) = L_1$
$x \preceq y \rightarrow merge([x \mid R_1], [y \mid R_2]) = [x] + merge(R_1, [y \mid R_2])$
$\neg x \preceq y \rightarrow merge([x \mid R_1], [y \mid R_2]) = [y] + merge([x \mid R_1], R_2)$

Implementierung

```
def sort(L):
    n = len(L)
    if n < 2:
        return L
    L1, L2 = L[:n//2], L[n//2:]
    return merge(sort(L1), sort(L2))
```

```
def merge(L1, L2):
    if L1 == []:
        return L2
    if L2 == []:
        return L1
    x1, R1 = L1[0], L1[1:]
    x2, R2 = L2[0], L2[1:]
    if x1 <= x2:
        return [x1] + merge(R1, L2)
    else:
        return [x2] + merge(L1, R2)
```

Komplexität

Immer

$$f(n) \in \mathcal{O}(n \cdot log_2(n))$$

Komplexität in Form des Master-Theorems

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

Somit sind $\alpha = 2, \beta = 2, \delta = 1$ gegeben und der zweite Fall des Master-Theorems gilt:

$$f(n) \in \mathcal{O}(n \cdot log_2(n))$$

Quick Sort

Man wählt ein Pivot-Element und vergleicht die anderen Elementen mit diesem. Ist das Pivot-Element größer, wird das Element in die Linke Teilliste gegeben und für den Fall, dass es kleiner ist in die rechte Teilliste. Dieser Schritt wird rekursiv ausgeführt.

Formale Definition

$$sort([]) = []$$
$$sort([x] + R) = sort([y \in R \mid y < x]) + [x] + sort([y \in R \mid y \geq x])$$

Implementierung

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

Komplexität

Worst-Case

$$\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n \in \mathcal{O}(n^2)$$

Durchschnittlicher Fall

$$2 \cdot ln(2) \cdot n \cdot log_2(n) + \mathcal{O}(n)$$

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

Counting Sort

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

Formale Definition

Counting Sort vergleicht die Elemente nicht, daher ist seine Komplexität linear mit $\mathcal{O}(n)$. Counting Sort durchläuft drei Schritte, um eine Sortierung zu erreichen.

Counting Stage

Die Häufigkeit der vorkommenden Elemente wird bestimmt.

Indexing-Stage

Es werden Sublisten angelegt mit der Länge der Häufigkeiten der einzelnen Elemente. Bspw. kommt ein Element x:10 (also x kommt 10 Mal vor) dann hat die Subliste die Länge 10. Diese Sublisten werden dann verkettet und bilden eine große Liste aus Listen.

Distribution-Stage

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

Implementierung

```
def countingSort(Names, Grades):
    assert len(Names) == len(Grades)
    maxGrade     = 256
    Counts       =    [0] * maxGrade
    Indices      = [None] * maxGrade
    SortedNames  = [None] * len(Names)
    SortedGrades = [None] * len(Names)
    # Phase 1: Counting
    for g in Grades:
        Counts[g] += 1
    # Phase 2: Indexing
    Indices[0] = 0
    for g in range(1, maxGrade):
        Indices[g] = Indices[g-1] + Counts[g-1]
    # Phase 3: Distribution
    for i in range(len(Names)):
        grade              = Grades [i]
        idx                = Indices[grade]
        SortedNames [idx] = Names  [i]
        SortedGrades[idx] = Grades [i]
        Indices[grade]   += 1
    return SortedNames, SortedGrades
```

Komplexität

Wie bereits beschrieben ist die Komplexität linear

$\mathcal{O}(n)$

Radix Sort

Ziel ist es eine Liste mit 32-Bit Zahlen zu sortieren.

Vorgang

Eine 32-Bit Zahl kann wie folgt aufgeteilt werden.

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

Radix Sort verwendet Counting Sort und besteht aus vier Schritten.

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

Da Counting Sort stabil ist kann man so vorgehen.

Implementierung

```
def extractByte(n, k):
    return n >> (8 * (k-1)) & 255
```

```
def radixSort(L):
    for k in range(1, 4+1):
        Grades = [extractByte(n, k) for n in L]
        L      = countingSort(L, Grades)[0]
    return L
```

Abstrakte Datentypen (ADT)

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

Die formale Definition eines ADT

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

$$D = \langle N, P, Fs, Ts, A\,x \rangle$$

1. N ist ein String. N ist der Name des ADT's.
2. P ist Menge der Typ-Parameter
3. Fs ist Menge der Funktionszeichen
4. Ts ist Menge der Typspezifikationen

   Für jedes $f \in Fs$ gibt es in Ts genau eine Typspezifikation.

   $f : T_1 \times .. \times T_n \rightarrow S$
5. Ax ist Menge der mathematischen Formeln. Diese spezifizieren den ADT

Vorteile von ADT's

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

Der ADT Stack

1. N = „Stack“
2. P = {Element}
3. Fs = {Stack, push, pop, top, isEmpty}
4. Ts:
    a. Stack : Stack
    b. push : Stack x Element → Stack
    c. pop : Stack → Stack
    d. top : Stack → Element
    e. isEmpty : Stack → $\mathbb{B}$
5. Ax:
    a. Stack().top() = Ω
    b. S.push(x).top() = x
    c. Stack().pop() = Ω
    d. S.push(x).pop() = S
    e. Stack().isEmpty() = true
    f. S.push(x).isEpmty = false

Definition Generatoren

Generatoren sind Funktionszeichen, mit denen man jeden möglichen zustand Erreichen kann (Bei Stack: Stack und push)

Implementierung

```
class Stack:
    def __init__(self):
        self.mStackElements = []

    def push(self, e):
        self.mStackElements.append(e)

    def pop(self):
        assert len(self.mStackElements) > 0, "popping empty stack"
        self.mStackElements = self.mStackElements[:-1]

    def top(self):
        assert len(self.mStackElements) > 0, "top of empty stack"
        return self.mStackElements[-1]

    def isEmpty(self):
        return self.mStackElements == []

    def copy(self):
        C = Stack()
        C.mStackElements = self.mStackElements[:]
        return C

    def __str__(self):
        C = self.copy()
        result = C._convert()
        dashes = "-" * len(result)
        return '\n'.join([dashes, result, dashes])

    def _convert(self):
        if self.isEmpty():
            return '|'
        t = self.top()
        self.pop()
        return self._convert() + ' ' + str(t) + ' |'

def createStack(L):
    S = Stack()
    n = len(L)
    for i in range(n):
        S.push(L[i])
        print(S)
    return S

createStack(range(10))
```

The Shunting-Yard-Algorithm

Der Algorithmus soll einen Arithmetischen Ausdruck auswerten.

Der Shunting-Yard verwaltet die Eingabesymbole:

- Ts: Token Stack, enthält die Eingabesymbole „1+2*3“ TS = [3, ‘*‘, 2, ‘+‘, 1]
- AS: Argument Stack, enthält Zahlen, Anfangs leer. Zahlen aus dem TS auf den AS
- OS: Operator Stack, enthält Operatoren, als auch ‘(‘

Präzidenz	Operator
1	+, -
2	*, /
3	**

Vorgang

1. Lesevorgang:
   Die Tokens aus dem TS werden entweder auf den AS oder OS gelegt. Es können je nach Präzedenz jedoch auch hier schon Evaluierungen ausgeführt werden.

2. Evaluierungsphase:
   Die Operatoren werden evaluiert
   a) Element vom OP entnehmen
   b) Element vom AS entnehmen
   c) Das Ergebnis bestimmen
   d) Das Ergebnis auf den AS legen

```
def evaluate(self):
    while not self.mTokens.isEmpty():
        nextOp = self.mTokens.top(); self.mTokens.pop()
        if isinstance(nextOp, int):
            self.mArguments.push(nextOp)
            continue
        if self.mOperators.isEmpty():
            self.mOperators.push(nextOp)
            continue
        if nextOp == "(":
            self.mOperators.push(nextOp)
            continue
        stackOp = self.mOperators.top()
        if stackOp == "(" and nextOp == ")":
            self.mOperators.pop()
            continue
        if nextOp == ")":
            self.popAndEvaluate()
            self.mTokens.push(nextOp)
            continue
        if stackOp == '(':
            self.mOperators.push(nextOp)
            continue
        if evalBefore(stackOp, nextOp):
            self.popAndEvaluate()
            self.mTokens.push(nextOp)
        else:
            self.mOperators.push(nextOp)
    while not self.mOperators.isEmpty():
        self.popAndEvaluate()
    return self.mArguments.top()
```

```
def popAndEvaluate(self):
    rhs = self.mArguments.top(); self.mArguments.pop()
    lhs = self.mArguments.top(); self.mArguments.pop()
    op  = self.mOperators.top(); self.mOperators.pop()
    result = None
    if op == '+':
        result = lhs + rhs
    if op == '-':
        result = lhs - rhs
    if op == '*':
        result = lhs * rhs
    if op == '/':
        result = lhs // rhs
    if op == '%':
        result = lhs % rhs
    if op == '**':
        result = lhs ** rhs
    self.mArguments.push(result)
```

Set and Maps

Eine Funktionale Relation sind Maps. In Python sind sie als Dictionaries dargestellt.

Der ADT Map

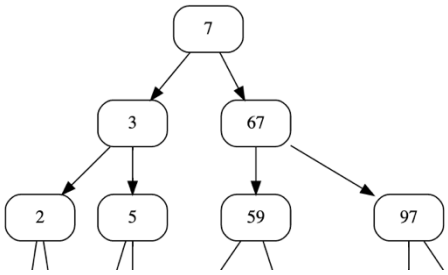
$D = \langle N, P, Fs, Ts, A\,x \rangle$

1. N = Map
2. P = {Key, Value}
3. Fs = {map, find, insert, delete}
4. Ts:
   a. map : Map
   b. Map x key → Value ∪ {Ω}
   c. insert : Map x key x value → Map
   d. delete : Map x key → Map
5. Ax:
   a. $map().find(k) = \Omega$
   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. $k_1 \neq k_2 \rightarrow m.delete(k_1).find(k2) = m.find(k_2)$

```python
class Map:
    def __init__(self):
        self.mDictionary = {}

    def find(self, k):
        return self.mDictionary[k]

    def insert(self, k, v):
        self.mDictionary[k] = v

    def delete(self, k):
        del self.mDictionary[k]

    def __str__(self):
        return str(self.mDictionary)
```

Geordnete binäre Bäume

Ein binärer Baum heißt geordnet, wenn die Daten eine Ordnungsrelation erfüllen, also vergleichbar sind und für jeden Knoten gilt, dass der linke Nachfolger kleiner (oder kleiner gleich) dem rechten Nachfolger ist.



1. Nil ist ein geordneter Binärbaum
2. $Node(k,v,l,r) \in B$ g.d.w.
   a. k ist ein Schlüssel der Menge Key
   b. v ist ein Value der Menge Value
   c. l ist ein geordneter Binärbaum
      l ist der linke Unterbaum von $Node(k,v,l,r)$
   d. r ist ein geordneter Binärbaum
      r ist der rechte Unterbaum von $Node(k,v,l,r)$
   e. alle Keys in l sind kleiner k
   f. alle Keys in r sind größer k

   } Ordnungsbedingung
3.
   a. $map() = Nil$
   b. $find : B \times key \to Value \cup \{\Omega\}$
   c. $insert : B \times key \times value \to B$
   d. $delete : B \times key \to B$
   e. $delMin : B \to B \times key \times value$
4. FIND
   a. $Nil\,.find(k) = \Omega$
   b. $Node(k,v,l,r)\,.find(k) = v$
   c. $k_1 < k_2 \to Node(k_2,v,l,r)\,.find(k_1) = l\,.find(k_1)$
   d. $k_1 > k_2 \to Node(k_2,v,l,r)\,.find(k_1) = r\,.find(k_1)$

   INSER
   e. $Nil\,.insert(k,v) = Node(k,v,l,r)$
   f. $Node(k_2,v_2,l,r)\,.insert(k,v_1) = Node(k,v_1,l,r)$
   g. $k_1 < k_2 \to Node(k_2,v_2,l,r)\,.insert(k_1,v_2) = Node(k_2,v_2,l\,.insert(k_1,v_1),r)$
   h. $k_1 > k_2 \to Node(k_2,v_2,l,r)\,.insert(k_1,v_2) = Node(k_2,v_2,l,r\,.insert(k_1,v_1))$

   DELMIN
   i. $Node(k,v,l,r)\,.delMin() = \langle r,k,v\rangle$
   j. $l \neq Nil \wedge l\,.delMin() = \langle l',k_{min},v_{min}\rangle$
      $\to Node(k,v,l,r)\,.delMin() = \langle Node(k,v,c',r),k_{min},v_{min}\rangle$

   DELETE
   k. $Nil\,.delete(k) = Nil$
   l. $Node(k,v,Nil,r)\,.remove(k) = r$
   m. $Node(k,v,l,Nil)\,.remove(k) = l$
   n. $l \neq Nil \wedge r \neq Nil \wedge delMin() = \langle r',k_{min},v_{min}\rangle$
      $\to Node((k,v,l,r)\,.delete(k) = Node(k_{min},v_{min},l,r')$
   o. $k_1 < k_2 \to Node(k_2,v_2,l,r)\,.delete(k_1) = Node(k_2,v_2,l\,.delete(k_1),r)$
   p. $k_1 > k_2 \to Node(k_2,v_2,l,r)\,.delete(k_1) = Node(k_2,v_2,l,r\,.delete(k_1))$

Komplexität

Average Case:

Logarithmisch $\mathcal{O}(ln(n))$

Worst Case:

Linear $\mathcal{O}(n)$

AVL Bäume

Der AVL-Baum ist eine Datenstruktur in der Informatik, und zwar ein binärer Suchbaum mit der zusätzlichen Eigenschaft, dass sich an jedem Knoten die Höhe der beiden Teilbäume um höchstens eins unterscheidet.

Höhenbedingung

1. $Nil.height() = 0$
2. $Node(k,v,l,r).height() = max(l.height(), r.height()) + 1$

Menge A der AVL Bäume

1. $Nil \in A$
2. $Node(k,v,l,r) \in A$ genau dann, wenn:
   a. $Node(k,v,l,r) \in B$
   b. $l, r \in A$
   c. $|l.height() - r.height()| \leq 1$
3.
   a. $restore : B \to A$
   b. $insert : B \times key \times value \to B$
   c. $find : B \times key \to value \cup \{\Omega\}$
   d. $delete : V \times key \to B$
   e. $delMin : B \to B \times key \times value$
4. FIND
   a. $Nil.find(k) = \Omega$
   b. $Node(k,v,l,r).find(k) = v$
   c. $k_1 < k_2 \to Node(k_2,v,l,r).find(k_1) = l.find(k_1)$
   d. $k_1 > k_2 \to Node(k_2,v,l,r).find(k_1) = r.find(k_1)$

   INSERT
   e. $Nil.insert(k,v) = Node(k,v,Nil,Nil)$
   f. $Node(k,v_2,l,r).insert(k,v_1) = Node(k,v_1,l,r)$
   g. $k_1 < k_2 \to Node(k_2,v_2,l,r).insert(k_1,v_1) = Node(k_1,v_2,l.insert(k_1,v_2),r).restore()$
   h. $k_1 > k_2 \to Node(k_2,v_2,l,r).insert(k_1,v_1) = Node(k_1,v_2,l,r.insert(k_1,v_2)).restore()$

   DELMIN
   i. $Node(k,v,Nil,r).delMin() = \langle r,k,v\rangle$
   j. $l \neq Nil \wedge \langle l', k_{min}, v_{min}\rangle := l.delMin()$
   $\to Node(k,v,l,r).delMin() = \langle Node(k,v,l',r).restore(), k_{min}, v_{min}\rangle$

   DELETE
   k. $Nil.delete(k) = Nil$
   l. $Node(k,v,Nil,r).delete(k) = r$
   m. $Node(k,v,l,Nil).delete(k) = l$
   n. $l \neq Nil \wedge r \neq Nil \wedge \langle r', k_{min}, v_{min}\rangle := r.delMin()$
   $\to Node(k,v,l,r).delete(k) = Node(k_{min}, v_{min}, l, r').restore()$
   o. $k_1 < k_2 \to Node(k_2,v_2,l,r).delete(k_1) = Node(k_2,v_2,l.delete(k_1),r).restore()$
   p. $k_1 > k_2 \to Node(k_2,v_2,l,r).delete(k_1) = Node(k_2,v_2,l,r.delete(k_1)).restore()$

Verbesserungsansätze

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

Komplexität

Die komplexität von AVL Bäumen beträgt immer $\mathcal{O}(ln(n))$

Tries

Ein Trie oder Präfixbaum ist eine Datenstruktur, die in der Informatik zum Suchen nach Zeichenketten verwendet wird. Es handelt sich dabei um einen speziellen Suchbaum zur gleichzeitigen Speicherung mehrerer Zeichenketten.

Bei einem Trie können kann jeder Knoten so viele Kinder haben, wie Buchstaben im Alphabet vorhanden sind. Tries unterstützen die String-Vervollständigung.

Notation

$\sum$: Endliche Mege der Buchstaben (Alphabet)
$\sum^*$: Die Menge der Wörter die in mit $\sum$ gebildet werden können.
ϵ : leeres Wort ϵ = „“
Value : Menge der Werte, die den Schlüssel zugeordnet werden kann.

$\mathbb{T}$: Menge der Tries

Node : $Value \times Liste(\sum) \times Liste(\mathbb{T}) \to \mathbb{T}$

$Node(k, v, l, r) \in \mathbb{T}$, wenn gilt:

a) $v \in Value \cup \{\Omega\}$
b) $Cs = [c_1, \ldots, c_n] \in List(\sum)$ (Eine Liste verschiedener Charaktere der Länge n)
c) $Ts = [t_1, \ldots, t_n] \in List(\mathbb{T})$ (Eine Liste von Tries derselben Länge)

find : $\mathbb{T} \times \sum^* \to Value \cup \{\Omega\}$
insert : $\mathbb{T} \times \sum^* \times Value \to \mathbb{T}$
isEmpty : $\mathbb{T} \to \mathbb{B}$
delete : $\mathbb{T} \times \sum^* \to \mathbb{T}$

FIND

$Node(v, cs, ts).find(\epsilon) = v$
$Node(v, [c_1, \ldots c_n], [t_1, \ldots, t_n]).find(cr) = t_i.find(r)$
$c \neq Cs \to Node(v, ls, Ts).find(cr) = \Omega$

INSERT

$Node(v_1, l, T).insert(\epsilon, v_2) = Node(v_2, l, T)$
$Node(v_1, [c_1, \ldots, c_i, \ldots, c_n], [t_1, \ldots, t_i, \ldots, t_n])$
$= Node(v_1, [c_1, \ldots, c_i, \ldots, c_n], [t_1, \ldots, t_i, \ldots, t_n])$
$c \neq \{c_1, c_n\} \to Node(v_1, [c_1, \ldots, c_n], [t_1, \ldots, t_n]).insert(cr, v_2)$
$= Node(v_1, [c_1, \ldots c_n, c], [t_1, \ldots, t_n, Node(\Omega, [], []).insert(r, v_2)])$

ISEMPTY

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

DELETE

$Node(v, cs, Ts).delete(\epsilon) = Node(\Omega, Cs, Ts)$
$t_i.delete(r).isEmpty() \to Node(v, [c_1, \ldots, c_i, \ldots, C_n], [t_1, \ldots, t_i, \ldots, t_n])$
$= Node(v, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, C_n], [t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n])$
$\neg t_i.delete(r).isEmpty() \to Node(v, [c_1, \ldots, c_i, \ldots, C_n], [t_1, \ldots, t_i, \ldots, t_n]).delete(c_i r)$
$= Node(v, [c_1, \ldots, c_i, \ldots, C_n], [t_1, \ldots, t_i.delete(r), \ldots, t_n])$
$c \neq cs \to Node(v, Cs, Ts).delete(cr) = Node(v, Cs, Ts)$

Binary Tries

Der Binary Trie fungiert wie der Normale Trie, nur dass sein Alphabet aus der Menge {0,1} besteht.

Idee $\sum \{0,1\}$
BT = Menge der Binary Tries

1. $Nil \in BT$
2. $Bin(v,l,r) \in BT$, wenn gilt:
   a. $v \in Value \cup \{\Omega\}$
   b. $l \in BT$
   c. $r \in BT$
3. a. $find : BT \times \mathbb{N} \to Value \cup \{\Omega\}$
   b. $insert : BT \times \mathbb{N} \times value \to BT$
   c. $delete : BT \times key \to BT$
4. INSERT
   a. $Nil.insert(0,v) = Bin(v, Nil, Nil)$
   b. $n \neq 0 \to Nil.insert(2n,v) = Bin(\Omega, Nil.insert(n,v), Nil)$
   c. $Nil.insert(2n+1,v) = Bin(\Omega, Nil, Nil.insert(n,v))$
   d. $Bin(v_1,l,r).insert(0,v_2) = Bin(v_2,l,r)$
   e. $n \neq 0 \to Bin(v_1,l,r).insert(2n,v_2) = Bin(v_1,l.insert(n,v_2),r)$
   f. $Bin(v_1,l,r).insert(2n+1,v_2) = Bin(v_1,l,r.insert(n,v_2))$

   FIND
   g. $Nil.find(n) = \Omega$
   h. $Bin(v,l,r).find(0) = v$
   i. $\neq 0 \to Bin(v,l,r).find(2 \cdot n) = l.find(n)$
   j. $Bin(v,l,r).find(2 \cdot n + 1) = r.find(n)$

   SIMPLIFY
   k. $Bin(\Omega, Nil, Nil).simplify() = Nil$
   l. $v \neq \Omega \vee l \neq Nil \vee r \neq Nil \to Bin(v,l,r).simplify() = Bin(v,l,r)$

   DELETE
   m. $Nil.delete(n) = Nil$
   n. $Bin(v,l,r).delete(0) = Bin(\Omega,l,r).simplify()$
   o. $n \neq 0 \to Bin(v,l,r).delete(2 \cdot n) = Bin\big(v,l.delete(n),r\big).simplify()$
   p. $Bin(v,l,r).delete(2 \cdot n + 1) = Bin\big(v,l,r.delete(n)\big).simplify()$

Priority Queues

Der ADT Priority Queue

In der Informatik ist eine Vorrangwarteschlange eine spezielle abstrakte Datenstruktur, genauer eine erweiterte Form einer Warteschlange. Den Elementen, die in die Warteschlange gelegt werden, wird ein Schlüssel mitgegeben, der die Reihenfolge der Abarbeitung der Elemente bestimmt.

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

1. $N = ProQueue$
2. $P = \{priority, value\}$
3. $Fs = \{proiQueue, insert, remove, top, isEmpty\}$
4.
   a. $proiQueue : ProiQueue$
   b. $insert : ProQueue \times Proirity \times Value \rightarrow ProirityQueue$
   c. $remove : ProiQueue \rightarrow ProiQueue$
   d. $top : ProiQueue \rightarrow (ProiQueue \times value) \cup \{\Omega\}$
   e. $isEmplty : ProiQueue \rightarrow \mathbb{B}$
5.
   a. $prioQueue(), toList() = []$
   b. $Q.insert(p, r).toList() = insertList(p, v, Q.toList())$
   c. $Q.isEmpty() = (Q.toList() = [])$
   d. $Q.toList() = [] \rightarrow Q.top() = \Omega$
   e. $Q.toList() \neq [] \rightarrow Q.top() = Q.toList()[0]$
   f. $Q.toList() = [] \rightarrow Q.remove() = Q$
   g. $Q.toList() \neq [] \rightarrow Q.remove().toList() = Q.toList[1:]$

Hilfsfunktonen

6. TOLIST
   a. $ProiQueue \rightarrow List < Proirity, Value >$
   b. $Q.isEmpty() \rightarrow Q.toList() = []$
   c. $\neg[Q.isEmpty()] \rightarrow Q.toList() = [Q.top()] + Q.remove().toList()$

   INSERLIST
   d. $priority \times value \times List < Priority, Value > \rightarrow List < Priority, Value >$
   e. $insertList(p, v, []) = [< p, v >]$
   f. $p_1 \leq p_2 \rightarrow insertList(p_1, v_1, [p_2, v_2] + R) = [< p_1, v_1 > , < p_2, v_2 >] + R$
   g. $p_1 > p_2 \rightarrow insertList(p_1, v_1, [p_2, v_2] + R) = [< p_2, v_2 >] + insertList(p_1, v_1 + R)$

Heap

Ein Heap in der Informatik ist eine zumeist auf Bäumen basierende abstrakte Datenstruktur. In einem Heap können Objekte oder Elemente abgelegt und aus diesem wieder entnommen werden. Sie dienen damit der Speicherung von Mengen. Den Elementen ist dabei ein Schlüssel zugeordnet, der die Priorität der Elemente festlegt.

1. $Nil \in H$
2. $Node(p,v,l,r) \in H$
   a. $p \leq l \wedge p \leq r$ (Heapbedingung)
   b. $|l.count() - r.count()| \leq 1$ (Balancierbedingung)
   c. $l \in H \wedge r \in H$
3. $count : H \to \mathbb{N}$

COUNT

a. $Nil.top() = 0$
b. $Node(p,v,l,r).count() = 1 + l.count() + r.count()$

TOP

c. $Nil.top() = \Omega$
d. $Node(p,v,l,r).top() = < p,v >$

ISEMPTY

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

INSERT

g. $p_{top} \leq p \wedge l.count() \leq r.count()$
   $\to Node(p_{top},v_{top},l,r).insert(p,v) = Node(p_{top},v_{top},l.insert(p,v),r)$
h. $p_{top} \leq p \wedge l.count() > r.count()$
   $\to Node(p_{top},v_{top},l,r).insert(p,v) = Node(p_{top},v_{top},l,r.insert(p,v))$
i. $p_{top} < p \wedge l.count() \leq r.count()$
   $\to Node(p_{top},v_{top},l,r).insert(p,v) = Node(p,v,l.insert(p_{top},v_{top}),r)$
j. $p_{top} > p \wedge l.count() > r.count()$
   $\to Node(p_{top},v_{top},l,r).insert(p,v) = Node(p,v,l,r.insert(p_{top},v_{top}))$

REMOVE

k. $Nil.remove() = Nil$
l. $Node(p,v,Nil,r) = r$
m. $Node(p,v,l,Nil) = l$
n. $l = Node(p_1,v_1,l_1,r_1) \wedge r = Node(p_2,v_2,l_2,r_2) \wedge p_1 \leq p_2$
   $\to Node(p,v,l,r).remove() = Node(p_1,v_1,l.remove(),r)$
o. $l = Node(p_1,v_1,l_1,r_1) \wedge r = Node(p_2,v_2,l_2,r_2) \wedge p_1 > p_2$
   $\to Node(p,v,l,r).remove() = Node(p_1,v_1,l,r.remove())$

Heap Sort

Hat ebenso wie Merge Sort eine Komplexität von $\mathcal{O}(n \cdot log_2(n))$, benötigt aber nicht so viel Speicherplatz

Heapsort nach Williams

A ist ein Array von Keys

1. Die Elemente von A werden in einem Heap H eingefügt.
2. Nun liegt das kleinste Element oben an der Wurzel an. Wenn man nun einzeln die Elemente an der Wurzel entnimmt und löscht, erhält man eine Liste von den Elementen in aufsteigender Reihe.

3.
   a. $Nil.toList() = []$
   b. $h \neq Nil \wedge \langle p, \_ \rangle = h.top()$
   $\rightarrow h.toList() = [p] + h.remove().toList()$

```python
def heap_sort(L):
    H = Nil()
    for p in L:
        H = H.insert(p, None)
    S = []
    while isinstance(H, Node):
        p, _ = H.top()
        S.append(p)
        H = H.remove()
    return S
```

```python
def swap(A, i, j):
    A[i], A[j] = A[j], A[i]

def sink(A, k, n):
    while 2 * k + 1 <= n:
        j = 2 * k + 1
        if j + 1 <= n and A[j] > A[j + 1]:
            j += 1
        if A[k] < A[j]:
            return
        swap(A, k, j)
        k = j

def heap_sort(A):
    n = len(A) - 1
    for k in range((n + 1) // 2 - 1, -1, -1):
        sink(A, k, n)
    while n >= 1:
        swap(A, 0, n)
        n -= 1
        sink(A, 0, n)
```