Technical University Berlin

Vorlesungsmitschrift

Computerorientierte Mathematik II

gelesen von Prof. Dr. Michael Joswig im Sommersemester 2020

Inhaltsverzeichnis

Mitschrift von Viktor Glombik.

Zuletzt bearbeitet am 13. April 2022.

Abbildungsverzeichnis

1 Elementare Datenstrukturen

1.1 Dynamische Mengen

21.04.2020

Dynamische Mengen (in `python`-Sprech: `dictionaries`) sind in der Informatik Mengen, die sich z.B. während eines Algorithmus verändern, obwohl sie im Code oder in der mathematischen Beschreibung mit dem gleichen Symbol bezeichnet werden. Ein Beispiel für eine dynamische Menge ist eine Liste, der während eines Algorithmus Elemente hinzugefügt werden.

Die notwendige Operationen auf einer dynamischen Menge sind das Hinzufügen, das Entfernen und das Testen auf Enthaltensein. Oft werden komplizierte Operationen benötigt (vgl. Beispiel 1.1.1).

Die Elemente einer dynamischen Menge werden von Objekten dargestellt, welche Attribute besitzen. Die Attribute können gelesen und modifiziert werden, sofern ein Zeiger auf dieses Objekt existiert.

Beispiel 1.1.1 (Operationen auf einer dynamischen Menge S) Oft wird das Attribut key zur Identifizierung eines Objekts genutzt.

- `Search(S,k)` gibt einen Zeiger auf eine Element $x \in S$ mit `x.key = k` zurück und `None` sonst.
- `Insert(S,k)` Fügt das durch x referenzierte Element zu S hinzu.
- `Delete(S,k)` Löscht das durch x referenzierte Element aus S.
- `Maximum(S)` gibt einen Zeiger x auf das Element in S, für das `x.key` maximal ist. (`Minimum(S)` analog)
- `Successor(S,x)` gibt einen Zeiger auf das Element mit nächstgrößerem `key`. (`Predecessor(S,x)` analog)

Je nach Anwendung werden nicht alle Operationen benötigt oder sind gar nicht definiert. ◇

Bemerkung 1.1.2 (Das Attribut `key`) Das Attribut `key` eines Objekts kann mit dem Objekt übereinstimmen, z.B. $S = \mathbb{N}$, muss es jedoch nicht: Sind die Objekte Autos, könnte das Attribut `key` das Kennzeichen des Fahrzeugs sein.

Während auf den Objekten einer Menge keine Ordnung existieren muss (z.B. Autos), sind die Attribute `key` sortierbar (z.B. alphabetisch).

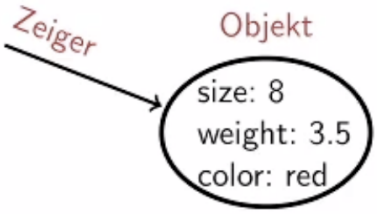


Abbildung 1: Ein Objekt mit drei Attributen und ein Zeiger.

```python
class myobject(object):
    def __init__(self, s, w, c):
        self.size = s
        self.weight = w
        self.color = c

x = myobject(8,3.5, "red")
```

Abbildung 2: Die Klasse `myobject` wird definiert und im Konstruktor (`__init__`) werden die Attribute initialisiert. Dann wird ein Zeiger auf ein Objekt x mit bestimmten Attributen erstellt.

Stacks und Queues

stack

Der stack (Stapel) ist eine sehr restriktive Datenstruktur, da man nicht auf alle Elemente zugriefen kann. Man stelle sich eine Stapel von Tellern vor: Nur das jeweils zuletzt hinzugefügte (`push`) Element kann entnommen (`pop`) werden (Last-in-First-out, kurz LIFO).

Die Laufzeit der Operation `push`, `pop` und `peek` ist konstant ($(O(1))$. Hierbei beschreibt `peek` das ansehen des zuletzt hinzugefügten Elements.

Bei der eng verwandten Datenstruktur queue (Warteschlange) kann nur das zuerst eingefügte (`enqueue`) Element gelöscht (`dequeue`) werden (First-in-First-out, FIFO). Wieder ist die Laufzeit alle Operationen konstant.

queue

Diese Datenstrukturen können also nicht viel, aber das sehr schnell.

Algorithmus 1 : Graphendurchmusterung: Algorithmus

Data : Ein ungerichteter Graph $G = (E, V)$, ein Knoten $r \in V$.

Result : Die Menge $R \subset V$ der von r aus erreichbaren Knoten, eine Menge $F \subset E$, sodass (R, F) ein Baum ist.

$R \leftarrow \{r\},\ F \leftarrow \emptyset,\ Q \leftarrow \{r\}$

while $Q \neq \emptyset$ **do**

  wähle $v \in Q$

  if $\exists e = \{v, w\} \in \delta(v)$ mit $w \notin R$ **then**

    $R \leftarrow R \cup \{w\},\ Q \leftarrow Q \cup \{w\},\ F \leftarrow F \cup \{e\}$

  else $Q \leftarrow Q \backslash \{v\}$

return (R, F)

Die Mengen R, F und Q sind dynamischen Mengen. Wir können Q anstatt als Liste nun als queue oder stack implementieren.

```python
class Queue:
    def __init__(self):
        self.items = []

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

    def enqueque(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

```python
def bfs(adj, r):
    R = [r]
    F = []
    Q = Queue()
    Q.enqueue(r)
    while not Q.isEmpty():
        v = Q.peek()
        i = 0
        while len(adj[v]) > i and adj[v][i] in R:
            i += 1
        if len(adj[v]) > i:
            w = adj[v][i]
            R.append(w)
            Q.enqueue(w)
            F.append((v,w))
        else:
            Q.dequeue()
    return (R, F)
```

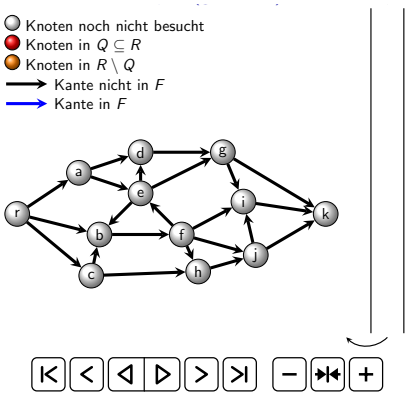


Abbildung 3: Breitensuche

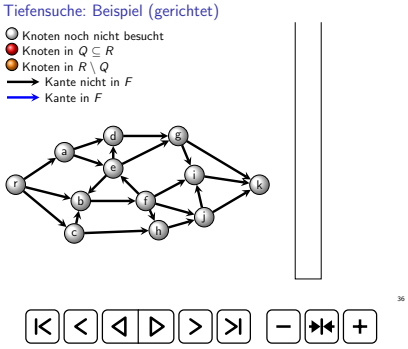


Abbildung 4: Tiefensuche

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

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

```python
def dfs(adj, r):
    R = [r]
    F = []
    Q = Stack()
    Q.push(r)
    while not Q.isEmpty():
        v = Q.peek()
        i = 0
        while len(adj[v]) > i and adj[v][i] in R:
            i += 1
        if len(adj[v]) > i:
            w = adj[v][i]
            R.append(w)
            Q.push(w)
            F.append((v,w))
        else:
            Q.pop()
    return (R, F)
```

Theorem 1.1.1: D

r Speicherbedarf von Breiten- und Tiefensuche ist linear in der Anzahl der Knoten und Kanten von G. Die Laufzeit ist ebenfalls linear, sofern die Datenstruktur für Q geeignet realisiert wird.

Bemerkung 1.1.3 (Gerichtete Graphen) Das vorgestellte Verfahren lässt sich leicht für gerichtete Graphen modifizieren, dann ist die Ausgabe (R, F) eine Aboreszenz mit Wurzel r. Der obige `python`-Code muss nicht verändert werden.

Bemerkung 1.1.4 (Anwendungen) Breiten- oder Tiefensuche kann für die Konstruktion eines aufspannenden Baums, einen Test auf Zusammenhang eines ungerichteten Graphen oder für das finden (gerichteter) Wege zwischen zwei Knoten.

27.04.2020

Die beiden Klassen `Queue` und `Stack` ähnelten sich sehr. Es ist also naheliegend, eine übergeordnete Klasse zu schreiben, welche die Gemeinsamkeiten der Klassen beinhaltet, sodass nur noch deren Unterschiede in den Klassen selbst definiert werden müssen:

```python
class AbstractSet:
    def __init__(self):
        self.items = []

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

    def insert(self, item):
        pass

    def pop(self):
        pass

    def peek(self):
        return self.items[-1]

    def size(self):
```

```
18          return len(self.items)
19
20  class Queue(AbstractSet):
21      def insert(self, item):
22          self.items.insert(0, item)
23
24      def pop(self):
25          return self.items.pop()
26
27  class Stack(AbstractSet):
28      def insert(self, item):
29          self.items.append(item)
30
31      def pop(self):
32          return self.items.pop()
```

Die ist ein Beispiel für die Polymorphie der objektorientierten Programmierung.

Polymorphie

Die Graphendurchmusterung, in der Graph als Adjazenzliste übergeben wird, sieht dann so aus

```
1  def traverse(adj, r, Q):
2      R = [r]
3      F = []
4      Q.insert(r)
5
6      while not Q.isEmpty():
7          v = Q.peek()
8          i = 0
9          while len(adj[v]) > i and adj[v][i] in R:
10             i += 1
11         if len(adj[v]) > i:
12             w = adj[v][i]
13             R.append(w)
14             Q.insert(w)
15             F.append((v,w))
16         else:
17             Q.pop()
18
19     return (R,F)
20
21 #Beispielaufruf
22
23 Q = Queue()
24 traverse([[1, 3], [0, 2], [1, 3], [0, 2]], 0, Q)
25
26 Q = Stack()
27 traverse([[1, 3], [0, 2], [1, 3], [0, 2]], 0, Q)
```

Die entsprechende Ausgaben sind `([0, 1, 3, 2], [(0, 1), (0, 3), (1, 2)])` und `([0, 3, 2, 1], [(0, 1), (1, 2), (2, 3)])`, welche sich in der Reihenfolge der Knoten und in den Spannbäumen unterscheiden. **TODO: Bild des Graphen zeichnen.**

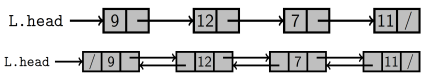


Abbildung 5: Graphische Notation für einfach und doppelt verkettete Listen

Einfach und doppelt verkettete Listen unterstützen beide die Operationen aus Beispiel 1.1.1, aber nicht unbedingt effizient:

	einfach verkettet	doppelt verkettet	sortiert
`Search(L,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`Insert(L,x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`Delete(L,x)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`Minimum(L)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`Maximum(L)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$ / $\mathcal{O}(n)$

Abbildung 6: Laufzeiten für eine Liste `L` der Länge n, wobei `k` für den `key` des Elements `x` steht.

Wir werden uns in diesem Kurs mit Datenstrukturen beschäftigen, bei denen das Einfügen und Löschen etwas länger dauert, dafür aber das Suchen aber viel schneller ist als mit doppelt verketteten Liste.

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root, newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(1, [newBranch, t, []])
    else:
        root.insert(1, [newBranch, [], []])
    return root

def insertRight(root, newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [newBranch, [], t])
    else:
        root.insert(2, [newBranch, [], []])
    return root

def getRootVal(root):
    return root[0]

def setRootVal(root, newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

In dem obigen Codebeispiel kann man das Konzept der Datenkapselung sehen: Es gibt Funktionen `get` und `set` für den Zugriff und das Setzen von Daten.

Datenkapselung

todo

```
class Node:
    def __init__(self, initdata):
        self.data = initdata
        self.next = None
        self.previous = None    # nur bei doppelt verketteter
     Liste

    def getData(self):
```

```
        return self.data

    def getNext(self):
        return self.next

    def setData(self, newdata):
        self.data = newdata

    def setNext(self, newnext):
        self.next = newnext

class UnorderedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    def add(self, item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp

    def search(self, item):
        current = self.head
        found = None

        while current != None and found == None:
            if current.getData() == item:
                found = current
            else:
                current = current.getNext()

        return found

    def remove(self, item):
        current = self.head
        previous = None
        found = False

        while not found:
            if current.getData() == item:
                found == True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

Binärbäume

Einfach verkettete Liste: ein Zeiger, doppelt verkettete Liste: zwei Zeiger, Binärbaum: drei Zeiger.

Abbildung 7: **TODO**

2 Heaps und Heapsort

Wir führen die neue Datenstruktur Heap ein, mit der wir ein neues Sortierverfahren, Heapsort realisieren.

Heap

Verfahren	Worst-Case	Average-Case	„in-place'"
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	ja
Mergesort	$\Theta(n \log(n))$	$\Theta(n \log(n))$	nein
Quicksort	$\Theta(n^2)$	$\Theta(n \log(n))$	ja
Heapsort	$\Theta(n \log(n))$	$\Theta(n \log(n))$	ja

Tabelle 1: Motivation (Überblick über einige vergleichsbasierte Sortierverfahren)

28.04.2020

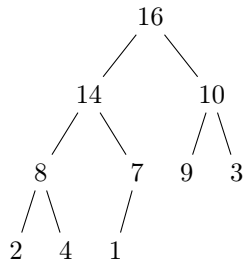
Die Datenstruktur MaxHeap (H) unterstützt die folgenden Operationen:

Operation	Laufzeit
`Maximum`(H)	$\mathcal{O}(1)$
`ExtractMax`(H)	$\mathcal{O}(\log(n))$
`Insert`(H, x)	$\mathcal{O}(\log(n))$
`ChangeKey`(H, x, k)	$\mathcal{O}(\log(n))$,

wobei `ExtractMax`(H) einen Zeiger x auf ein Element in H liefert, sodass x`.key` maximal ist und x aus H löscht und `ChangeKey`(H, x, k) x`.key` $= k$ setzt.

Bemerkung 2.0.1 (MinHeap) Analog gibt es eine Datenstrukturen MinHeap mit Operationen `Minimum`(H) (anstatt von `Maximum`(H)) und `ExtractMax`(H) (anstatt von `ExtractMax`(H))

Beispiel 2.0.2 (Heaps: Interpretation von Arrays als Binärbäume) Man identifiziere die Elemente eines Heaps mit ihren Schlüsseln. Die Datenstruktur Heap besteht aus einem Array, welches wir als fast vollständigen Binärbaum interpretieren. Gegeben sei zum Beispiel den Array `[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]`. Indizieren wir die Elemente des Arrays, beginnend mit 1, so ist die Mutter des i-ten Eintrags der $\lfloor \frac{i}{2} \rfloor$-te Eintrag, sein linkes Kind der $2i$-te Eintrag und sein rechtes Kind der $2i + 1$-te Eintrag:



◇

Definition 2.0.3 (MaxHeap-Eigenschaft)
Ein Array A der Größe n besitzt die MaxHeap-Eigenschaft, wenn

MaxHeap-Eigenschaft

$$A[\texttt{Parent}(i)] \geqslant A[i]$$

für alle $i \in \{1, \ldots, n\}$ gilt. Der dazugehörige Binärbaum heißt dann MaxHeap.

Bemerkung 2.0.4

- In einem MaxHeap A steht das größte Element in der Wurzel $A[1]$.
- Entlang eines Weges von der Wurzel zu einem Blatt fallen die Elemente monoton.
- Die Höhe des Baumes (die Länge des längsten Weges von Wurzel zu einem Blatt) ist $\lfloor \log(n) \rfloor$.
- Ist A absteigend sortiert, so hat A die MaxHeap-Eigenschaft.

Algorithmus 2 : `Max-Heapify`(A, i)

$\ell =$`Left`(i), $r =$`Right`(i),
if $\ell \leqslant n$ und $A[\ell] > A[i]$ **then**
  `largest` $\leftarrow \ell$
else `largest` $\leftarrow o$
if $r \leqslant n$ und $A[r] > A[largest]$ **then** `largest` $\leftarrow r$
if largest $\neq i$ **then**
  `swap`$(A[i], A[$`largest`$])$
  `Max-Heapify`$(A,$ `largest`$)$

Lemma 2.0.5 ((Wieder-)Herstellen der MaxHeap-Eigenschaft) *Erfüllen Die Binärbäume unterhalb der Knoten* `Left`(i) *und* `Right`(i) *die MaxHeap-Eigenschaft, so erfüllen nach dem Aufruf von* `Max-Heapify`(A, i) *der Binärbaum unterhalb des Knotens i die MaxHeap-Eigenschaft. Die Laufzeit ist linear in der Höhe des Binärbaums unterhalb von i.*

Beweis. ... □

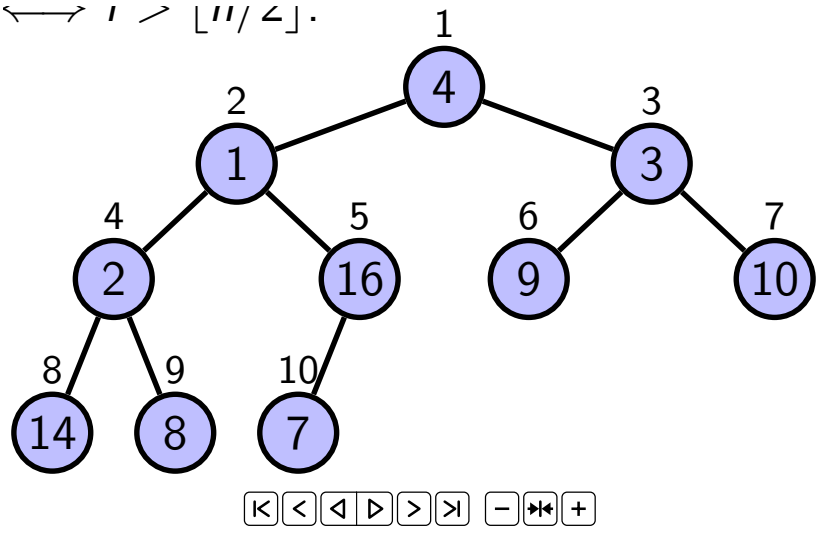


Abbildung 8: BuildMaxHeap

Wenden wir `Max-Heapify`(A, i) wiederholt auf die Knoten i, deren Teilbäume bereits die MaxHeap-Eigenschaft erfüllen, können wir einen Heap aufbauen. Wir iterieren dazu beispielsweise über alle Nicht-Blätter des

Binärbaumes von unten nach oben. Ein Knoten i ist genau dann ein Blatt, wenn $i > \lfloor \frac{n}{2} \rfloor$.

Algorithmus 3 : `Build-Max-Heap`(A)

for $i = \lfloor \frac{n}{2} \rfloor$ downto 1 **do**
    `Max-Heapify`(A, i)

Theorem 2.0.1

Nach Aufruf von Build-Max-Heap(A) besitzt A die MaxHeap-Eigenschaft. Die Laufzeit beträgt $\mathcal{O}(n)$.

`Build-Max-Heap`(A) **while** $n > 1$ **do**
    `swap`$(A[1], A[n])$;
    $n \leftarrow n - 1$;
    `Max-Heapify`(A, i)

Theorem 2.0.2

Heapsort sortiert ein Array A der Länge n in Zeit $\Theta(n \log(n))$ und benötigt dafür nur konstant viel zusätzlichen Speicher („in-place“).

Bemerkung 2.0.6 Die Laufzeit von Heapsort ist empirisch etwa vergleichbar mit Mergesort.

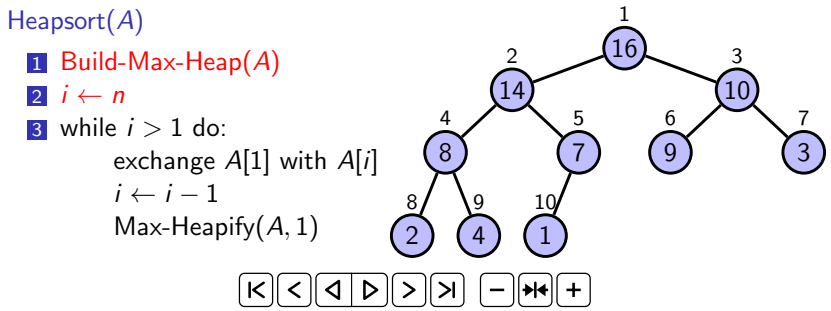


Abbildung 9: Heapsort

04.05.2020

```
def Maximum(A):
    return A[1]
```

```
def ExtractMax(A):
    swap(A[1], A[n]) n ← n − 1;
    Max-Heapify(A, 1);
    return A[n + 1]
```

```
def Insert(A, k):
    n → n + 1;
    A[n] ← −∞;
    IncreaseKey(A, n, k)
```

```
def IncreaseKey(A, i, k):
    if k < A[i] then return „too small“;
    A[i] ← k;
    while i > 1 und A[Parent(i)] < A[i] do
        swap(A[i], A[Parent(i)];
        i ←Parent(i)
```

```
def DecreaseKey(A, i, k):
    if k > A[i] then return „k too large“;
    A[i] ← k;
    Max-Heapify(A, i)
```

Bemerkung 2.0.7 (FIBONACCI-Heaps) FIBONACCI-Heaps sind eine Weiterentwicklung von Heaps. Die *amortisierte* Laufzeit einiger Operationen ist besser als bei Heaps.

Operation	MinHeaps (worst-case)	FIBONACCI-Heaps (amortisiert)
Minimum(H)	$\Theta(1)$	$\Theta(1)$
ExtractMin(H)	$\Theta(\log(n))$	$\Theta(\log(n))$
Insert(H, x)	$\Theta(\log(n))$	$\Theta(1))$
DecreaseKey(H, x, k)	$\Theta(\log(n))$	$\Theta(1)$

Eine Anwendung von MinHeaps ist Prim's Algorithmus.

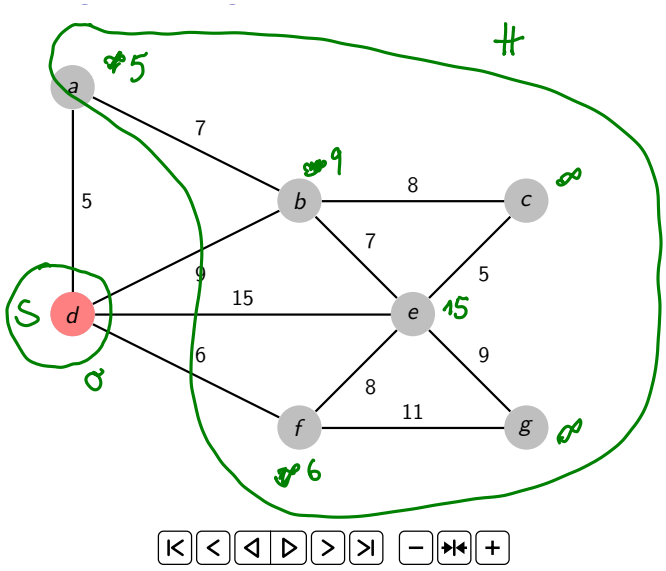


Abbildung 10: Wiederholung: Algorithmus von PRIM

Sei im folgenden H eine Prioritätswarteschlange.

---

for $v \in V$ **do**
    v.key $\leftarrow \infty$, v.pred $\leftarrow$ None, v.inTree $\leftarrow$ false,
    Insert(H, v)
wähle $v_0 \in V$ beliebig
DecreaseKey$(H, v_0, 0)$ **while** not isEmpty(H) **do**
    $v \leftarrow$ ExtractMin(H), v.inTree $\leftarrow$ True
    for $vw \in \delta(v)$ **do**
        if $c(vw) < w$.key und not w.inTree **then**
            DecreaseKey$(H, w, c(vw))$, w.pred $\leftarrow v$

Die Anzahl der Operationen Insert(H, v) und ExtractMin(H) sind jeweils $|V|$, von DecreaseKey(H, v, c) höchstens $|E| + 1$.

Korollar 2.0.8

Eine Implementation von ***Prims*** *Algorithmus mit einem MinHeap H besitzt die Laufzeit $\mathcal{O}(|E| \log(|V|))$. Mit Hilfe eines* FIBONACCI*-MinHeaps kann die Laufzeit auf $\mathcal{O}(|E| + |V| \log(|V|))$ verbessert werden.*

3 Kürzeste Pfade in Digraphen

Sei $G := (V, E)$ ein Digraph (*directed graph*) mit Kantenlängen $(c_e)_{e \in E}$.

Definition 3.0.1 ((geschloss.) Pfad und seine Länge)
Ein (v_0-v_k-)Pfad (Kantenzug) P, für $k \in \mathbb{N}$, ist eine Folge

$$(v_0, a_1, v_1, a_2, \ldots, v_{k-1}, a_k, v_k)$$

und $a_i := (v_{i-1}, v_i) \in E$ für $i \in \{1, \ldots, k\}$. Die Länge von P ist $c(P) := \sum_{i=1}^{k} c_{a_i}$. Ist $v_0 = v_k$, so ist P geschlossen.

Pfad

Definition 3.0.2 (Weg, Kreis)
Ein Weg (Kreis) ist ein (geschlossener) Pfad, sodass gilt:

$$v_i \neq v_j \qquad \text{für alle } 0 \leqslant i < j \overset{(<)}{\leqslant} k.$$

Weg

3.1 Das kürzeste-Pfade-Problem

Gegeben seien ein Digraph $G := (V, E)$, ein Startknoten $r \in V$ und Kantenlängen c_e für $e \in E$. Die Aufgabe ist es, für alle $v \in V$ einen $r - v$-Pfad minimaler Länge zu finden (falls einer existiert).

Bemerkung 3.1.1 Die Existenz eines solchen Pfades kann man z.B. mit Breitensuche überprüfen.

Manchmal ist es hilfreich (vgl. später), dafür zu sorgen, dass alle solche Pfade existieren. Dies kann durch z.B. durch zulassen von ∞ (oder hinreichend große reelle Zahl) als Kantengewicht erreicht werden, indem man diese Kanten „künstlich“ hinzufügt.

Bemerkung 3.1.2 Ähnliche Aufgaben sind, zwischen zwei festen Knoten einen kürzesten Pfad zu finden (z.B. Navigation) oder für je zwei verschiedene Knoten den kürzesten Pfad zischen ihnen zu finden.

Es stellt sich heraus, dass das „Navigationsproblem“ am besten durch studieren des oben genannten Problems zu lösen ist. Die zwei Aufgabe ist Anwendung des obigen Problems für jeden Knoten als Startknoten.

Die grundlegende Idee aller Verfahren zur Berechnung kürzester Pfade liefert das folgende

Lemma 3.1.3 (Hauptlemma)
Für $v \in V$ sei y_v die Länge eines kürzesten r-v-Pfades. Dann gilt für alle $(v, w) \in E$

$$y_v + c_{(v,w)} \geqslant y_w.$$

Beweis. Klar? □

Die Ungleichung besagt, dass von den Wegen von r nach w alle Umwege über v mindestens so lang sind, wie der kürzeste r-w-Weg.

05.05.2020

Bemerkungen 3.1.4 (Elementare Fakten zu kürzesten Pfaden)

- **Teilpfade kürzester Pfade sind kürzeste Pfade.** Gäbe es einen kürzen Teilpfad, können man den entsprechende Teilpfade ersetzen und erhielte so einen kürzeren „Gesamtpfad“, was einen Widerspruch

darstellt.

- **Enthält ein kürzester r-v-Pfad einen geschlossenen Kantenzug (z.B. einen Kreis), so hat dieser Länge Null.** Hätte der Kantenzug positive Länge, so könnten man ihn aus dem Pfad entfernen, um einen kürzeren zu erhalten, hätte er negative Länge, so gäbe es keinen kürzesten Pfad.
- **Enthält ein kürzester r-v-Pfad enthält immer einen kürzesten r-v-Weg gleicher Länge.** Folgt direkt aus dem vorherigen Stichpunkt.
- **Gibt es einen kürzesten r-v-Pfad für alle $v \in V$, so existiert ein Kürzeste-Wege-Baum, d.h. ein gerichteter Baum T mit Wurzel r, sodass der eindeutige r-v-Weg in T ein kürzester r-v-Pfad in G.** Nach Voraussetzung ist der aus den kürzesten r-v-Wegen bestehende Teilgraph zusammenhängend. Nach dem zweiten Stichpunkt ist er auch kreisfrei und somit ein Baum.

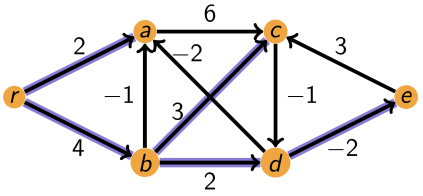


Abbildung 11: Dieser Graph enthält keine negativen Kreise und es gibt kürzeste r-v-Wege für $v \in \{a, b, c, d, e\}$. Die violett markierten Kanten bilden einen Kürzeste-Wege-Baum, welcher durch die Vorgänger-Vorschriften $p(r) =$ `None`, $p(a) = p(b) = r$, $p(c) = p(d) = b$ und $p(e) = d$ eindeutig kodiert ist.

Zulässige Potenziale

Definition 3.1.5 (Zulässiges Potenzial)
Ein Vektor $y \in \mathbb{R}^{|V|}$ heißt zulässiges Potenzial, wenn gilt

zulässiges Potenzial

$$y_v + c_{(v,w)} \geqslant y_w \qquad \text{für alle } (v, w) \in E.$$

Die Bedingung kann als $c_{(v,w)} \geqslant y_w - y_v$ umgeschrieben werden.

Bemerkung 3.1.6 Sind x, y zulässige Potenziale, so ist es aus $\max(x, y)$: Für $(v, w) \in E$ gilt $x_v + c_{(v,w)} \geqslant x_w$ und $y_v + c_{(v,w)} \geqslant y_w$ und somit

$$\max(x_w, y_w) \leqslant \max(x_v + c_{(v,w)}, y_v + c_{(v,w)}) = \max(x_v, y_v) + c(v, w).$$

Lemma 3.1.7 (Potenziale und Pfadlängen)
Ist y ein zulässiges Potenzial mit $y_r = 0$ und P ein r-v-Pfad, so ist $y_v \leqslant c(P)$.

Beweis. Für $P := (v_0, a_0, v_1, \ldots, a_k, v_k)$ mit $v_0 := r$ und $v_k := v$ gilt

$$c(P) = \sum_{i=1}^{k} c_{a_i} \geqslant \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} = y_v - y_r = y_v. \qquad \square$$

Korollar 3.1.8
*Sei G ein stark zusammenhängender **(WHY??)** Digraph ohne negative Kreise und $y \in \mathbb{R}^{|V|}$ ein zulässiges Potenzial. Ist $y_t \geqslant 0$, so gilt $-y_v \leqslant \operatorname{dist}(v, t)$ für jeden $v \in V$, wobei $\operatorname{dist}(v, t)$ die Länge des kürzesten v-t-Weges ist.*

in eine stark zusammenhängenden Graph gibt es zwischen je zwei Knoten u und v eine Kante von u nach v und eine Kanten von v nach u.

Beweis. Nach dem obigen Beweis gilt für den kürzesten v-t-Weg P von v nach t

$$c(P) \geqslant y_t - y_v \geqslant -y_v. \qquad \square$$

Bemerkung 3.1.9 (o.B.d.A. $y_r = 0$)
Ist $(y_v)_{v \in V}$ ein zulässiges Potenzial, so ist auch $(y_v - y_0)_{v \in V}$ ein zulässiges Potenzial, welches nun aber die Voraussetzung des obigen Lemmas erfüllt.

Korollar 3.1.10 (Potenziale und Pfadlängen)
Ist y ein zulässiges Potenzial mit $y_r = 0$ und P ein r-v-Pfad mit $c(P) = y_v$, so ist P ein kürzester r-v-Pfad.

Algorithmus 4 : Algorithmus von Ford

Result : Kürzeste-Wege-Baum, wie er in Abbildung 11 kodiert ist.

for $v \in V$ **do**
  $y_v \leftarrow \infty$, $p(v) \leftarrow$ `None`
$y_r \leftarrow 0$
while $\exists (v,w) \in E$ with $y_w > y_v + c_{(v,w)}$ **do**
  $y_w \leftarrow y_v + c_{(v,w)}$, $p(w) \leftarrow v$

Man beachte, dass $y_w > y_v + c_{(v,w)}$ genau die Negation der Potenzialbedingung ist. Der Algorithmus terminiert nicht, wenn der Graph negativen Kreise (d.h. Kreise negativer Länge) enthält.

Lemma 3.1.11 (Korrektheit des Ford Algorithmus)
Enthält G keine negativen Kreise, so gilt während des gesamten Algorithmus

1. *Ist $y_v \neq \infty$, so entspricht y_v der Länge eines r-v-Weges.*
2. *Ist $p(v) \neq$ `None`, definiert p einen r-v-Weg der Länge höchstens y_v.*

Beweis. 1. Sei $v \in V$ mit $y_v = \infty$, sodass eine Kante von r nach v existiert. Dann gilt $y_v > y_r + c_{(r,v)} = c_{(r,v)}$, also setzt der Algorithmus $y_v = c_{(r,v)}$, also auf die Länge des r-v-Pfades. Für die benachbarten Knoten von v folgt und somit induktiv für alle Knoten $q \in V$ folgt nun, dass y_q die Länge eines r-q-Weges angibt.

2. Ist $p(v) = w$, so existiert ein Weg von r nach w und kann somit zu eine r-v-Weg fortgesetzt werden. Die Aussage folgt somit induktiv. □

Theorem 3.1.1: Ford Algorithmus Korrektheit

Enthält G keine negativen Kreise, so terminiert der Algorithmus nach endlich vielen Iterationen. Am Ende ist y ein zulässige Potenzial mit $y_r = 0$ und für jede $v \in V$ definiert p einen kürzesten r-v-Weg der Länge y_v.

Beweis. Da keine negativen Kreis existieren, gibt es kürzeste Wege. Der Algorithmus überprüft die Bedingung von Lemma 3.1.3 und somit ist y ein zulässiges Potenzial. Somit folgt der Beweis aus Lemma 3.1.11. □

Theorem 3.1.2: Zulässige Potenziale und negative Kreise

Ein Digraph $G := (V, E)$ mit Kantenlängen $c \in \mathbb{R}^{|E|}$ besitzt genau dann ein zulässiges Potenzial, wenn er keinen negativen Kreis enthält.

Beweis. „$\Longrightarrow$“: Sei y ein zulässiges Potenzial. Nach Bemerkung 3.1.9 können wir $y_r = 0$ annehmen. Nach Korollar 3.1.10 existiert ein kürzester

r-v-Pfad für alle $v \in V$. Nach Bemerkung 3.1.4 gibt es keine negativen Kreise.

„ $\Longleftarrow$ “: Existieren keine negativen Kreis, so liefert der Ford Algorithmus ein zulässiges Potenzial nach Satz 3.1.1. □

Bemerkung 3.1.12 (Pfade vs. Wege)
Gibt es einen r-v-Pfad, aber keinen kürzesten r-v-Pfad, so gibt es beliebig kurze r-v-Pfade, die jedoch keine r-v-Wege sind.

In dieser Situation ist es schwierig (im Sinne der Komplexitätstheorie), eine kürzesten r-v-Weg zu finden

Lemma 3.1.13 (Laufzeit von Fords Algorithmus)
Ist c ganzzahlig und existieren keine negativen Kreise, so terminiert Fords Algorithmus nach höchstens $(2\max_{e\in E}|c_e|+1)|V|^2$ Iterationen.

Beweis. Hausaufgabe. □

Der Algorithmus von Bellman-Ford

Wir bemerken, dass, da keine Art vorgegeben ist, die Kanten zu durchlaufen, bestimmte Potenziale oft verändert werden. Somit hängt die Anzahl der Iterationen von der gewählten Kantensequenz (hier: $\mathcal{S} := (f_k)_{k=1}^{\ell}$) ab. Wir wollen nun Fords Algorithmus verfeinern.

Ein Pfad P ist eingebettet in $\mathcal{S}$, wenn die Sequenz von P eine Teilsequenz von $\mathcal{S}$ ist.

Lemma 3.1.14
Ist ein r-v-Pfad in $\mathcal{S}$ eingebettet, so ist $y_v \leqslant c(P)$ nachdem der Algorithmus $\mathcal{S}$ abgearbeitet hat.

Beweis. Folgt direkt aus dem Beweis von Lemma 3.1.7, da nach abarbeiten der Sequenz der Algorithmus lokal die Potenzialbedingung hergestellt hat. □

Wir wollen als ein kurze Sequenz $\mathcal{S}$, sodass für alle $v \in V$ ein kürzester r-v-Pfad in $\mathcal{S}$ eingebettet ist.

Die grundlegende Idee des Bellmann-Ford Algorithmus ist die folgende: für eine beliebige Sortierung $\mathcal{S}_i$ von E ($i \in \{1,\dots,|V|-1\}$) ist jeder Weg in $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{|V|-1}$ eingebettet. Somit kann der Kürzeste-Wege-Algorithmus mit einer Laufzeit von $\mathcal{O}(|V|\cdot|E|)$ realisiert werden. Bellmann-Ford Algorithmus

Algorithmus 5 : Bellmann-Ford Algorithmus

initialisiere y und p wie im Ford Algorithmus;
for $i \in \{1,\dots,|V|-1\}$ **do**
  for $(v,w) \in E$ **do**
    if $y_w > y_v + c_{(v,w)}$ **then**
      $y_w \leftarrow y_v + c_{(v,w)}$, $p(w) \leftarrow v$

Theorem 3.1.3: Bellmann-Ford korrekt

Der Bellmann-Ford Algorithmus besitzt die Laufzeit $\mathcal{O}(|V|\cdot|E|)$. Ist y am Ende ein zulässiges Potenzial, so beschreibt p für alle $v \in V$ ein kürzesten r-v-Pfad. Andernfalls enthält G einen negativen Kreis.

Azyklische Digraphen und topologische Sortierungen

11.05.202

Sei fortan $G := (V, E)$ ein Digraph.

Definition 3.1.15 (Topologische Sortierung)

Eine Sortierung $v_1, v_2, \ldots, v_{|V|}$ von V mit $i < j$ für alle $(v_i, v_j) \in E$ heißt topologische Sortierung. Besitzt G eine topologische Sortierung, so ist G azyklisch.

topologische Sortierung

Bemerkungen 3.1.16

- Ein Diagraph ist genau dann azyklisch, wenn er keinen Kreis enthält.
- Ist G azyklisch, kann man eine topologische Sortierung in $\mathcal{O}(|V| + |E|)$ finden. (Hausaufgabe)
- Sei G azyklisch und $\mathcal{S}$ eine Sortierung von E, sodass (v_i, v_j) vor (v_k, v_ℓ) kommt, wenn $i < k$. Dann ist jeder Pfad in G in $\mathcal{S}$ eingebettet.

Theorem 3.1.4

Das Kürzeste-Pfade-Problem für azyklische Digraphen kann in $\mathcal{O}(|V| + |E|)$ gelöst werden.

Beweis. 6:18 - 9:06 □

Wir kombinieren nun den Ford-Algorithmus und Prioritätswarteschlangen.

Beispiel 3.1.17 (Algorithmus von Dijkstra)

Betrachte den Spezialfall nichtnegativer Kantenlängen, d.h. $c_a \geqslant 0$ für alle $a \in E$. (Ohne negative Kantengewichte gibt es keine negativen Kreise)
Im folgenden sei H ein MinHeap, wobei die Objekte die Knoten $v \in V$ sind und die Schlüssel y_v.

---

initialisiere r (Startknoten), y (Potenzial) und p (Kürzeste-Wege-Baum per Rückverweis) wie im Ford-Algorithmus
for $v \in V$ **do** `Insert`(H, v)
while not `isEmpty`(H) **do**
  $v \leftarrow$ `ExtractMin`(H)
  for $(v, w) \in \delta^+(v)$ **do**
    if $y_w > y_v + c_{(v,w)}$ **then**
      `DecreaseKey`$(H, w, y_v + c_{(v,w)})$, $p(w) \leftarrow v$

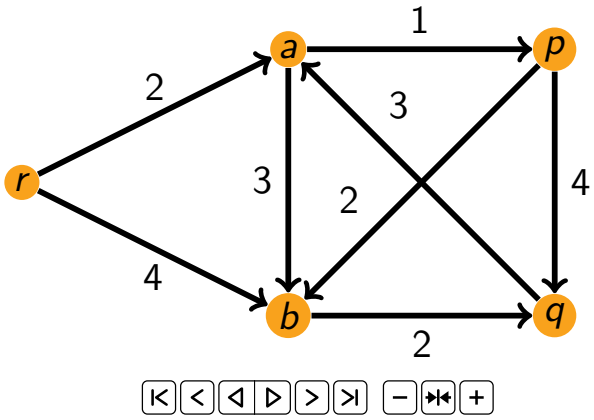


Abbildung 12: Dijkstra-Algorithmus

Lemma 3.1.18
Für $w \in V$ sei y'_w der Wert von y_v, wenn w aus H entfernt wird. Wird u vor v aus H entfernt, so ist $y'_u \leqslant y'_v$.

Beweis. In einem Schritt des Algorithmus wir $y_w \leftarrow y_v + c(v,w)$ gesetzt. Es ist $y_v = y'_v$ und $c(v,w) \geqslant 0$ und somit gilt $y_w \geqslant y'_v$. □

Theorem 3.1.5: Dijkstra-Algorithmus

Ist $c \geqslant 0$, so löst der Dijkstra-Algorithmus das Kürzeste-Pfade-Problem korrekt. Seine Laufzeit ist $\mathcal{O}(|V|^2)$ oder $\mathcal{O}(|E|\log(|V|))$ (MinHeap) oder sogar $\mathcal{O}(|E| + |V| + \log(|V|))$ (Fibonacci-MinHeap).

Bemerkung 3.1.19 Bei Navigation (Straßennetz) ist meist $|E| \in \mathcal{O}(|V|)$ und somit läuft der Algorithmus von Dijkstra in $\mathcal{O}(n\log(n))$ für $n := |V|$.

Beweis. Korrektheit. Wir zeigen, dass zum Zeitpunkt $v \leftarrow$ ExtractMin(H) gilt $y_v = y'_v =$ die Länge eines kürzesten Weges von r nach v.

Wir führen einen Beweis per Induktion nach der Reihenfolge nach der die Knoten entnommen werden. Induktionsanfang: Für den Knoten r gilt $y_r = 0$. Induktionsschritt: Die Länge eines kürzesten Weges von r nach v führt über einen Knoten u mit $y'_u \leqslant y'_v$ folgt aus dem obigen Lemma.

Laufzeit: $\mathcal{O}(|V|^2)$ denn: $|V|$ Iterationen der while-Schleife, und in jeder Iteration werden höchstens n Knoten „upgedated“.

$\mathcal{O}(|E| \cdot \log|V|$ denn: `ExtractMin` wir n mal ausgeführt $\implies \mathcal{O}(n \cdot \log n)$ und `DecreaseKey` benötigt $n \cdot \log n$. Die Initialisierung ist in $\mathcal{O}(n)$ und `Insert`(H, v) für alle v kosten zusammen $\mathcal{O}(n)$. □

Bemerkung 3.1.20 Die for-Schleife kann so modifiziert werden, dass nur Kanten (v,w) mit $w \in H$ betrachtet werden.

4 Verwaltung disjunkter Mengen

Wiederholung (Kruskal-Algorithmus.)

Algorithmus 6 : Algorithmus von Kruskal.

Data : zusammenhängender Graph $G = (V, E)$ mit $V = \{v_1, \dots, v_n\}$, $E := \{e_1, \dots, e_m\}$ und Gewichten $w \colon E \to \mathbb{R}_{\geqslant 0}$.

Result : MST in G.

Sortiere Kanten: $w(e_1) \leqslant \dots \leqslant w(e_m)$

$F \leftarrow \{\{v_1\}, \dots, \{v_n\}\}$, $B \leftarrow \varnothing$

for $i \leftarrow 1$ to m **do**

  if e_i verbindet $C, D \in F$, $C \neq D$ **then**

    $F \leftarrow (F \backslash \{C, D\}) \cup \{C \cup D\}$

    $B \leftarrow B \cup \{e_i\}$

return B

Die genaue Komplexität dieses Algorithmus hängt wesentlich von den verwendeten Datenstrukturen ab!

Hier ist der Graph G als Inzidenzliste gegeben, B realisiert als Liste und F realisiert als Array (jeder Knoten zeigt auf seine Komponente). Die Kosten sind dann: Sortieren $\mathcal{O}(m \log(m))$, Initialisierung $\mathcal{O}(n)$ und dann m Durchläufe von Kante testen $\mathcal{O}(1)$, F aktualisieren $\mathcal{O}(n)$ und B aktualisieren $\mathcal{O}(1)$. Der Gesamtaufwand ist also $\mathcal{O}(mn + m \log(m))$ $(= \mathcal{O}(mn)$ für vernünftige Graphen).

Wie können wir das verbessern? Wir benutzen Union-Find, eine Datenstruktur zur Verwaltung einer Familie F disjunkter nichtleerer Teilmengen einer Grundmenge V. Jede Teilmenge $X \in F$ besitzt dabei einen Repräsentanten. Union-Find

Diese Dantestruktr stellt die folgenden Operationen bereit

- `Make-Set`(x): $F \leftarrow F \cup \big\{\{x\}\big\}$, wobei zuvor $x \notin \bigcup_{X \in F} X$.
- `Union`(x, y): $F \leftarrow (F \backslash \{S_x, S_y\}) \cup \{S_x \cup S_y\}$ mit $x \in S_x \in F$, $y \in S_y \in F$.
- `Find-Set`(x): liefert einen Zeiger auf den Repräsentanten $X \in F$ von $x \in X$.

Wir nehmen an, dass $|V| = n$ ist, also n `Make-Set`-Operationen, und insgesamt $\mathcal{O}(m)$ `Make-Set`-, `Union`- und `Find-Set`-Operationen, wobei $m \in \Omega(n)$ (genauer: $m \leqslant n - 1$, da G zusammenhängend).

Sortiere Kanten: $w(e_1) \leqslant \dots \leqslant w(e_m)$

for $x \in V$ **do** `Make-Set`(x)

$B \leftarrow \varnothing$

for $i \leftarrow 1$ to m **do**

  if `Find-Set`$(x_i) \neq$ `Find-Set`(y_i) **then**

    `Union`(x_i, y_i)

    $B \leftarrow B \cup \{x_i y_i\}$

Die Anzahl der Operationen ist: `Make-Set`: n, `Find-Set` $2m$, `Union`: $n-1$.

Beispiel 4.0.1 (Implementation als Array)
Seien $V := \{1, \ldots, n\}$ und A ein Array der Länge n. Setze anfangs $A[i] \leftarrow 0$ für $i \in V$.

```
def Make-Set(x):
    A[x] ← x
def Find-Set(x):
    return A[x]
def Union(x, y):
    c ← A[x]
    for i ← i to n do
        if A[i] = c then
            A[i] ← A[y]
```

Die Laufzeiten für `Make-Set` und `Find-Set` also $\mathcal{O}(1)$ und von `Union` $\mathcal{O}(n)$. Also ist die Worst-Case Laufzeit des Algorithmus von Kruskal $\mathcal{O}(mn)$ (genauer: $\mathcal{O}(m + n^2)$, da höchstens $n - 1$ echte `Union`-Operationen). ◇

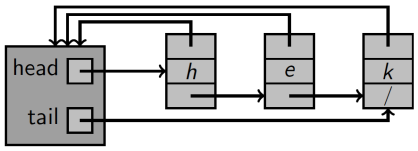


Abbildung 13: Implementation mit einfach verketteten Listen

Beispiel 4.0.2 (Implementation mit einfach verketteten Listen)
Neben den normalen Zeigern auf das nächste Objekt benutzen wir zwei weitere Verwaltungszeiger: jedes Element der Liste hat einen Verweis auf den Kopf der Liste und es gibt einen Verweis von dem Kopf der Liste zu ihrem Schwanz. Eine Teilmenge $X \in F$, z.B. $X = \{e, h, k\}$ kann man dann so darstellen, wie im Bild rechts.

Der Repräsentant der Teilmenge ist das erste Objekt der Liste (hier h). Die Operationen `Make-Set`(x) und `Find-Set`(x) (ein Zeiger vom x zum head und einer vom head zum Repräsentanten) brauchen in dieser Implementation $\mathcal{O}(1)$. Die Laufzeit von `Union`(x, y) ist $\mathcal{O}(n)$.

Insgesamt erhält man wieder eine Laufzeit von $\mathcal{O}(m + n^2)$. ◇

Bemerkung 4.0.3 (Beschleunigung der Operation `Union`(x, y))
Zunächst werden die Repräsentanten von x und y gefunden. Nun kommt der tail-Zeiger ins Spiel. Auf das letzte Element der einen Liste können wir in konstanter Zeit zugreifen und den Nachfolger-Zeiger auf das erste Element der anderen Liste setzen. Als nächstes wird der tail-Zeiger der ersten Liste auf das letzte Element der zweiten Liste gesetzt (wieder konstanter Aufwand). Als letztes werden die head-Zeiger der Elemente aus der zweiten Liste auf die erste Liste umgebogen (linearer Aufwand in der Anzahl der Einträge der zweiten Liste). DAs ist auch der gesamte Aufwand dieser Operation.

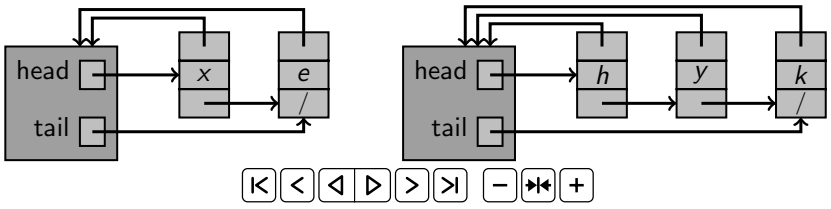


Abbildung 14: Beschleunigung der Operation `Union`(x, y)

Somit können wir `Union` beschleunigen, indem wir (anders als in der Animation) die kürzere an die längere Liste anhängen.

Theorem 4.0.1

Die Gesamtlaufzeit für n `Make-Set`-Operationen und insgesamt $\mathcal{O}(m)$ `Make-Set-`, `Union-` und `Find-Set`-Operationen beträgt bei dieser Implementation $\mathcal{O}(m + n \log(n))$.

Beweis. Aufteilungsbeschleunigungssatz. □

12.05.2020

Fast lineare Laufzeit

Für Union-Find gibt es eine speziell geeignete Baum-Datenstruktur.

Theorem 4.0.2: Fast lineare Laufzeit

Damit beträgt die Laufzeit für n `Make-Set`-Operationen und m `Make-Set-`, `Union-` und `Find-Set`-Operationen insgesamt $O(m \log^*(n))$.

Dabei ist $\log^*(n)$ (zur Basis zwei) für $'n > 0$ rekursiv definiert ist als

$$\log^*(n) = \begin{cases} 0, & \text{wenn } n \leqslant 1, \\ 1 + \log^*(\log(n)), & \text{sonst.} \end{cases}$$

Folglich ist

$$\log^*(n) = \begin{cases} 0, & \text{wenn } 0 < n \leqslant 1, \\ 1, & \text{wenn } 1 < n \leqslant 2, \\ 2, & \text{wenn } 2 < n \leqslant 4, \\ 3, & \text{wenn } 4 < n \leqslant 16, \\ 4, & \text{wenn } 16 < n \leqslant 2^{16}, \\ 5, & \text{wenn } 2^{16} < n \leqslant 2^{2^{16}}, \end{cases}$$

d.h. $\log^*(n)$ ist für alle praktisch relevanten n eine kleine Konstante: $\mathcal{O}(m \log^*(n)) \sim \mathcal{O}(m)$, wobei $\sim$ bedeutet, dass man es in der Praxis nicht unterscheiden kann.

5 Huffman-Kodierung und Datenkompression

Wir wollen Daten unter Verwendung von möglichst wenig Speicherplatz speichern, also Kompression auf das Originalformat der Daten anwenden, um eine komprimiertes Format zu erhalten.

Hierbei gibt es zwei Ansätze:

1. Verlustfreie Datenkompression: Originaldaten können vollständig aus komprimierten Daten zurückgewonnen werden, z.B. zip.
2. Verlustbehaftete Datenkompression: Originaldaten können nur näherungsweise zurückgewonnen werden, z.B. jpeg, mpeg.

Im Folgenden beschäftigen wir uns mit verlustfreier Kompression von Textdaten mit Binärcodes.

Wir arbeiten unter der Annahme, dass die Datei aus einzelnen Zeichen eines Alphabets (o. Zeichensatz) C besteht, z.B: ASCII, Unicode ...

Definition 5.0.1
Ein Code heißt Blockcode, wenn alle Zeichen als $0-1$-Strings („Bits“) fester Länge kodiert werden, z.B. ASCII mit Länge acht.

Blockcode

Der Vorteil von Blockcode ist einfache (De)Kodierung.

Beispiel 5.0.2 (Blockcode)
Die folgende Tabelle zeigt einen Blockcode mit Länge drei.

a	b	c	d	e	f	g	␣
000	001	010	011	100	101	110	111

Warum reicht Blockcode nicht aus? Da verschiedene Zeichen mit verschiedenen Häufigkeiten auftreten, könnte man Speicherplatz sparen, in dem man Code-Wörter variabler Länge schafft: häufige Zeichen bekommen kurze Code-Wörter und seltene Zeichen lange. Nun ist aber die Dekodierbarkeit nicht mehr notwendigerweise gegeben, betrachte z.B. die folgende Codierung (für einen Text, in dem a und b häufig auftreten)

a	b	c	d	e	f	g	␣
0	1	10	11	100	101	110	111

Die Wort „abba“ und „ag“ führen zu dem selben Code „0110“, also ist keine eindeutige Dekodierbarkeit gegeben.

Definition 5.0.3 (eindeutig dekodierbar)
Ein Code heißt eindeutig dekodierbar, falls verschiedene Originaldateien zu verschiedenen kodierten Dateien führen (injektive Kodierung).

eindeutig dekodierbar

Blockcodes sind **eindeutig decodierbar**.

a	b	c
1	10	00

Beispiel 5.0.4 (Eindeutige Codierbarkeit)
ist eindeutig dekodierbar. Das liegt daran, dass kein Codewort als Präfix eines anderen auftaucht. ◇

Definition 5.0.5 (Präfixcode)
Ein Code heißt Präfixcode, wenn kein Codewort als Präfix eines anderen Codeworts auftaucht.

Präfixcode

Beispiel 5.0.6 (Präfixcode)
Der folgende Code ist Präfixcode

a	b	c	d	e
00	01	100	11	101

,

jedoch ist

a	b	c	d	e	f	g	␣
0	1	10	11	100	101	110	111

keiner. ◇

Blockcode ist Präfixcode, da alle Codewörter gleich lang und verschieden sind.

Präfixcode ist eindeutig dekodierbar und lässt sich eindeutig mit einem Binärbaum identifizieren: das Codewort eines Zeichens entspricht Weg von der Wurzel zum Blatt (0 = links, 1 = rechts).

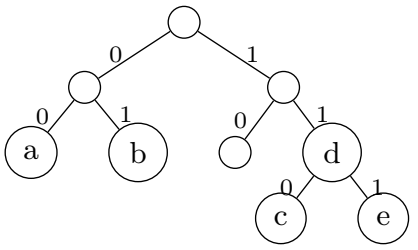


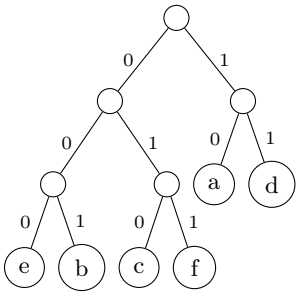
Abbildung 15: Der zu dem Präfixcode aus Beispiel 5.0.6 korrespondierende Binärbaum.

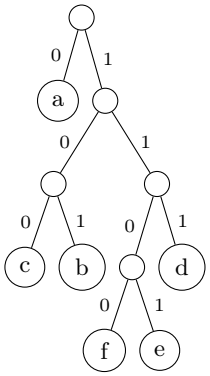
Sei `c.key` die Häufigkeit eines Zeichens $c \in C$ in der zu kodierenden Datei.

Definition 5.0.7 ($B(T)$, optimaler Präfixcode)
Für einen Präfixcode (mit zugehörigem Binärbaum) T ist die Größe (= Anzahl Bits) der kodierten Datei $B(T) = \sum_{c \in C} \texttt{c.key} \cdot d_T(c)$, wobei $d_T(c)$ die Tiefes des Blatts c im Baum T (= Länge des Codeworts) ist. Ein Präfixcode T mit $B(T) \leqslant B(T')$ für alle Präfixcodes T' heißt **optimal**.

Beispiel 5.0.8
Betrachte die folgende Häufigkeitsanalyse eine Dokuments und zwei mögliche Kodierungen.

Zeichen	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	d
Code T_1	10	001	010	11	000	011
Code T_2	0	101	100	111	1101	1100





Daran, dass T_1 und T_2 sich als Binärbäume darstellen lassen, deren Blätter die Codewörter sind, garantiert, dass es sich um Präfixcode handelt. Es gilt

$$B(T_1) = 3(9 + 13 + 12 + 5) + 2(45 + 16) = 239 \quad \text{und} \quad B(T_2) = 224. \diamond$$

Im Folgenden sei Q eine Prioritätswarteschlange, implementiert als Min-Heap, welcher zunächst die Zeichen enthält, deren Schlüssel ihre Häufigkeiten darstellen.

Algorithmus 7 : `Huffman`-Algorithmus

Data : Zeichensatz C mit Häufigkeiten `c.key`, $c \in C$
Result : Optimaler Präfixcode T
for $c \in C$ **do** `Insert`(Q, c)
while Q enthält mehr als einen Baum **do**
  $T_1 \leftarrow$ `ExtractMin`(Q)
  $T_2 \leftarrow$ `ExtractMin`(Q)
  sei T Baum mit linkem Teilbaum T_1 und rechtem Teilbaum T_2
  T`.key` $= T_1$`.key` $+ T_2$`.key`
  `Insert`(Q, T)
return `ExtractMin`(Q)

Anwenden des Algorithmus zeigt, dass der Code T_2 optimal ist, sofern der `Huffman`-Algorithmus korrekt ist.

Zeichen	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	5

Abbildung 16: HUFFMANN-Algorithmus

18.05.2020

Preprocessing $\mathcal{O}$(Dateigröße): Bestimme Zeichensatz C und Häufigkeiten `c.key` für $c \in C$ durch einmaliges Lesen der zu komprimierenden Textdatei.

Der HUFFMAN-Algorithmus benötigt $2n-1$ `Insert`-Operationen (benötigen jeweils $\mathcal{O}(\log(n))$ im Heap), n für die Initialisierung und jeweils einmal in den $n-1$ Durchläufen der While-Schleife. Ferner benötigt der Algorithmus $2(n-1)+1 = 2n-1$ `ExtractMin`-Operationen (jeweils $\mathcal{O}(\log(n))$ und $n-1$ mal die Operation „$T \leftarrow T_1 + T_2$“, welche konstanten Aufwand benötigt.

THEOREM 5.0.1: HUFFMAN-ALGORITHMUS

Der HUFFMAN-Algorithmus konstruiert einen optimalen Präfixcode. Seine Laufzeit beträgt $\mathcal{O}(n \log(n))$.

Zum Beweis der Korrektheit benötigen wir zwei Lemmata.

Lemma 5.0.9

Seien $x, y \in C$ verschieden mit $\texttt{x.key} \leqslant \texttt{y.key} \leqslant \texttt{c.key}$ für alle $c \in C \backslash \{x\}$. Dann gibt es einen optimalen Präfixcode T, in dem x und y Geschwister sind.

Beweis. Sei T ein optimaler Präfixcode, welcher existiert, da es nur endlich viele Binärbäume gibt, welche die Kodierungsmöglichkeiten darstellen. (also nicht so wirklich, aber egal???)

Seien $a, b \in C$, sodass $d_T(a) = d_T(b) \geqslant d_T(c)$ für alle $c \in C \backslash \{a, b\}$.

Wir konstruieren einen Präfixcode T' aus T, in welchem a und x ausge-

tauscht werden. Dann gilt

$$\begin{aligned}
B(T) - B(T') &= \sum_{c\in C} \texttt{c.key} d_T(c) - \sum_{c\in C} \texttt{c.key} d_{T'}(c) \\
&= \texttt{a.key} d_T(a) + \texttt{x.key} d_T(x) - \texttt{a.key} d_{T'}(a) \\
&\qquad - \texttt{x.key} d_{T'}(x) \\
&= \texttt{a.key}(d_T(a) - d_{T'}(a))\texttt{x.key}(d_T(x) - d_{T'}(x)) \\
&= \texttt{a.key}(d_T(a) - d_T(x))\texttt{x.key}(d_T(x) - d_T(a)) \\
&= \underbrace{(\texttt{a.key} - \texttt{x.key})}_{\geqslant 0} \underbrace{(d_T(a) - d_T(x))}_{\geqslant 0} \geqslant 0
\end{aligned}$$

und somit $B(T') \leqslant B(T)$, also ist T' optimal. Das geht nun analog für das Vertauschen von y und b. □

Lemma 5.0.10
Seien $x, y \in C$ wie oben und T ein Präfixcode, in dem x und y Geschwister mit gemeinsamen Mutterknoten z sind. Sei

$$C' \coloneqq (C\backslash\{x,y\}) \cup \{z\} \qquad \textit{und} \qquad \texttt{z.key} = \texttt{x.key} + \texttt{y.key}.$$

Weiter entstehe der Baum T' aus T durch Löschen der Blätter x und y. Der Präfixcode T ist genau dann optimal für C, wenn T' optimal für C' ist.

Beweis. Es ist T ein Präfixcode für C und T' ein Präfixcode für C'. Es gilt

$$\begin{aligned}
B(T) &= \sum_{c\in C} \texttt{c.key} \cdot d_T(c) \\
&= \sum_{c\in C\backslash\{x,y\}} \texttt{c.key} \cdot d_T(c) + \texttt{x.key} \cdot d_T(x) + \texttt{y.key} \cdot d_T(y)
\end{aligned}$$

und

$$B(T') = \sum_{c\in C\backslash\{x,y\}} \texttt{c.key} \cdot d_{T'}(c) + \underbrace{\texttt{z.key}}_{\texttt{x.key}+\texttt{y.key}} \cdot \underbrace{d_{T'}(z)}_{=d_T(x)-1} .$$

Somit gilt

$$\begin{aligned}
B(T) - B(T') &= \cancel{\sum_{c\in C\backslash\{x,y\}} \texttt{c.key} \cdot d_T(c)} + \texttt{x.key} \cdot d_T(x) + \texttt{y.key} \cdot d_T(y) \\
&\quad - \cancel{\sum_{c\in C\backslash\{x,y\}} \texttt{c.key} \cdot d_{T'}(c)} - (\texttt{x.key} + \texttt{y.key}) \cdot (d_T(x) - 1) \\
&= \texttt{x.key} \cdot d_T(x) + \texttt{y.key} \cdot d_T(y) \\
&\qquad - \texttt{x.key}(d_T(x) - 1) - \texttt{y.key}(d_T(y) - 1) \\
&= \texttt{x.key} - \texttt{y.key}.
\end{aligned}$$

Angenommen, es existiert ein Präfixcode T'' für C mit $B(T'') < B(T)$. Nach Lemma 5.0.9 können wir annehmen, dass x und y Blätter größter Tiefe in T'' sind (da sie die niedrigsten Häufigkeiten haben). Konstruiere nun ein Baum T''' für C', indem man das Paar (x, y) durch die Mutter z ersetzt. Aus $B(T) - B(T') = \texttt{x.key} - \texttt{y.key}$ folgt $B(T'') < B(T')$. Ist der eine Präfixcode nicht optimal, ist es der andere auch. Aufgrund von Symmetrie kann man das Argument umdrehen und das beweist die Äquivalenz. □

Bemerkung 5.0.11 HUFFMAN-Codes sind optimale Präfixcodes und sogar optimal im Vergleich zu alle Kodierungsverfahren, die die Eingabedaten Zeichen für Zeichen kodieren.

Gibt es stochastische Abhängigkeiten (z.B kommt im Deutschen nach einem e oft ein n) zwischen aufeinanderfolgenden Zeichen, können durch Kombination und gemeinsame Kodierung solcher Zeichen eventuell bessere Ergebnisse erzielt werden.

Beispiel 5.0.12

Betrachte die Eingabedatei

```
abc abc abc def def abc def def def abc abc def ...
```

Hier ist es besser, die Zeichenketten `abc` und `def` zu kodieren (z.B. mit 0 und 1) anstatt die einzelnen Buchstaben zu kodieren. ◇

Bemerkung 5.0.13 (Dynamische HUFFMAN-Codes)

Es gibt auch dynamische HUFFMAN-Codes, die beim Lesen der zu kodieren Datei erzeugt und dynamisch verändert werden.

6 Binäre Suchbäume

Ein binärer Suchbaum T unterstützt Operationen für dynamische Menge:

- `Tree-Search`(x, k): gibt einen Zeiger y auf das Element aus Teilbaum unterhalb von x mit `y.key` $= k$ zurück oder `None`, falls es kein solches Element unterhalb von x gibt.
- `Tree-Insert`(z): fügt das durch z referenzierte Element zu T hinzu.
- `Tree-Delete`(z): löscht das durch z referenzierte Element aus T.

Die folgenden Operationen sind neu und anders als bevor, das sie voraussetzen, dass es eine Ordnung auf den Schlüsseln gibt.

- `Tree-Minimum`(x): lieft eine Zeiger y auf ein Element im Teilbaum unterhalb von x mit `y.key` minimal.
- `Tree-Maximum`(x): analog zu `Tree-Minimum`(x).
- `Tree-Successor`(x): liefert einen Zeiger auf ein Element mit nächstgrößerem `key`.
- `Tree-Predecesor`(x): analog zu `Tree-Successor`(x).

Wir wünschen uns, dass die Laufzeit der obigen Operationen in $\mathcal{O}(h)$ sind, wobei h die Höhe des Baumes ist.

6.1 Binäre Suchbäume

Definition 6.1.1 (Binärer Suchbaum)
Ein binärer Suchbaum T ist ein binärer Baum, dessen Knoten Objekte mit den folgenden Attributen sind: `x.p` (Elternknoten), `x.key`, `x.data`, `x.left`, `x.right` (linkes bzw. rechtes Kind). Darüber hinaus muss die folgende Suchbaum-Eigenschaft erfüllt sein: ist x Knoten in T und y Knoten in dem linken (rechten) Teilbaum von x, so ist `y.key` $\leqslant$ ($\geqslant$) `x.key`.

Definition 6.1.2 (Inorder-Reihenfolge)
Die eindeutige Sortierung der Knoten eines Binärbaums, bei der jeder Knoten nach den Knoten seines linken Teilbaums und vor den Knoten seines rechten Teilbaums kommt, heißt inorder-Reihenfolge.

Bemerkung 6.1.3 Die Suchbaum-Eigenschaft ist genau dann erfüllt, wenn die inorder-Reihenfolge eine aufsteigende Sortierung der Knotenschlüssel liefert.

Algorithmus 8 : `Inorder-Tree-Walk`

Data : Zeiger x auf Baumknoten
Result : Sortierte Liste der Schlüssel des Teilbaums unter x
if $x \neq$ `None` **then**
    `Inorder-Tree-Walk`$(x$`.left`$)$
    print$(x$`.key`$)$
    `Inorder-Tree-Walk`$(x$`.right`$)$

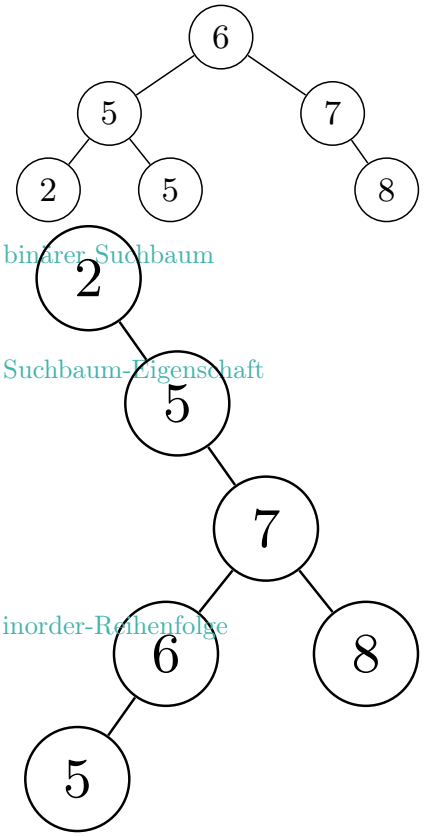
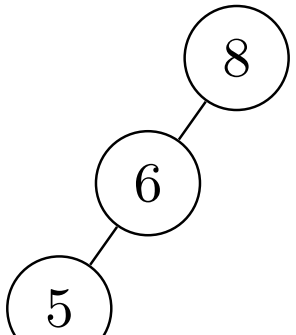


Abbildung 17: Zwei binäre Suchbäume zur selben Schlüsselmenge.



Lemma 6.1.4
Die Laufzeit von `Inorder-Tree-Walk`(x) *ist linear in der Anzahl der Knoten des Teilbaums unter* x*.*

Beweis. (Idee) `Inorder-Tree-Walk`(y) wird für jeden Knoten y des Teilbaums unter x genau einmal aufgerufen. □

Algorithmus 9 : `Tree-Search` (rekursiv)

Data : Zeiger x auf Baumknoten mit Schlüsselwert k
Result : Zeigen y auf einen Knoten unterhalb von x mit y`.key` $= k$ oder `None`, wenn kein solcher existiert.

if $x =$ `None` oder $k = x$`.key` **then return** x
if $k <$ x`.key` **then return** `Tree-Search`$(x$`.left`$, k)$
else return `Tree-Search`$(x$`.right`$, k)$

Algorithmus 10 : `Tree-Search` (iterativ)

Data : Zeiger x auf Baumknoten mit Schlüsselwert k
Result : Zeigen y auf einen Knoten unterhalb von x mit y`.key` $= k$ oder `None`, wenn kein solcher existiert.

while $x \neq$ `None` und $k \neq x$`.key` **do**
  if $k <$ x`.key` **then** $x \leftarrow x$`.left`
  else $x \leftarrow x$`.right`
return x

Lemma 6.1.5 (Laufzeit von `Tree-Search`)
Die Laufzeit beider `Tree-Search`*-Varianten ist* $\mathcal{O}(h)$*, wobei* h *die Höhe des Teilbaums unter* x *ist.*

Beweis. (Idee) Die besuchten Knoten liegen auf dem Weg von x zu dem Blatt. □

Ist der Baum nicht leer, so kann man die folgenden Methoden verwenden.

Algorithmus 11 : `Tree-Minimum`(x) (iterativ)

Data : Zeiger x auf einen Baumknoten
Result : Zeiger y auf eine Knoten unterhalb von x mit y`.key` minimal
while x`.left` $\neq$ `None` **do** $x \leftarrow x$`.left` ;
return x

`Tree-Maximum`(x) funktioniert analog, es wird lediglich x`.left` durch x`.right` ersetzt.

Lemma 6.1.6 (Laufzeiten von `Tree-Minimum` und `Tree-Maximum`)
Die Laufzeiten von `Tree-Minimum` *und* `Tree-Maximum` *sind in* $\mathcal{O}(h)$*, wobei* h *die Höhe des Teilbaumes unter* x *ist.*

Beweis. (Idee) Die besuchten Knoten liegen auf dem Weg von x zu dem Blatt. □

Algorithmus 12 : `Tree-Successor`(x)

Data : Zeiger x auf einen Baumknoten
Result : Zeiger y auf eine Knoten nach x in der inorder-Reihenfolge

if x`.right` $\neq$ `None` **then return** `Tree-Minimum`$(x$`.right`$)$
$y \leftarrow x$`.p`
while $y \neq$ `None` und $x = y$`.right` **do**
  $x \leftarrow y$, $y \leftarrow y$`.p`
return y

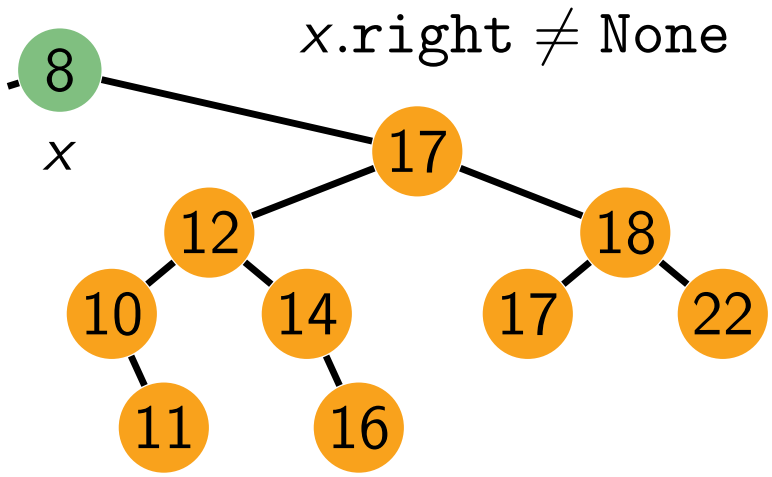


Abbildung 19: `Tree-Successor`(x)

In `Tree-Predecessor`(x) ersetzt alle x`.right` durch x`.left` und `Tree-Minimum` durch `Tree-Maximum`.

Lemma 6.1.7 (Laufzeiten von `Tree-Successor` und `Tree-Predecessor`)
Die Laufzeiten von `Tree-Successor` und `Tree-Predecessor` sind $\mathcal{O}(h)$, wobei h die Höhe des Baums ist.

Algorithmus 13 : `Tree-Insert`(z)

Data : Zeiger z auf ein neues Objekt mit z.`left` $=$ z.`right` $=$ `None`
Result : Modifikation von T, sodass z ein neuer Baumknoten ist.

$y \leftarrow$ `None`, $x \leftarrow T$.`root`
while $x \neq$`None` **do**
    $y \leftarrow x$
    if z.`key` $<$ x.`key` **then** $x \leftarrow x$.`left`
    else $x \leftarrow x$.`right`
z.`p` $\leftarrow y$
if $y =$`None` **then** T.`root` $\leftarrow z$
else if z.`key` $<$ y.`key` **then** y.`left` $\leftarrow z$
else y.`right` $\leftarrow z$

Lemma 6.1.8
Die Laufzeit von `Tree-Insert` *ist in* $\mathcal{O}(h)$*, wobei* h *die Höhe des Baumes* T *ist.*

Beweis. (Idee) Die besuchten Knoten liegen auf dem Weg von der Wurzel zu dem Blatt. □

19.05.2020

Löschen eines Knotens in einem binären Suchbaum

Sei T ein binärer Suchbaum und z ein Knoten in T. Das Löschen von z aus T ist die komplizierteste Operation. Wir unterscheiden deshalb in drei Fälle.

① Hat der Knoten z in T keine Kinder, so kann z einfach gelöscht werden, da die Suchbaumeigenschaft von T erhalten bleibt.

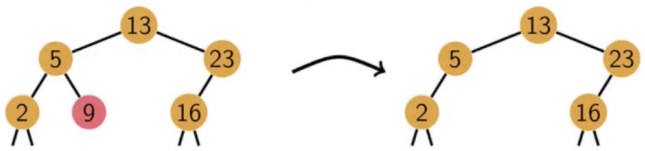


Abbildung 20: Fall 1: Löschen eines Blatts erhält die Suchbaumeigenschaft des Binärbaums.

② Hat der Knoten z in T genau ein Kind y, so nimmt y nach dem Löschen von z dessen Position ein.

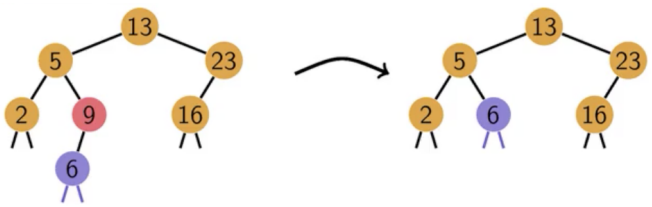


Abbildung 21: Fall 2.

③ Hat der Knoten z in T zwei Kinder, so können wir nicht wie in Fall

(2) den Kindknoten „weiter hoch reichen“, da es zwei Kindknoten gibt. Deswegen ermitteln wir $y =$`Successor`(z).

(a) Ist y rechtes Kind von z, so ersetzen wir z durch y. Der rechte Teilbaum von y bleibt dabei unverändert und der linke Teilbaum von z wir zum linken Teilbaum von y.

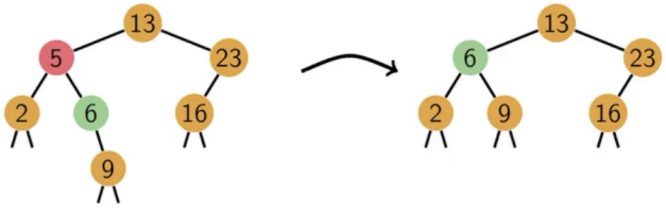


Abbildung 22: Fall 3 a.

(b) Ist y nicht das rechte Kind von z, so ersetzen wir zunächst y durch sein rechtes Kind x. Dann ersetzen wir z durch y, wobei y den linken und rechten Teilbaum von z übernimmt.

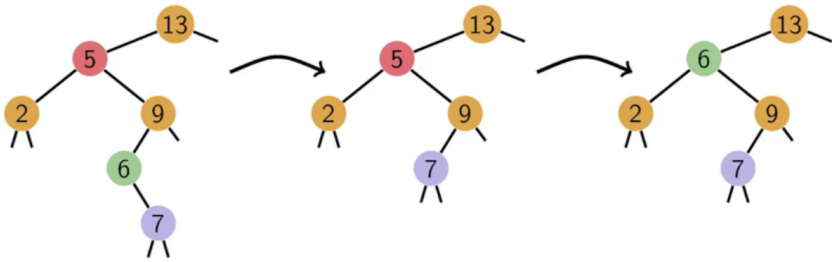


Abbildung 23: Fall 3b.

In (a) und (b) nutzen wir aus, dass 6 kein linkes Kind haben kann, da es nach der Nachfolger von z ist.

Algorithmus 14 : `Tree-Delete`(z)

```
if z.left = None then Transplant(z, z.right)
else if z.right = None then Transplant(z, z.left)
else
    y ← Tree-Minimum(z.right)
    if y.p ≠ z then
        Transplant(y, y.right)
        y.right ← z.right
        y.right.p ← y
    Transplant(z, y)
    y.left ← z.left
    y.left.p ← y
```

Algorithmus 15 : `Transplant`(u, v)

Data : Knoten u in T, Knoten v oder $v =$`None`.
Result : Ersetze den Teilbaum unter u durch den Teilbaum unter v.

```
if u.p = None then T.root ← v ;
else if u = u.p.left then u.p.left ← v ;
else u..p.right ← v;
if v ≠ None then v.p ← u.p ;
```

Lemma 6.1.9
Die Laufzeit von `Transplant`(u, v) *ist* $\mathcal{O}(1)$ *und die Laufzeit von* `Tree-Delete`(z) *ist* $\mathcal{O}(h)$*, wobei* h *die Höhe des rechten Teilbaums von* z *ist.*

Beweis. `Transplant`(u, v) setzt konstant viele Zeiger neu gesetzt. Bei `Tree-Delete`(z) werden konstant viele Zeiger neu gesetzt und `Tree-Minimum`$(z.$`right`$)$ angewandt. □

Summa summarum gilt: auf einem binären Suchbaum der Höhe haben die Operationen `Tree-Minimum`, `Tree-Maximum`, `Tree-Successor`, `Tree-Predecessor`, `Tree-Insert` und `Tree-Delete` die Laufzeit $\mathcal{O}(h)$.

Wir wissen jedoch nichts über h, z.B. kann der Baum völlig degenerieren: die Zahlen $1, \ldots, n$ können nacheinander in eine leeren Baum eingefügt werden, sodass das linke Kind von 1 die 2 ist und das linke von 2 die 3 .... Der binäre Suchbaum hat dann Höhe $n - 1$ und ist zu einer Liste degeneriert. Dies wollen wir verhindern.

Deshalb suchen wir nach balancierten Bäumen, bei denen, anders als in dem vorherigen Beispiel, nicht das „Gewicht“ des Baumes auf einer Seite liegt.

Diese Klasse der balancierten Bäume soll die folgenden Eigenschaften erfüllen:

1. Für jedes $n \in \mathbb{N}$ existiert in der Klasse ein Baum T mit n Knoten und Höhe $h(T) \in \mathcal{O}(\log(n))$.
2. Die Klasse ist abgeschlossen unter den Operationen `Insert` und `Delete`.
3. Alle Operationen kosten auf dieser Klasse $\mathcal{O}(h(T))$ Zeit.

Bemerkung 6.1.10 Um (2) zu gewährleisten, muss der Baum nach Einfügen oder Löschen jeweils modifiziert werden. Diese Modifikationen nennen wir Rotationen. Rotationen

Die Laufzeit von Einfügen und Löschen darf durch die Rotationen aber nicht zu sehr anwachsen und die anderen Operationen (`Tree-Search`, `Tree-Minimum`, `Tree-Maximum`, `Tree-Successor` und `Tree-Predecessor`) sollen in $\mathcal{O}(h(T))$ bleiben.

Algorithmus 16 : `Rotate-Right`(x)

Data : Knoten x in T mit x`.left` $\neq$ `None`

Result : x`.left` nimmt dem Platz von x ein; x wird das rechte Kind von x`.left`.

```
y ← x.left
if x.p = None then T.root = y
else if x.p.left == x then
    x.p.left = y
else x.p.right = y
y.p = x.p;   x.left = y.right
if x.left then x.left.p = x
y.right = x;   x.p = y
```

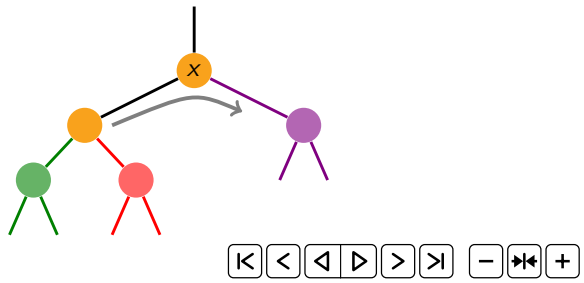


Abbildung 24: Rechtsrotation

Algorithmus 17 : `Rotate-left`(x)

```
y = x.right
if x.p == None then   T.root = y
else if x.p.left = x then
    x.p.left ← y
else x.p.right = y
y.p = x.p;   x.right = y.left
if x.right then   x.right.p = x
y.left = x;   x.p = y
```

Bemerkung 6.1.11 `Rotate-Right` und `Rotate-Left` erhalten die Suchbaumeigenschaft und haben Laufzeit $\mathcal{O}(1)$, da nur konstant viele Zeige neu gesetzt werden müssen.

`Rotate-Right` und `Rotate-Left` sind zueinander inverse, d.h.

- Ist $y = x$`.left`, so bleibt T bei Hintereinanderausführung von `Rotate-Right`(x) und `Rotate-Left`(y) unverändert.
- Ist $y = x$`.right`, so bleibt T bei Hintereinanderausführung von

`Rotate-Left`(x) und `Rotate-Right`(y) unverändert.

`Rotate-Right` macht den rechten Teilbaum höher und den linken weniger hoch.

`Rotate-Left` macht den linken Teilbaum höher und den rechten weniger hoch.

Die beiden Rotationen `Rotate-Right` und `Rotate-Left` sind mächtig genug, einen Suchbaum in jede gewünschte Gestalt zu bringen.

Theorem 6.1.1: Mächtigkeit der Rotationen

Sind T und T' Suchbäume mit denselben Schlüsselwerten, so gibt es eine endliche Folge von Rotationen, die T in T' überführt.

Beweis. Die Idee dieses Beweises ist, T durch Rotationen in eine „Standardform“ zu transformieren. Der „Standardbaum“ für eine Baum mit n Knoten soll der zur Liste rechts-entartete Baum der Höhe n sein. Die Transformation kann so in Pseudocode geschrieben werden:

```
while Es existiert ein Knoten x mit x.left ≠ None do
    T.Rotate-right(x)
```

Da `Rotate-right`(x) die Höhe des linken Teilbaums von x um eins reduziert und die des rechten Teilbaums von x um eins erhöht, wird nach endlich vielen Durchläufen der `while`-Schleife der Baum in Standardform sein: der Baum besitzt nur noch Knoten mit einem rechten oder keinen Kindern.

Nach dem obigen Beweis kann auch T' mit endlich vielen Rotationen in Standardform gebracht werden. Also können wir mit endlich vielen Rotationen T in T' transformieren, indem wir erst T in Standardform bringen und dann die Rotationen, die T' in Standardform bringen, umkehren (also anstatt der Rechtsrotationen Linksrotationen in umgekehrter Reihenfolge anwenden). □

Bemerkung 6.1.12 Man kann also durch Einfügen / Löschen entstandene Ungleichgewichte in binären Suchbäumen durch Rotationen ausgleichen.

Man kann sogar zeigen, dass sogar $\mathcal{O}(n)$ Rotationen genügen, um T in T' zu überführen. (Man kann den oben beschriebenen Algorithmus rekursiv „von unten aus“ anwenden, um immer linke Teilbäume der Höhe eins zu eliminieren.)

6.2 AVL-Bäume

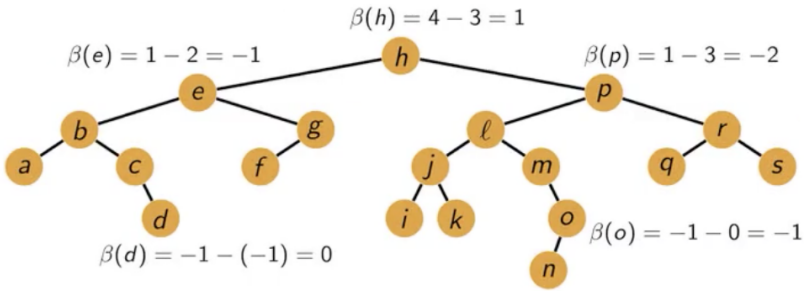
Definition 6.2.1

Sei T ein Binärbaum, v ein Knoten von T und T_r, T_ℓ die rechten bzw. linken Teilbäume unterhalb von v. Die **Balance** des Knotens v ist

$$\beta(v) := h(T_r) - h(T_\ell),$$

wobei wir $h(\varnothing) := -1$ setzen

Beispiel 6.2.2
Betrachte den folgenden Baum mit den zugehörigen Balancen.



25.05.2020

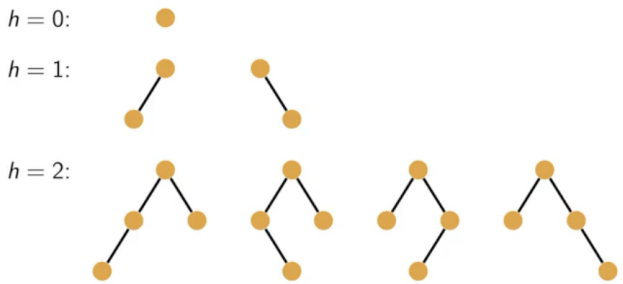
Definition 6.2.3 (AVL-Baum (Adelson-Velski, Landis))
Ein binärer Suchbaum T heißt **AVL-Baum**, falls $|\beta(v)| \leqslant 1$ („Knoten v ist balanciert“) für jeden Knoten $v \in T$.

In jedem Knoten v eines AVL-Baums ist die Balance $\beta(v)$ gespeichert

Im Weiteren werden wir zeigen, dass ein AVL-Baum mit n Knoten eine Höhe $h(T) \in \mathcal{O}(\log(n))$ besitzt, indem wir extremale AVL-Bäume betrachten, die zu einer gegebenen Höhe h die minimale Knotenanzahl $n(h)$ besitzen und dann zeigen, dass $n(h) \geqslant 2^{\Omega(h)}$ gilt.

Ein Problem ist, dass durch Einfügen oder Löschen eines Knotens unbalancierte Knoten entstehen. Wir werden jedoch sehen, dass dies mit Hilfe von Rotationen effizient in Laufzeit $\mathcal{O}(h(T))$ behoben werden kann.

Definition 6.2.4
Ein AVL-Baum der Höhe h heißt **extremal**, wenn er $\min(\{n : \text{es gibt einen AVL-Baum der Höhe } h \text{ mit } n \text{ Knoten }\})$ Knoten enthält.



Beispiel 6.2.5 (Extremale AVL-Bäume)

Lemma 6.2.6 (Höhe extremaler AVL-Bäume I)
Für einen extremalen AVL-Baums der Höhe $h \geqslant 2$ gilt

1. *Die Balance jedes inneren Knotens ist 1 oder -1.*
2. *Der linke und der rechte Teilbaum der Wurzel sind extremale AVL-Bäume zu den Höhe $h-1$ und $h-2$ (oder umgekehrt).*

Beweis. (1) Ist $h \geqslant 2$, so ist der linke Teilbaum von w und der rechte Teilbaum von w nicht leer, wobei w die Wurzel ist, da sonst $|\beta(w)| > 2$ wäre.

(2) Offensichtlich sind diese beiden Teilbäume AVL-Bäume.

(3) Angenommen, $\beta(w) = 0$, dann gilt $h(\mathrm{LTB}(w)) = h(\mathrm{RTB}(w)) = h - 1$. Dann kann der linke Teilbaum von w durch den tieferen seiner Teilbäume ersetzt werden, ohne das $|\beta(w)| \leqslant 1$ verletzt wird. Dies stellt einen Widerspruch zur Minimalität der Knotenanzahl dar.

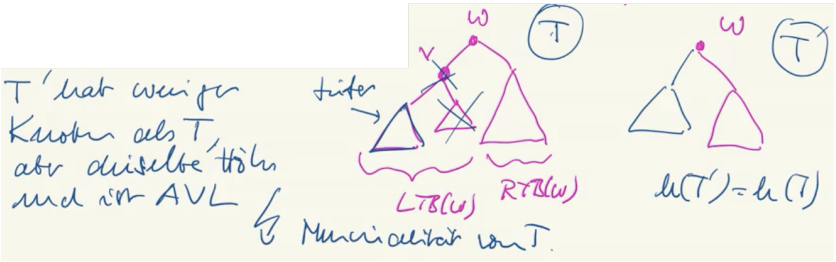


Abbildung 25: Illustration des obigen Beweisschrittes.

(4) Somit ist $h(\mathrm{LTB}(w)) \neq h(\mathrm{RTB}(w))$, d.h. $|h(\mathrm{LTB}(w)) - h(\mathrm{RTB}(w))| = 1$. Also sind die beiden Höhe $h - 1$ und $h - 2$.

(5) Angenommen, $\mathrm{LTB}(w)$ ist nicht knotenminimal zur Höhe $h' \in \{h-1, h-2\}$. Dann kann man $\mathrm{LTB}(w)$ durch einen knotenminimalen AVL-Baum der Höhe h' ersetzen und erhält insgesamt einen AVL-Baum, der Höhe h mit weniger Knoten. Dies ist ein Widerspruch dazu, dass T extremal ist. □

Korollar 6.2.7
Für $h \geqslant$ sei $n(h)$ die Anzahl der Knoten eines extremalen AVL-Baums der Höhe h. Dann ist $n(0) = 1$, $n(1) = 2$ und $n(h) = 1+n(h-1)+n(h-2)$ für alle $h \geqslant 2$.

Bemerkung 6.2.8 (Fibonacci-Bäume) Wegen der Ähnlichkeit der Rekursionsformel zur rekursiven Definition der Fibonacci-Zahlen spricht man auch von Fibonacci-Bäumen.

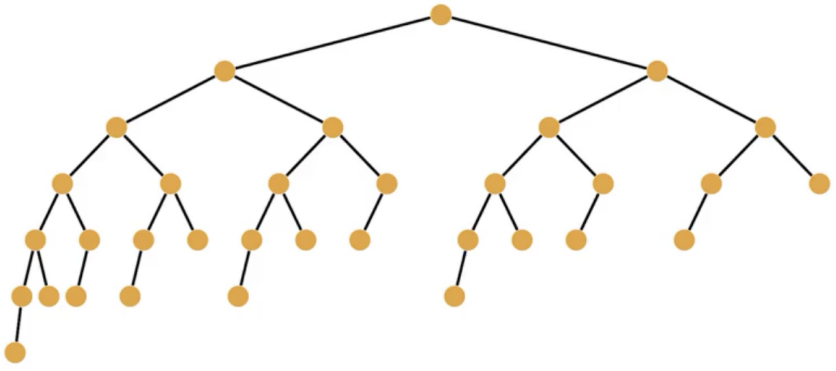


Abbildung 26: Ein [Fibonacci-Baum der Höhe sechs mit 33 Knoten.

Beispiel 6.2.9 (Fibonacci-Bäume)

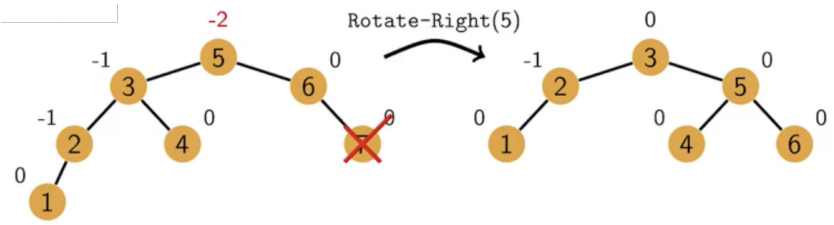


Abbildung 27: Anwenden von `Rotate-Right`(5) sorgt dafür, dass wieder ein AVL-Baum entsteht.

Lemma 6.2.10 (Höhe extremaler AVL-Bäume II)
Die minimale Knotenanzahl $n(h)$ eines AVL-Baums der Höhe h erfüllt

$$n(h) \geqslant 2^{\frac{h}{2}}.$$

Beweis. Wir verwenden Induktion über h. Für $h = 0$ gilt $n(h) = 1 = 2^{\frac{0}{2}}$, für $h = 1$ gilt $h(1) = 2 > 2^{\frac{1}{2}}$ und für $h = 2$ gilt $n(h) = 4 > 2^{\frac{2}{2}}$. Für $h > 2$ gilt

$$n(h) = 1 + n(h-1) + n(h-2) = 2 + 2n(h-2) + n(h-3) > 2n(h-2).$$

Wiederholtes Anwendens dies Arguments liefert

$$n(h) > 4n(h-4) > \ldots > \begin{cases} 2^{\frac{h}{2}} n(0), & \text{wenn } h \text{ gerade ist,} \\ 2^{\frac{h-1}{2}} n(1) & \text{sonst.} \end{cases}$$

Es gilt $2^{\frac{h-1}{2}} n(1) = \sqrt{2} \cdot 2^{\frac{h}{2}} > 2^{\frac{h}{2}}$ und $2^{\frac{h}{2}} n(0) = 2^{\frac{h}{2}}$. □

Theorem 6.2.1: Höhe eines AVL-Baums

Ein AVL-Baum T mit n Knoten hat eine Höhe $h(T) \leqslant 2\log_2(n)$.

Beweis. Es gilt

$$2^{\frac{h}{2}} \leqslant n(h(T)) \leqslant n$$

und somit $h \leqslant 2\log_2(n)$ □

Bemerkung 6.2.11 Für AVL-Bäume gilt sogar

$$h(T) \leqslant \frac{1}{\log(\varphi)} \log(n) + \mathcal{O}(1),$$

wobei $\varphi := \frac{1+\sqrt{5}}{2}$ und $\frac{1}{\log(\varphi)} \approx 1.44$ (betrachte Zusammenhang mit Fibonacci-Folge).

Wiederherstellen der Balance nach dem Löschen

In dem unten stehen Baum wird die 7 gelöscht. In dem unten stehen Baum wird auch die 7 gelöscht, aber Rotate-right stellt nicht wieder die AVL-Baum-Eigenschaft her: Man bemerke, dass der erste Ausgangsbaum aus dem zweiten durch `Rotate-Left`(2) erzeugt werden kann. Dies motiviert Doppelrotationen, d.h. zunächst die Rotation eines Teilbaums

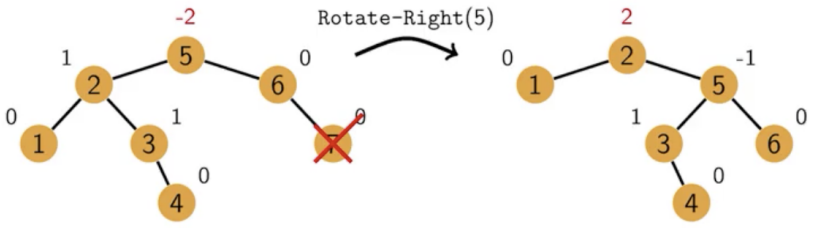
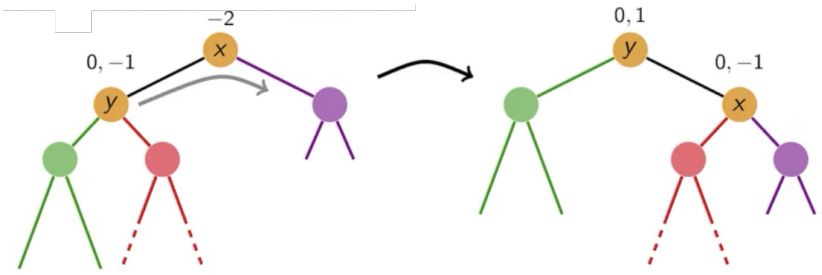


Abbildung 28: Anwenden von `Rotate-Right`(5) sorgt nicht dafür, dass wieder ein AVL-Baum entsteht.



des problematischen Knotens und dann die Rotation des eigentlichen Knotens.

Das allgemeine Vorgehen sieht so aus: Sei T ein AVL-Baum.

Fügt man neuen Knoten als Blatt v zu T hinzu, so ändern sich Balancen nur entlang des Weges von v zu $T.\texttt{root}$.. Löscht man einen Knoten v aus T, so ändern sich die Balancen nur entlang Weg von v bzw. $\texttt{Successor}(v)$ (falls v zwei Kinder hat) zu $T.\texttt{root}$. Die Balancen ändern sich in beiden Fällen höchstens um eins.

Man kontrolliert also die Knoten x entlang des Weges zu $T.\texttt{root}$ und repariert dann wenn $|\beta(x)| = 2$. Dabei achte man darauf, dass die Balancen der Knoten entlang des weiteren Weges betragsmäßig nie größer als zwei werden.

Wir betrachten im Folgenden zunächst den Fall, dass die Wurzel unbalanciert ist. Diese Ansatz können wir später auf Teilbäume anwenden.

Lemma 6.2.12 (Analyse der Rechtsrotation)
Seien T ein Suchbaum mit Wurzel x, $\beta(x) = -2$, $y = x.\texttt{left}$ und $\beta(y) \in \{0, -1\}$. Sind $T_{x.\texttt{left}}$ und $T_{x.\texttt{right}}$ AVL-Bäume und entsteht T' aus T durch $\texttt{Rotate-Right}(x)$, so gilt für die Balancen β' in T':

$$\beta'(x) = \begin{cases} 0, & \text{wenn } \beta(y) = -1, \\ -1, & \text{wenn } \beta(y) = 0 \end{cases} \quad \text{und} \quad \beta'(y) = \begin{cases} 0, & \text{wenn } \beta(y) = -1, \\ 1, & \text{wenn } \beta(y) = 0 \end{cases}$$

und $\beta'(v) = \beta(v)$ für alle $v \notin \{x, y\}$. Ferner gilt

$$h(T') = \begin{cases} h(T), & \text{wenn } \beta(y) = 0, \\ h(T) - 1 & \text{wenn } \beta(y) = -1. \end{cases}$$

Die Berechnung von T' und das Updaten der Balancen benötigt Laufzeit $\mathcal{O}(1)$.

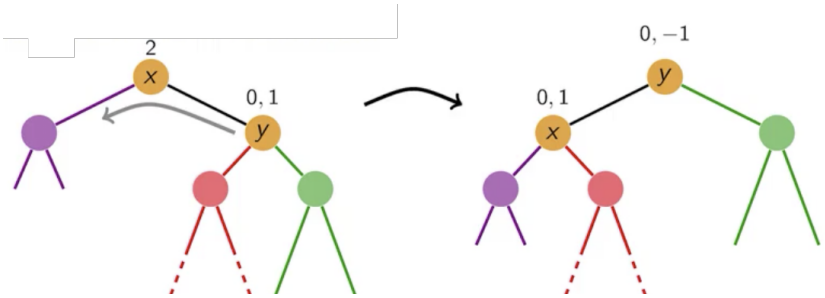
Lemma 6.2.13 (Analyse der Linksrotation)
Seien T ein Suchbaum mit Wurzel x, $\beta(x) = 2$, $y = x.\texttt{right}$ und $\beta(y) \in \{0,1\}$. Sind $T_{x.\texttt{left}}$ und $T_{x.\texttt{right}}$ AVL-Bäume und entsteht T' aus T durch `Rotate-Left`(x)*, so gilt für die Balancen β' in T':*

$$\beta'(x) = \begin{cases} 0, & \text{wenn } \beta(y) = 1, \\ 1, & \text{wenn } \beta(y) = 0 \end{cases} \quad \text{und} \quad \beta'(y) = \begin{cases} 0, & \text{wenn } \beta(y) = 1, \\ -1, & \text{wenn } \beta(y) = 0 \end{cases}$$

und $\beta'(v) = \beta(v)$ für alle $v \notin \{x, y\}$. Ferner gilt

$$h(T') = \begin{cases} h(T), & \text{wenn } \beta(y) = 0, \\ h(T) - 1 & \text{wenn } \beta(y) = 1. \end{cases}$$

Die Berechnung von T' und das Updaten der Balancen benötigt Laufzeit $\mathcal{O}(1)$.



Wir kommen nun zu den oben motivierten Doppelrotation.

Algorithmus 18 : `Rotate-LR`(x)

Data : Knoten x in T mit $x.\texttt{left} \neq \texttt{None}$ und $x.\texttt{left.right} \neq \texttt{None}$.
`Rotate-left`$(x.\texttt{left})$
`Rotate-right`(x)

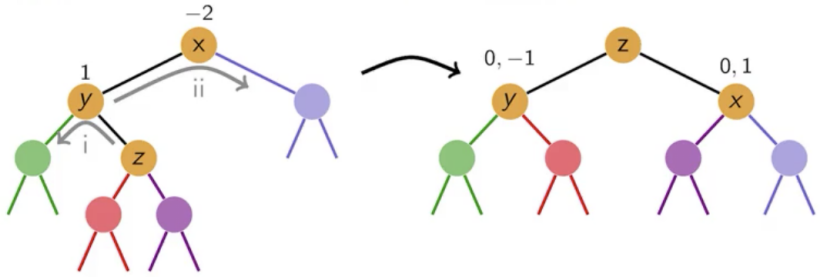


Abbildung 29: Doppelrotation Links-Rechts.

Lemma 6.2.14 (Analyse the Links-Rechts-Rotation)
Seien T ein Suchbaum mit Wurzel x, $y = x.\texttt{left}$, $z = y.\texttt{right}$, $\beta(x) = -2$ und $\beta(y) = 1$. Sind $T_{x.\texttt{left}}$ und $T_{x.\texttt{right}}$ AVL-Bäume und entsteht T' aus T durch `Rotate-LR`(x)*, so $h(T') = h(T) - 1$; für die Balancen β' in T' gilt*

$$\beta'(x) = \begin{cases} 0, & \text{wenn } \beta(z) \in \{0,1\}, \\ 1, & \text{wenn } \beta(z) = -1, \end{cases} \quad \text{und} \quad \beta'(y) = \begin{cases} 0, & \text{wenn } \beta(z) \in \{0,-1\}, \\ -1, & \text{wenn } \beta(z) = 1, \end{cases}$$

sowie $\beta'(z) = 0$ *und* $\beta'(v) = \beta(v)$ *für alle* $v \notin \{x, y, z\}$.

Algorithmus 19 : `Rotate-RL`(x)

`Rotate-right`$(x$`.right`$)$
`Rotate-right`(x)

26.05.2020

Korollar 6.2.15
Sei T *ein Suchbaum mit Balancen* β*, Wurzel* x*,* $|\beta(x)| = 2$*, sodass* $T_{x.\texttt{left}}$ *und* $T_{x.\texttt{right}}$ *AVL-Bäume sind. Mit einer (Doppel-)Rotation kann man* T *in einen AVL-Baum* T' *mit* $h(T') \in \{h(T), h(T) - 1\}$ *in* $\mathcal{O}(1)$ *Zeit überführen.*

Beweis. Wir unterscheiden in vier Fälle.

1. Fall: Ist $\beta(x) = -2$, $\beta(x$`.left`$) \in \{0, -1\}$, so wende `Rotate-Right`(x) an.
2. Fall: Ist $\beta(x) = -2$, $\beta(x$`.left`$) = 1$, so wende `Rotate-LR`(x) an.
3. Fall: Ist $\beta(x) = 2$, $\beta(x$`.left`$) \in \{0, 1\}$, so wende `Rotate-Left`(x) an.
4. Fall: Ist $\beta(x) = 2$, $\beta(x$`.left`$) = -1$, so wende `Rotate-RL`(x) an. □

Einfügen und Löschen in einem AVL-Baum

Beispiel 6.2.16 (Einfügen in einen AVL-Baum)
Betrachte den folgenden Suchbaum, in denen die 8 und die 39 eingefügt werden sollen.

Füge 8 ein:

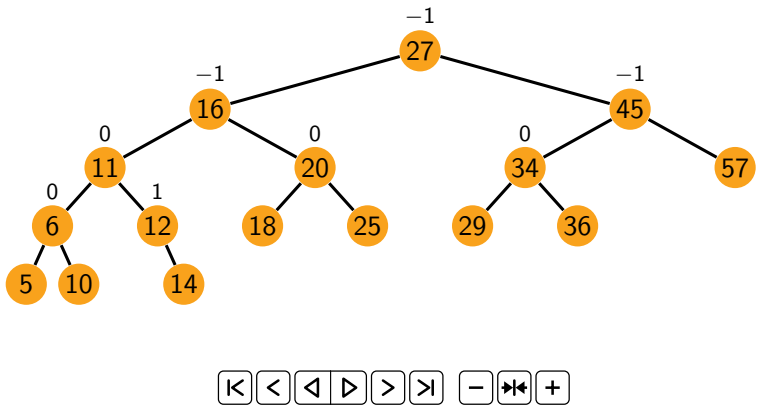


Abbildung 30: Einfügen in einen AVL-Baum

Theorem 6.2.2: Einfügen in einen AVL-Baum

Sei T ein AVL-Baum mit Balancen β. Mit `Tree-Insert`(x) werde ein neuer Knoten hinzugefügt und sei P der wegen zwischen T`.root` und x. Dann gilt für die Balancen β' des neuen Suchbaums

T':

$$\beta'(v) = \beta(v) \ \forall v \notin P \quad \text{und}$$
$$\beta'(v) \in \{\beta(v) - 1, \beta(v), \beta(v) + 1\} \ \forall v \in P \backslash \{x\}.$$

Gibt es ein $v \in P$ mit $|\beta'(v)| = 2$, so betrachte den von x aus gesehen ersten solchen Knoten v. Wandelt man den Teilbaum T'_v mit einer (Doppel-)Rotation in einen AVL-Baum um, so sind danach alle Knoten des Baumes balanciert.

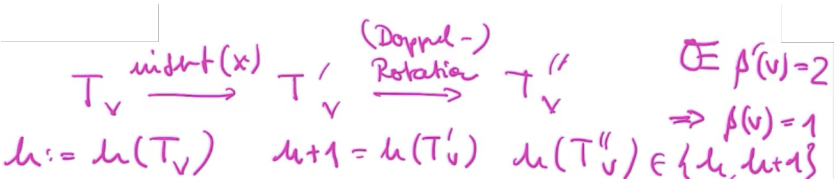


Abbildung 31: Visualisierung für den Beweis für den Fall, dass $\beta'(v) = 2$ gilt.

Korollar 6.2.17
Einfügen eines Knotens in einen AVL-Baum T kostet $\mathcal{O}(h(T))$ und damit $\mathcal{O}(\log(n))$ Zeit inklusive Wiederherstellen der Balancen.

Beispiel 6.2.18 (Löschen in einem extremalen AVL)
Der folgende extremale Baum der Höhe 6 zeigt, dass im Worst-Case an allen Knoten auf dem Weg zur Wurzel Rotationen durchgeführt werden müssen, um den Baum nach Löschen eines Blattes auszubalancieren.

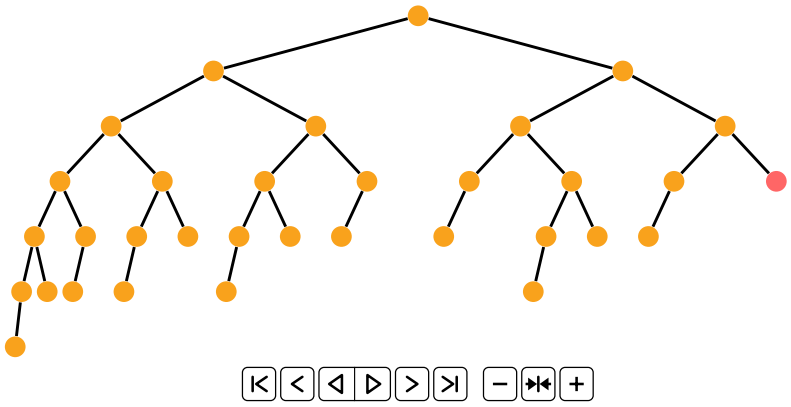


Abbildung 32: Löschen in einem extremalen AVL-Baum.

Im Gegensatz zum Einfügen, genügt es also im Allgemeinen nicht, nach `Tree-Delete` nur eine (Doppel-)Rotation an dem ersten unbalancierten Knoten auf dem Weg zur Wurzel durchzuführen. ◇

Lemma 6.2.19 (`Tree-Delete`)
Sei T ein AVL-Baum it Balancen β. Für gegebene Knoten y in T seien

$$x := \begin{cases} \texttt{Tree-Successor}(y), & \text{wenn } y \text{ zwei Kinder hat,} \\ y, & \text{sonst.} \end{cases}$$

und P der Weg zwischen T.`root` und x. Durch `Tree-Delete`(y) *entstehe ein neuer Suchbaum T'.*

(1) *Für die Höhen der Teilbäume in T' und T gilt*

$$h(T'_v) = h(T_v)\ \forall v \notin P \quad \text{und}$$
$$h(T'_v) \in \{h(T_v), h(T_v) - 1\}\ \forall v \in P \backslash \{x, y\} \quad \text{und}$$
$$h(T'_x) \in \{h(T_y), h(T_y) - 1\}\ \forall x \neq y.$$

(2) *Ist $v \in P$ in T' unbalanciert, so ist $h(T'_v) = h(T_v)$ oder $v = x$ und $h(T'_x) = h(T_y)$.*

(3) *Es gibt höchstens einen unbalancierten Knoten v in T'. Wandelt man T'_v mit einer (Doppel-)Rotation in einen AVL-Baum um, so gelten (1) und (2) (und damit auch (3)) weiterhin für alle Vorfahren von v in T'.*

Korollar 6.2.20 (Laufzeit des Löschens)
Löschen eines Knotens in einem AVL-Baum T kostet $\mathcal{O}(h(T))$ und damit $\mathcal{O}(\log(n))$ Zeit inklusive Wiederherstellen der Balancen.

Bemerkung 6.2.21 (Abbruchskriterium) Stößt man nach dem Einfügen oder Löschen eines Knotens beim Durchlaufen des Weges von x zur Wurzel auf einen Knoten v, dessen Teilbaumhöhe $h(T_v)$ unverändert ist, so sind auch die Teilbaumhöhen und Balancen aller Vorfahren von v unverändert.

Summa summarum gilt: die Klasse der AVL-Bäumen ist abgeschlossen unter den modifizierten Operationen `Tree-Insert` und `-Delete`, welche Laufzeit $\mathcal{O}(\log(n))$ haben.

6.3 Exkurs: Triangulierungen von Polygonen

Für $n \geqslant 3$ betrachte das (reguläre) n-Eck P mit den Ecken

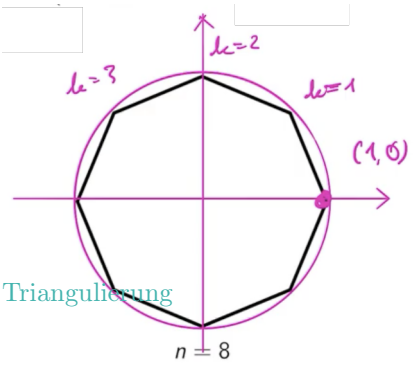
$$\left(\cos\left(\frac{2\pi k}{n}\right), \sin\left(\frac{2\pi k}{n}\right)\right)_{k=0}^{n-1} \subset \mathbb{R}^2.$$

Definition 6.3.1
Eine Triangulierung von P ist eine Zerlegung in endlich viele Dreiecke, deren Ecken auch Ecken von P sind, sodass

- je zwei Dreiecke sind in einer gemeinsamen Ecke oder Kanten schneiden (oder sich gar nicht schneiden) und
- alle Dreiecke gemeinsame P überdecken.

Eine äquivalente Charakterisierung eine Triangulierung besteht aus den $n - 3$ Diagonalen, welche sich paarweise nicht kreuzen. (Beweis per Induktion)

Bemerkung 6.3.2 Die nachfolgende Analyse gilt für beliebige konvexe Polygone.



Triangulierung

Abbildung 33: Ein reguläres (konvexes) Achteck.

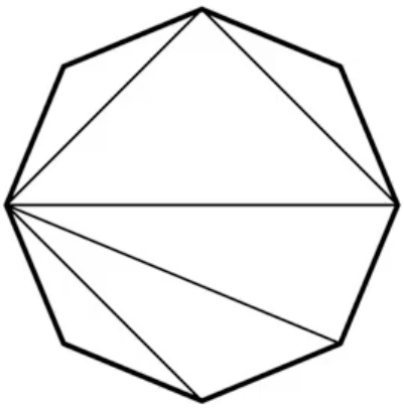


Abbildung 34: Eine Triangulierung eines regulären Achtecks.

Lemma 6.3.3 (Eigenschaften der Triangulierung)
Sei T eine Triangulierung des n-Ecks P. Dann gilt

1. *Jede Diagonale trennt P in ein k-Eck und ein $(n-k+2)$-Eck, welche beide nicht regulär seien müssen.*
2. *T hat genau $n-3$ Diagonalen und n Randkanten.*
3. *T hat genau $n-2$ Dreiecke.*
4. *Die n Ecken von P und die insgesamt $2n-3$ Kanten von T bilden einen Graphen, der überschneidungsfrei in die Ebene $\mathbb{R}^2$ eingebettet ist, d.h. dieser Graph ist* planar*.*

planar

Betrachte die folgenden verschiedenen Triangulierung desselben Achtecks, welche sich nur in einer Kante unterscheiden; das rote Viereck kann auf zwei verschieden Weisen (d.h. durch zwei verschiedene Kanten) trianguliert werden). Der Übergang von einer Triangulierung zu der anderen nennt man einen Kantenflip.

Kantenflip

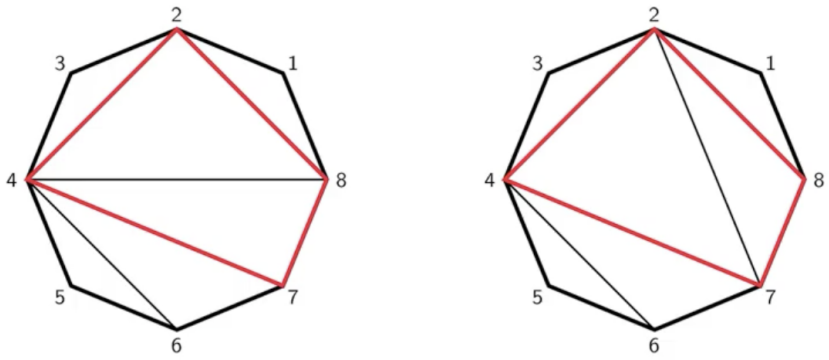


Abbildung 35: Ein Kantenflip.

Definition 6.3.4 (Dualer Graph)
Sei T eine Triangulierung von P. Der duale Graph $\Delta(T)$ ist ein ungerichteter (einfacher) Graph, sodass gilt:

1. Die Knoten in $\Delta(T)$ entsprechen den Dreiecken in T.
2. Die Kanten in $\Delta(T)$ entsprechen den zwei Dreiecken in T, die sich in einer Kante schneiden.

Theorem 6.3.1

Der duale Graph ist ein Baum.

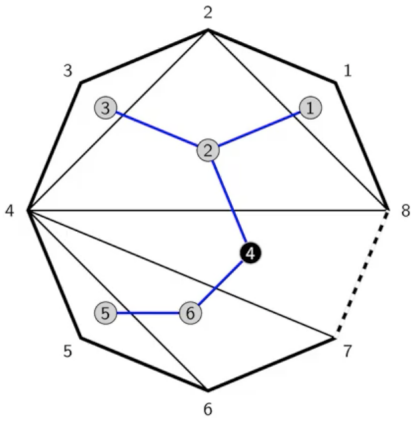


Abbildung 36: Der duale Graph einer Triangulierung.

Die Auswahl einer Randkante definiert einen Binärbaum. Die Wurzel ist der Knoten (oben: 4), welcher dem Dreieck entspricht, das die gestrichelte Kante enthält. Somit kann dieser Knoten höchstens zwei Nachbarn haben, ist also als Wurzel zulässig. Da jeder weitere Knoten höchstens drei Nachbarn hat, folgt induktiv die Binärbaumeigenschaft.

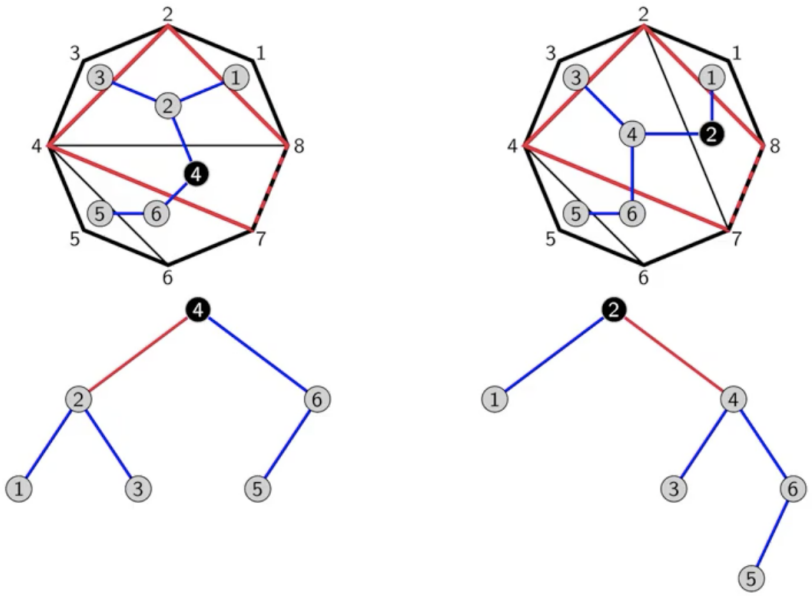


Abbildung 37: Kantenflips entsprechen Rotationen.

02.06.2020

Definition 6.3.5 (Flipdistanz)
Sei P ein reguläres n-Eck mit Triangulierungen T und T'.

$$\varphi(T, T') := \min\{k : \exists \text{ eine Folge von } k \text{ Flips von } T \text{ nach } T'\}.$$

Man stelle sich einen Graphen vor, bei dem jeder Knoten eine andere Triangulierung eines festen n-Ecks ist, und bei dem es genau dann eine Kante zwischen zwei Triangulierungen gibt, wenn die eine aus der anderen durch einen Flip hervorgeht. Der Ausdruck $\varphi(T, T')$ ist dann die Distanz von T und T' in diesem Graphen.

Es gilt $\varphi(T, T') \in \mathbb{N}$. Setze

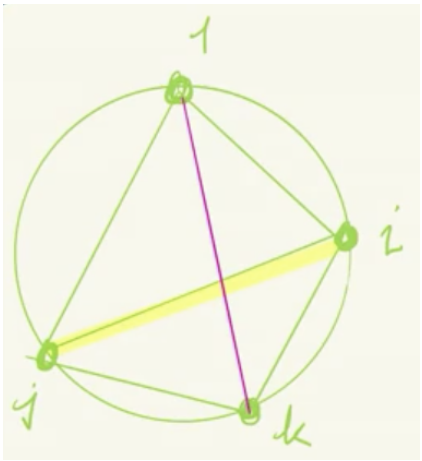
$$\Phi(n) := \max\{\varphi(T, T') : T \text{ und } T' \text{ sind Triangulierungen von } P\},$$

welches man als Durchmesser des oben beschrieben Triangulierungsgraphen sehen kann.

Theorem 6.3.2: E

gilt $\Phi(n) \leqslant 2(n-3)$.

Beweis. Sei P eine Triangulierung von P. Setze $\delta := \deg_T(1)$, wobei die Ecken von P zyklisch mit $\{1, \dots, n\}$ nummeriert sind. Wir können T mit den Ecken $\{1, \dots, n\}$ und den Randkanten von P sowie den Diagonalkanten von T als Graphen (planaren) auffassen. Es ist $\deg_T(1) = 2 + \#\text{Diagonalkanten durch } 1$. Betrachten Dreiecke $(1, i, j)$ (welche existieren, da die Triangulierung das Polygon nach Definition überdeckt), wobei $1 < i < j \leqslant n$. Ist $j \neq i + 1$, ist (i, j) eine Diagonale. Man flippt nun diese Diagonale (i, j) und erhält eine neue Triangulierung T_1 von P. Dann ist $\deg_{T_1}(1) = \delta + 1$, wie man in der Skizze auf der



rechten Seite sehen kann: da (i, j) eine Diagonale ist, existiert eine Ecke k, sodass (i, k, j) ein Dreieck ist. Die gelbe Diagonale (i, j) wird nun geflippt und ist dann die violette Diagonale $(1, k)$. Iteriere dieses Verfahren bis alle Diagonalen die 1 enthalten, und erhalten Triangulierungen $T_2, \dots, T_\ell$. Es gilt $\deg_{T_\ell}(1) = 2 + (n-3) = n-1$. Also ist $\Phi(T, T_\ell) = \ell \leqslant n-3$.

Wir können also mit höchstens $n-3$ Flips T in T_ℓ überführen und in $n-3$ Flips T_ℓ in T'. □

Die Triangulierung T_ℓ entspricht dem zu einer Liste ausgeartetem Binärbaum.

Bemerkung 6.3.6 2014 zeigte Pournin, dass $\Phi(n) = 2(n-5)$ für $n \geqslant 13$.

Sei t_n die Anzahl der Triangulierungen eines konvexen n-Ecks. Dann gilt

$$t_n = \sum_{k=0}^{n-3} t_{k+2} t_{n-k-1}$$

mit $t_2 = 1$.

Theorem 6.3.3

Es gilt

$$t_n = \frac{1}{n-1}\binom{2(n-2)}{n-2},$$

t_n ist die $(n-2)$-te Catalan-Zahl.

Korollar 6.3.7
Es gibt $\frac{1}{n+1}\binom{2n}{n}$ *binäre Wurzelbäume mit* n *Knoten.*

6.4 Splay-Bäume

Die grundlegende Idee von Self-adjusting Binary Serach Trees (Sleator und Tarjan, 1983) ist es, nach **jeder** Operation (ins. Suche) der Baum zu verändern (Operation Splay), wobei das betroffene Element „in die Wurzel wandert“. Somit sind vor kurzem genutzte Elemente nahe an der

Wurzel.

Algorithmus 20 : `Splaying-Step`(x)

```
if x.p = None then stop
else if x.p.p = None and x = x.p.left then
    Rotate-Right(x.p)                          // Fall 1a „Zick“
else if x.p.p = None and x = x.p.right then
    Rotate-Left(x)                             // Fall 1b „Zick“
else if x = x.p.left and x.p = x.p.p.left then
    Rotate-Right(x.p.p)
    Rotate-Right(x.p)                          // Fall 2a „Zick-Zick“
else if x = x.p.right and x.p = x.p.p.right then
    Rotate-Left(x.p.p)
    Rotate-Left(x.p)                           // Fall 2b „Zick-Zick“
else if x = x.p.left and x.p = x.p.p.right then
    Rotate-RL(x.p.p)                           // Fall 3a „Zick-Zack“
else Rotate-LR(x.p.p)                          // Fall 3a „Zick-Zack“
```

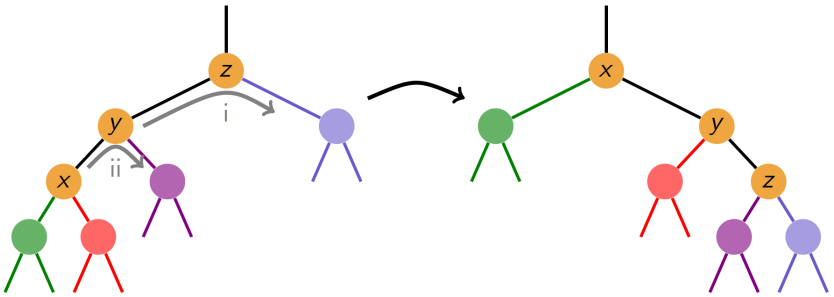


Abbildung 38: Fall 2a: Während z vor dem Splay der Großvater von x ist, ist x nach dem Splay der Großvater von z. Der Knoten y ändert seine Höhe nicht.

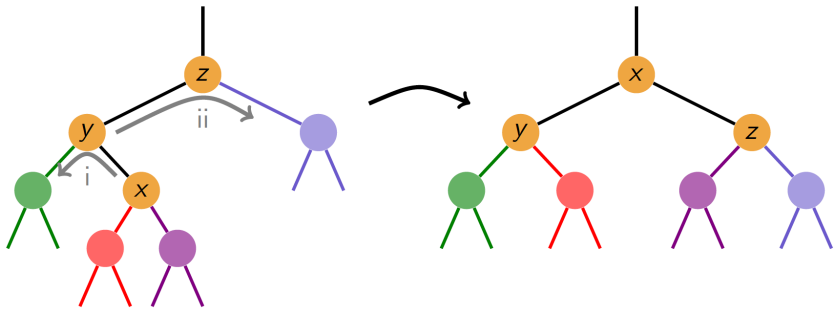


Abbildung 39: Fall 3b: Während z vor dem Splay der Großvater von x ist, ist x nach dem Splay der Vater von z. Der Knoten y ändert seine Höhe nicht.

Algorithmus 21 : `Splay`(x)

Data : Knoten x in T.

Result : Modifikation von T, sodass T`.root` $= x$.

while x.p $\neq$ None **do**

  `Splaying-Step`(x)

In den beiden obigen Schaubildern sehen wir, dass x immer ‘"nach oben wandert“, und somit terminiert Splay(x).

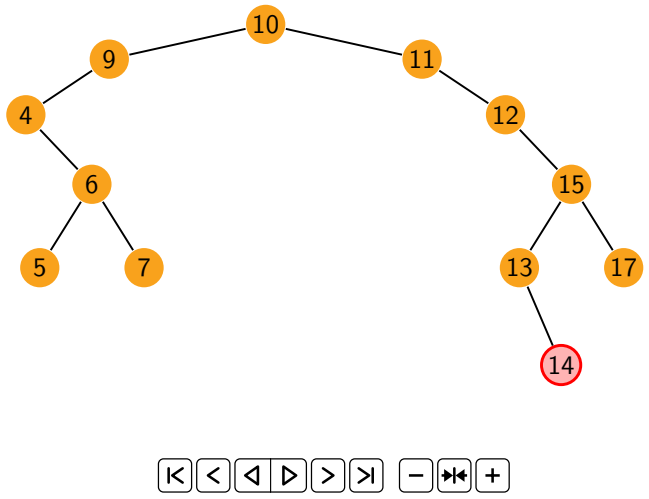


Abbildung 40: Operation `Splay` im Anschluss an die Suche nach 14.

Bemerkung 6.4.1 (Laufzeit von `Splay`) Die Operation `Splay`(x) benötigt $\mathcal{O}(d_T(x))$ Zeit, wobei $d_T(x)$ die Tiefe von x in T ist und somit $d_T(x) \leqslant h(T)$ gilt. Daher gilt: durch ausführen von Splay nach `Tree-Insert` oder `Tree-Search` ändern sich die asymptotischen Laufzeiten dieser Operationen nicht.

Insgesamt dominiert die Laufzeit für `Splay` die anderen Laufzeiten. Daher analysieren wir im Folgenden nur `Splay`.

Die Höhe jedes Knotens entlang des Weges von x nach T`.root` wird durch Ausführen von `Splay`(x) im Mittel ungefähr halbiert (siehe später). Dieser Effekt ist wesentlich für die Effizienz von Splay-Bäumen und kann nur dank der oben angesprochenen Asymmetrie der Rotationsreihenfolge erzielt werden (siehe später).

Beispiel 6.4.2 (Amortisierte Laufzeit-Analyse)
Diese wichtige Analysemethode haben wir bis jetzt in der Vorlesung noch nicht gesehen. Es geht nicht primär darum, die Kosten für jeden einzelnen Schritt an sich zu berechnen, sondern das Ziel ist, diese unmittelbaren Kosten, die für einen Schritt anfallen, in Relation zu setzen, mit dem, was man an Informationen oder an Nutzen für das jeweilige Problem gewinnt. Man nimmt in Kauf, dass man einzelne Schritte hat, die ein bisschen teurer sind, wenn sie einen weiter voranbringen.

Im konkreten Fall ist die Idee, die Qualität eines Binärbaumes T mit einer geeigneten Funktion Φ misst, die wir Potenzial nennen. Je höher der Wert $\Phi(T)$, desto besser ist der Baum.

Für eine Operation, die in Laufzeit t den Baum T in den Baum T' umwandelt, se die amortisierte Laufzeit definiert als

amortisierte Laufzeit

$$a := t + \Phi(T') - \Phi(T).$$

Eine Folge von m Operationen hat dann Laufzeit

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (a_j + \Phi(T_{j-1} - \Phi(T_j)) = \sum j = 1^m a_j + \Phi(T_0) - \Phi(T_m),$$

wobei die j-te Operation in Zeit t_j den Baum T_{j-1} in T_j umwandelt. Sind $\Phi(T_0)$ und $\Phi(T_m)$ Konstanten, so ist der Unterschied zwischen der Laufzeit, an der man interessiert ist, und der amortisierten Laufzeit, nur eine Konstante. ◇

08.06.2020

Geht das alles auch einfacher? Antworten liefert die Arbeit „Self-Organising Binary Search Trees“ von Allen und Munro aus dem Jahr 1978, die postuliert, bei jedem Zugriff den Knoten von unten nach oben zur Wurzel zu rotieren. Dafür spricht, dass, wenn die Schlüssel unabhängig mit Wahrscheinlichkeit $\frac{1}{n}$ gesucht werden, der Grenzwert der erwarteten Suchzeit $\mathcal{O}(\log(n))$ ist. Dagegen spricht, dass es für jedes $m \in \mathbb{N}$ eine Folge von m Suchoperationen mit Laufzeit $\Theta(mn)$ (bei AVL-Bäumen: $\mathcal{O}(m \log(n))$!).

Sleator und Tarjan schlangen in der Arbeit „Self-Adjusting Binary Search Trees“ aus dem Jahr 1983 folgendes vor: wenn zwei gleiche Rotationen aufeinander folgen, vertausche deren Reihenfolge. Für eine beliebige Folge von $m \in \mathbb{N}$ Suchoperationen ist hierbei die Laufzeit in $\mathcal{O}((n + m) \log(n))$.

In der folgenden Animation kann man Move-To-Root (baiserend auf „Self-Organising Binary Search Trees“, links) und Splay (rechts) vergleichen.

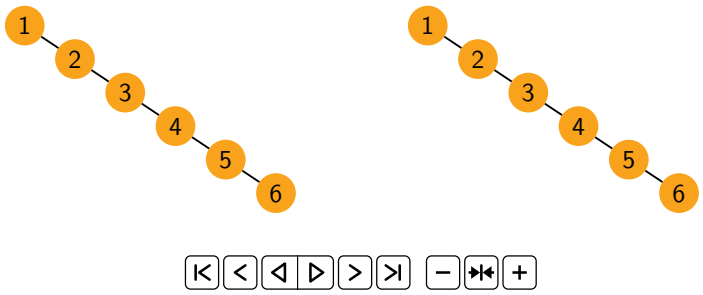


Abbildung 41: Vergleich von Move-To-Root und Splay.

Die Höhe des linken Baums ist zum Schluss fünf, die des rechten Baumes nur vier.

Definition 6.4.3 (Rang)
Sei T ein Binärbaum.

1. Für eine Knoten x in T sei $s(x)$ die Anzahl der Knoten in Teilbaum T_x.
2. Sei der Rang von x gleich $r(x) := \log_2(s(x))$.
3. Sei $\Phi(T) := \sum_{x \in T} r(x)$ das Potenzial von T.

Bemerkung 6.4.4 Ist n die Anzahl der Knoten in T, so ist $s(x) \leqslant n$ und $r(x) \leqslant \log(n)$. Folglich ist $\Phi(T) \leqslant n \log(n)$.

Lemma 6.4.5 (Access Lemma)
Die Amoritisierte Laufzeit (Anzahl Rotationen) für die Operation Splay *an einem Knoten x in eine Baum T ist durch $3(r(T.\texttt{root}) - r(x)) + 1 \in \mathcal{O}(\log(n))$ beschränkt.*

Es ist $r(T.\texttt{root}) \in \mathcal{O}(\log(n))$.
Beweis. ... □

Theorem 6.4.1

Die Gesamtlaufzeit für m Splay-Operationen ist beträgt $\mathcal{O}((n + m)\log(n))$.

Zusammenfassung. (Vergleich mit „herkömmlichen“ dynamischen Suchbäumen)

Vorteile: amortisierte Laufzeit ist kompetitiv und in vielen Situationen sogar effizienter (z.B. selbe Schlüssel werden immer wieder gesucht), kein zusätzlicher Speicherbedarf, relativ simpel und daher einfach zu implementieren.

Nachteile: Im Einzelfall: Laufzeit $\mathcal{O}(n)$ im Worst-Case (degeneriert), ständige Strukturänderungen, sogar bei Suchanfragen (schlecht für konkrete Hardware-Dinge),
Bemerkung 6.4.6 Splay-Bäume besitzt „statische Optimalität“: bezüglich der asymptotischer Laufzeit aller Anfragen sind sie so gut wie optimal (siehe später).

Man vermutet, dass Splay-Bäume sogar eine optimale dynamische Datenstruktur mit Blick auf die asymptotische Laufzeit der Anfragesequenz ist.

Es gibt eine Variante von Splay-Bäumen, die schon während der Suche „Top-Down Splaying“ betreibt.

6.5 Optimale statische Suchbäume

Problemstellung. Sei eine Schlüsselmenge $S := \{s_1, \ldots, s_n\}$ mit Zugriffshäufigkeiten $\beta_1, \ldots, \beta_n$, d.h. Schlüssel s_i wird β_i-mal gesucht.

Wir suchen einen Suchbaum T, sodass die totale Zugriffszeit klein ist, wobei die Zugriffszeit die Anzahl der Vergleiche bei der Suche nach s_i in T, welche $d_T(s_i) + 1$ ist.

totale Zugriffszeit

Definition 6.5.1 ()
Zugriffszeiten Die totale Zugriffszeit ist $C(T) := \sum_{i=1}^{n} \beta_i (d_T(s_i) + 1)$ und die mittlere Zugriffszeit $\overline{C}(T) := \frac{C(T)}{\sum_{i=1}^{n} \beta_i}$.

Definition 6.5.2 (Optimaler statischer Suchbaum)
Ein Suchbaum T ist ein optimaler statischer Suchbaum für S und

optimaler statischer Suchbaum

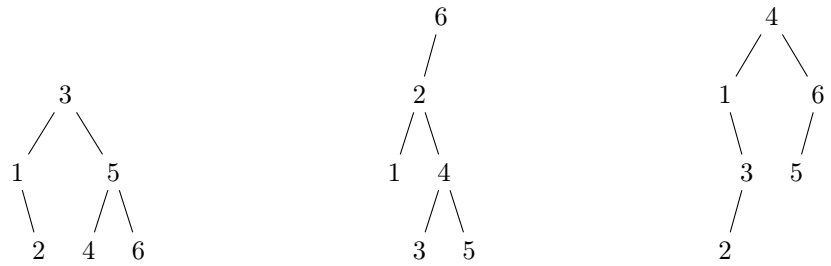
$\beta_1, \ldots, \beta_n$, wenn $C(T) \leqslant C(T')$ für alle Suchbäume T' gilt.

Bemerkung 6.5.3 In der obigen Definition können wir $C(T) \leqslant C(T')$ durch $\overline{C}(T) \leqslant \overline{C}(T')$ ersetzen.

Beispiel 6.5.4 (Optimaler statischer Suchbaum)
Betrachte die folgenden Schlüssel mit ihren Häufigkeiten:

Schlüssel s_i	1	2	3	4	5	6
Häufigkeiten β_i	20	5	10	25	10	30

und drei korrespondierende Suchbäume T_1, T_2 und T_3:



Es gilt

$$C(T_1) = 1 \cdot 10 + 2 \cdot (20 + 10) + 3 \cdot (5 + 25 + 30) = 250,$$
$$C(T_2) = 1 \cdot 30 + 2 \cdot 5 + 3 \cdot (20 + 25) + 4 \cdot (10 + 10) = 255,$$
$$C(T_3) = 1 \cdot 25 + 2 \cdot (20 + 30) + 3 \cdot (10 + 10) + 4 \cdot 5 = 205. \qquad \diamond$$

Wir nehmen als vereinfachende Annahme an, dass die Schlüsselmenge aufsteigend indiziert ist, d.h. $s_1 < s_2 < \ldots < s_n$.

Theorem 6.5.1: Prinzip der optimalen Substruktur

Seien T ein Suchbaum für S und T' ein Teilbaum von T.

1. Die Schlüsselmenge S' der Schlüssel in T' bildet eine konsekutive Teilsequenz (Intervall) $s_i < s_{i+1} < \ldots < s_j$ von $s_1 < \ldots s_n$.
2. Prinzip der optimalen Substruktur: Ist T optimal für S und $\beta_1, \ldots, \beta_n$, so ist T' optimal für S' und $\beta_i, \ldots, \beta_j$.

Prinzip der optimalen Substruktur:

Bemerkung 6.5.5 Das Prinzip der optimalen Substruktur haben wir bereits bei kürzesten Wegen und Huffman-Codes beobachtet und algorithmisch genutzt.

09.06.2020

Beweis. 3:42-12:42 □

Korollar 6.5.6
Ist T ein optimaler statischer Suchbaum für S und $\beta_1, \ldots, \beta_n$ mit Wurzel s_k, so ist $T_{s_k.\texttt{left}}$ ein optimaler statischer Suchbaum für $\{s_1, \ldots s_{k-1}\}$ und $\beta_1, \ldots, \beta_{k-1}$ und $T_{s_k.\texttt{right}}$ ein optimaler statischer Suchbaum für $\{s_{k+1}, \ldots s_n\}$ und $\beta_{k+1}, \ldots, \beta_n$.

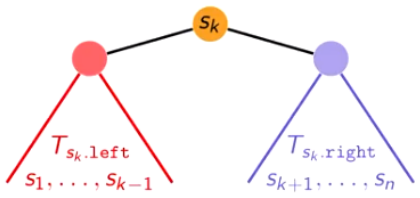


Abbildung 42: Visualisierung des nebenstehenden Korollars.

Insbesondere gilt für einen Suchbaum T mit Wurzel s_k

$$C(T) = \sum_{k=1}^{n} \beta_i + C(T_{s_k.\texttt{left}}) + C(T_{s_k.\texttt{right}})$$

Daraus entsteht bereits eine algorithmische Idee: wir testen als Wurzel $s_1, \ldots, s_n$ und berechnen jeweils optimale Teilbäume.

Korollar 6.5.7 (Algorithmische Umsetzung)
Für $0 \leqslant i \leqslant j \leqslant n$ seien

$$C_{i,j}^{\text{opt}} := \min(C(T_{i,j}) : T_{i,j} \textit{ ist ein Suchbaum für } \{s_{i+1}, \ldots, s_j\})$$

und $w_{i,j} := \sum_{q=i+1}^{j} \beta_j$. Dann gilt

$$C_{i,j}^{\text{opt}} = w_{i,j} + \min_{i+1 \leqslant k \leqslant j} (C_{i,k-1}^{\text{opt}} + C_{k,j}^{\text{opt}})$$

Die fertige algorithmische Idee ist also: berechne alle $C_{i,j}^{\text{opt}}$ und die dazugehörigen optimalen Suchbäume $T_{i,j}^{\text{opt}}$ nach aufsteigenden Differenzen $j - i$.

Beispiel 6.5.8
Betrachte $S := \{1, \ldots, 6\}$ und Häufigkeiten

s_i	1	2	3	4	5	6
β_i	10	10	20	5	30	25

für welche wir einen optimalen statische Suchbaum konstruieren wollen.

Für $j - i = 0$ gilt $i = j$ und somit $C_{i,i}^{\text{opt}} = 0$ für $i \in \{0, \ldots 6\}$. Für $j - i = 1$ gilt

$$\begin{aligned}
C_{0,1}^{\text{opt}} &= w_{0,1} + C_{0,0}^{\text{opt}} + C_{1,1}^{\text{opt}} = w_{0,1} = \beta_1 = 10 \\
C_{1,2}^{\text{opt}} &= w_{1,2} + C_{1,1}^{\text{opt}} + C_{2,2}^{\text{opt}} = w_{1,2} = \beta_2 = 10 \\
C_{2,3}^{\text{opt}} &= w_{2,3} + C_{2,2}^{\text{opt}} + C_{3,3}^{\text{opt}} = w_{2,3} = \beta_3 = 20 \\
C_{3,4}^{\text{opt}} &= w_{3,4} + C_{3,3}^{\text{opt}} + C_{4,4}^{\text{opt}} = w_{3,4} = \beta_4 = 5 \\
C_{4,5}^{\text{opt}} &= w_{4,5} + C_{4,4}^{\text{opt}} + C_{5,5}^{\text{opt}} = w_{4,5} = \beta_5 = 30 \\
C_{5,6}^{\text{opt}} &= w_{5,6} + C_{5,5}^{\text{opt}} + C_{6,6}^{\text{opt}} = w_{5,6} = \beta_6 = 25.
\end{aligned}$$

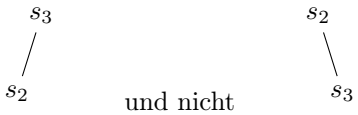
Die zugehörigen optimalen Teilbäume $T_{i,j}^{\text{opt}}$ sind (s_j).

Für $j - i = 2$ gilt

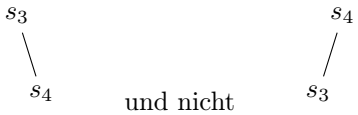
$$\begin{aligned}
C_{0,2}^{\text{opt}} &= w_{0,2} + \min_{k \in \{1,2\}} (C_{0,k-1}^{\text{opt}} + C_{k,2}^{\text{opt}}) = 20 + \min(0 + 10, 10 + 0) = 30, \\
C_{1,3}^{\text{opt}} &= w_{1,3} + \min_{k \in \{2,3\}} (C_{1,k-1}^{\text{opt}} + C_{k,3}^{\text{opt}}) = 30 + \min(0 + 20, 10 + 0) = 40, \\
C_{2,4}^{\text{opt}} &= w_{2,4} + \min_{k \in \{3,4\}} (C_{2,k-1}^{\text{opt}} + C_{k,4}^{\text{opt}}) = 25 + \min(0 + 5, 20 + 0) = 30, \\
C_{3,5}^{\text{opt}} &= w_{3,5} + \min_{k \in \{4,5\}} (C_{3,k-1}^{\text{opt}} + C_{k,5}^{\text{opt}}) = 35 + \min(0 + 30, 5 + 0) = 40, \\
C_{4,6}^{\text{opt}} &= w_{4,6} + \min_{k \in \{5,6\}} (C_{4,k-1}^{\text{opt}} + C_{k,6}^{\text{opt}}) = 55 + \min(0 + 25, 30 + 0) = 80.
\end{aligned}$$

Die zu $C_{0,2}^{\text{opt}}$ korrespondierenden Bäume sind

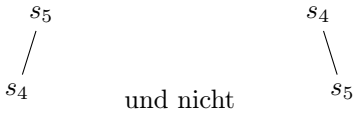
Der zu $C^{\text{opt}}_{1,3}$ korrespondierende Baum ist



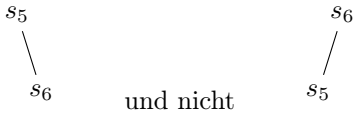
Der zu $C^{\text{opt}}_{2,4}$ korrespondierende Baum ist



Der zu $C^{\text{opt}}_{3,5}$ korrespondierende Baum ist



Der zu $C^{\text{opt}}_{4,6}$ korrespondierende Baum ist



Für $j - i = 3$ gilt

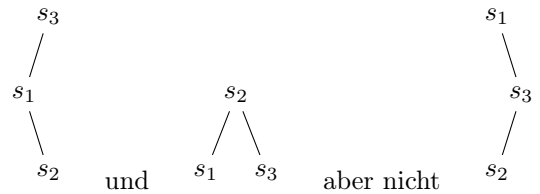
$$C^{\text{opt}}_{0,3} = w_{0,3} + \min_{k\in\{1,2,3\}}(C^{\text{opt}}_{0,k-1} + C^{\text{opt}}_{k,3}) = 40 + \min(0 + 40, 10 + 20, 30 + 0) = 70,$$

$$C^{\text{opt}}_{1,4} = w_{1,4} + \min_{k\in\{2,3,4\}}(C^{\text{opt}}_{1,k-1} + C^{\text{opt}}_{k,4}) = 35 + \min(0 + 30, 10 + 5, 40 + 0) = 50,$$

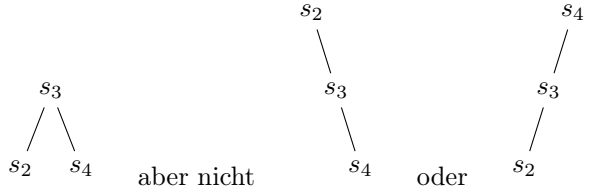
$$C^{\text{opt}}_{2,5} = w_{2,5} + \min_{k\in\{3,4,5\}}(C^{\text{opt}}_{2,k-1} + C^{\text{opt}}_{k,5}) = 55 + \min(0 + 40, 20 + 30, 30 + 0) = 85,$$

$$C^{\text{opt}}_{3,6} = w_{3,6} + \min_{k\in\{4,5,6\}}(C^{\text{opt}}_{3,k-1} + C^{\text{opt}}_{k,6}) = 60 + \min(0 + 80, 5 + 25, 40 + 0) = 90.$$

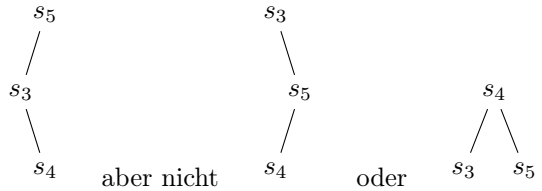
Die zu $C^{\text{opt}}_{0,3}$ korrespondierenden Bäume sind



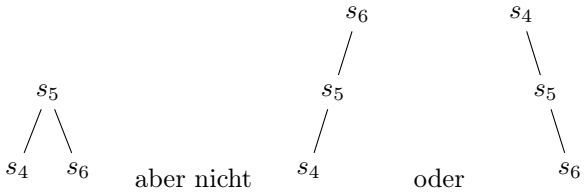
Der zu $C^{\text{opt}}_{1,4}$ korrespondierende Baum ist



Der zu $C^{\text{opt}}_{2,5}$ korrespondierende Baum ist



Der zu $C^{\text{opt}}_{3,6}$ korrespondierende Baum ist



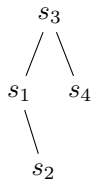
Für $j - i = 4$ gilt

$$C^{\text{opt}}_{0,4} = w_{0,4} + \min_{k\in\{1,\ldots,4\}}(C^{\text{opt}}_{0,k-1} + C^{\text{opt}}_{k,4}) = 45 + \min(0+50, 10+30, 30+5, 70+0) = 80$$

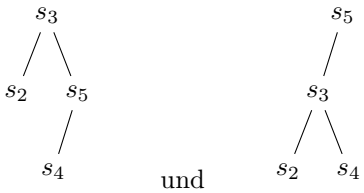
$$C^{\text{opt}}_{1,5} = w_{1,5} + \min_{k\in\{2,\ldots,5\}}(C^{\text{opt}}_{1,k-1} + C^{\text{opt}}_{k,5}) = 45 + \min(0+85, 10+40, 40+30, 50+0) = 115$$

$$C^{\text{opt}}_{2,6} = w_{2,6} + \min_{k\in\{3,\ldots,6\}}(C^{\text{opt}}_{2,k-1} + C^{\text{opt}}_{k,6}) = 80 + \min(0+90, 20+80, 30+25, 85+0) = 135.$$

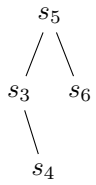
Der zu $C^{\text{opt}}_{0,4}$ korrespondierende Baum ist



Die zu $C^{\text{opt}}_{1,5}$ korrespondierenden Bäume sind



Der zu $C^{\text{opt}}_{2,6}$ korrespondierende Baum ist

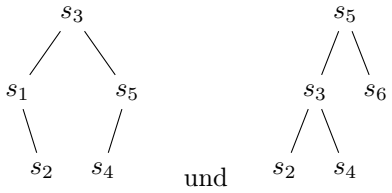


Für $j - i = 5$ gilt

$$C^{\text{opt}}_{0,5} = w_{0,5} + \min_{k\in\{1,\ldots,5\}}(C^{\text{opt}}_{0,k-1} + C^{\text{opt}}_{k,5}) = 75 + \min(0+115, 10+85, 30+40, 70+30, 80+0) = 145,$$

$$C^{\text{opt}}_{1,6} = w_{1,6} + \min_{k\in\{2,\ldots,6\}}(C^{\text{opt}}_{1,k-1} + C^{\text{opt}}_{k,6}) = 90 + \min(0+135, 10+90, 40+80, 50+25, 115+0) = 165.$$

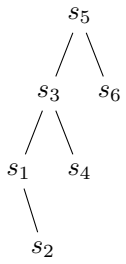
Die korrespondieren Suchbäume sind



Letztlich gilt für $j - i = 6$

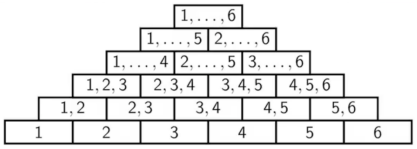
$$C^{\text{opt}}_{0,6} = w_{0,6} + \min_{k\in\{1,\ldots,6\}}(C^{\text{opt}}_{0,k-1} + C^{\text{opt}}_{k,6})$$
$$= 100 + \min(0 + 165, +10 + 135, 30 + 90, 70 + 80, 80 + 25, 145 + 0) = 205.$$

Der korrespondiere optimale Suchbaum für das gesamte Problem ist also

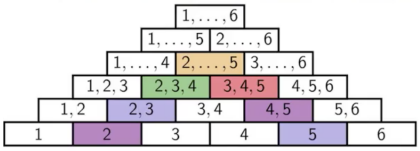


◇

Man kann sich diese dynamisches Programm so visualisieren:



Das Programm berechnet die optimalen Wert für Subprobleme zeilenweise von unten nach oben. Zum Beispiel: die Berechnung von $C^{\text{opt}}_{1,5}$ benötigt die Zahlen $C^{\text{opt}}_{1,1}$, $C^{\text{opt}}_{2,5}$, $C^{\text{opt}}_{1,2}$, $C^{\text{opt}}_{3,5}$, $C^{\text{opt}}_{1,3}$, $C^{\text{opt}}_{4,5}$, $C^{\text{opt}}_{1,4}$ und $C^{\text{opt}}_{5,5}$.



Theorem 6.5.2: Laufzeit

Der obige Algorithmus berechnet einen optimalen statischen Suchbaum in $\mathcal{O}(n^3)$ Zeit.

Beweis. Die Korrektheit folgt aus dem Prinzip der optimalen Substruktur. Die Anzahl der Paare (i, j) mit $0 \leqslant i < j \leqslant n$ ist in $\mathcal{O}(n^2)$. Die Berechnung der $w_{i,j}$ und die Berechnung der $C^{\text{opt}}_{i,j}$ benötigt jeweils $\mathcal{O}(n)$. Summa summarum benötigt also jeder der in $\mathcal{O}(n^2)$ vielen Pyramideneinträgen jeweils $\mathcal{O}(n)$. □

7 Hashing

16.06.2020

Eine Hashtabelle (eine Art verallgemeinerter Array) ist eine Datenstruktur zur Verwaltung dynamischer Mengen, welche nur die `Dictionary`-Operationen `Insert`(x), `Search`(k) und `Delete`(x) (also insbesondere nicht `predecessor` etc.) unterstützt. In Praxis kann es vorkommen, dass die `Delete`-Operation kaum von Relevanz ist.

Der Vorteil von Hashtabellen gegenüber Suchbäumen ist, dass der *mittlere* Aufwand pro Operation nur $\mathcal{O}(1)$ ist. Der Nachteil gegenüber Suchbäume ist dafür, dass der Worst-Case-Aufwand linear und somit schlechter ist, als bei Suchbäumen (immer logarithmisch).

Die Grundidee einer Hashtabelle ist es, auf die Daten direkt zugreifen zu können. Das heißt der Schlüssel soll so etwas werden wie der Index (bzw. Zeiger) eines Arrays.

Beispiel 7.0.1 (Einfachste Variante: Direct adress table)
Wir nehmen an, dass die Menge U der möglichen Schlüsselwerte relativ klein ist: $U := \{0, 1, \dots, m-1\}$ mit kleinem $m \in \mathbb{N}$. Ferner soll jedes Element x einen unterschiedlichen Schlüsselwert `x.key` haben.

Wir stellen die dynamische Menge durch einen Array $T[0, 1, \dots, m-1]$ dar, mit einen Eintrag für jeden möglichen Schlüsselwert. **TODO Bild aus Cormen, $m = 10$, T ist ein Array von Listen.**

Diese Konzept ist jedoch nicht so sinnvoll, da viel Speicher gebraucht wird, um verhältnismäßig wenig Daten zu speichern. ◇

Allgemeiner ist eine Hashtabelle. Wir nehmen an, dass die Menge U der möglichen Schlüsselwerte viel größer als die interessante Teilmenge K mit $|K| =: n \ll |U|$ ist. Die Schlüssel der n Datensätze sind a priori unbekannt.

Hashtabelle

Wir erwarten, dass der Speicheraufwand ungefähr n entspricht, also insbesondere unabhängig von (der Größe von) U ist.

Dafür verwenden wir einen Array $T[0, \dots, m-1]$ mit $m \in \Theta(n)$ und $m \ll |U|$. Wir speichern (den Zeiger auf) den Datensatz mit Schlüssel k im Feld $T[h(k)]$ für ein Hashfunktion $h\colon U \to \{0, \dots, m-1\}$.

Hashfunktion

Da h nicht injektiv seien kann, gibt es $k_1, k_2 \in U$ mit $h(k_1) = h(k_2)$. Dies nennt man Kollision. **TODO: Bild aus Cormen, die Kollisionen illustriert.**

todo: Geburtstagsparadoxon ...

Eine gute Hashfunktion sollte möglichst gleichmäßig über den Adressraum $\{0, \dots, m-1\}$ streuen. Im Idealfall wird jede Adresse $j \in \{0, \dots, m-1\}$ mit Wahrscheinlichkeit $\frac{1}{m}$ gewählt („uniform hashing“).

uniform hashing

Ist $p(k)$ die Wahrscheinlichkeit für Auftreten des Schlüssels k in einer Iteration, so soll also

$$\sum_{\substack{k \in U: \\ h(k) = j}} p(k) = \frac{1}{m}$$

für alle $j \in \{0, \dots, m-1\}$ gelten.

Beispiel 7.0.2 (Divisionsmethode)
Wir nehmen an, dass $U \subset \mathbb{Z}$ gilt. Als Hashfunktion wählen wir

$$h\colon U \to \{0,\dots,m-1\},\ k \mapsto k \mod m.$$

Wählen wir $m = 2^q$ mit $q \in \mathbb{N}$, so schneidet h die vorderen, höherwertigen Bits der Binärdarstellung von k ab und behält nur die q niedrigsten Bits.

Dies kann jedoch zu Häufungen von Kollisionen führen. Empirisch gut ist daher eine Primzahl m, die weit von einer Zweierpotenz entfernt ist.◇

Beispiel 7.0.3 (Multiplikationssmethode)
Wir nehmen an, dass $U \subset \mathbb{Z}$ gilt. Als Hashfunktion wählen wir

$$h\colon U \to \{0,\dots,m-1\},\ k \mapsto \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

für ein festes $A \in (0,1)$.

Die Zahl $x - \lfloor x \rfloor \in [0,1)$ ist der Dezimalanteil von x. todo: Bild aus dem Cormen für $m = 2^p$.

Ist $m = 2^p$, wähle $A = \frac{\sqrt{5}-1}{2} \in \mathbb{R} \setminus \mathbb{Q}$ (goldener Schnitt).

Für $k = 123456$, $m = 100000$ und $A = \frac{\sqrt{5}-1}{2}$, gilt z.B.

$$h(k) = \left\lfloor 100000 \cdot \underbrace{(123456 \cdot A - \lfloor 123456 \cdot A \rfloor)}_{\approx 0.0041151\dots} \right\rfloor = 41. \qquad \diamond$$

7.1 Umgang mit Kollisionen

Unabhängig von der Wahl der Hashfunktion ist die Wahl der Kollisionsumgangsstrategie.

Hashing mit Verkettung (chaining)

Die Idee hierbei ist, dass man pro Hashfeld eine doppelt verkettete Liste anlegt und alle Datensätze, die den selben Hashwert bekommen, in dieser Liste ablegt. **(todo:Bild aus dem Cormen)**

`Insert`(x) benötigt $\mathcal{O}(1)$, da $h($`x.key`$)$ berechnet wird $(\mathcal{O}(1))$ und x zu Beginn der entsprechenden Liste eingefügt wird $(\mathcal{O}(1))$.

`Delete`(x) benötigt $\mathcal{O}(1)$, Löschen aus einer doppelt verketten Liste in $\mathcal{O}(1)$ ist (setzt voraus, dass die Position von x in T bekannt ist).

`Search`(k) benötigt $\mathcal{O}(1 +$ Länge der Liste$)$, da $h(k)$ berechnet wird $(\mathcal{O}(1))$ und die entsprechende Liste durchsucht wird.

Wir analysieren die zu erwartende Länge der liste unter den folgenden Annahmen:

- Es gibt n Datensätze in der Hashtabelle der Größe m. Die Auslastung ist also $\alpha := \frac{n}{m}$.
- uniform hashing: $h(k) \in \{0,\dots,m-1\}$ ist gleichverteilt und unabhängig für alle $k \in U$.

Lemma 7.1.1
Die erwartete Länge der Liste ist bei erfolgloser Suche α und bei erfolgreicher Suche $1 + \frac{n-1}{m} < 1 + \alpha$.

Beweis. Sei $n_{h(k)}$ die Zufallsvariable, welche die Länge der Liste mit Adresse $h(k)$ angibt.

Fall 1: erfolglose Suche. T enthält k nicht, also $k \notin \{k_1, \ldots, k_n\}$. Nehmen wir an, dass $h(k_1), \ldots, h(k_n), h(k)$ unabhängig und gleichverteilt in $\{0, \ldots, m-1\}$ sind. Seien $n_{h(k)}$ die Länge der Liste, in der k gesucht wird, und $X_i := \mathbb{1}_{h(k_i)=h(k)}$ für $i \in \{1, \ldots, m\}$. Dann gilt $n_{h(k)} = \sum_{i=1}^{n} X_i$ und $\mathbb{E}[X_i] = \Pr[h(k_i) = h(k)] = \frac{1}{m}$ für alle $i \in \{0, \ldots, m-1\}$. Es folgt

$$\mathbb{E}(n_{h(k)}) = \sum_{i=1}^{n} \mathbb{E}[x_i] = \frac{m}{m}.$$

Fall 2: erfolgreiche Suche. k ist in T enthalten. Es sind $n-1$ Schlüssel verschieden, $k \in \{k_1, \ldots, k_n\}$, sagen wir $k = k_j$. Dann gilt $X_j = 1$ und $\mathbb{E}[x_i] = \frac{1}{m}$ für alle $i\{1, \ldots, n\}\backslash\{j\}$ und somit

$$\mathbb{E}[n_{h(k)}] = \sum_{i=1}^{n} \mathbb{E}[X_i] = 1 + \frac{n-1}{m}. \qquad \square$$

Korollar 7.1.2

1. *Wählt man $m = \Theta(n)$, so gilt $\alpha \in \mathcal{O}(1)$ und alle Operation erfordern im Mittel Aufwand $\mathcal{O}(1)$.*
2. *Der Worst-Case ($h(k) = c$ für alle $k \in U$) Aufwand für `Search`(k) ist jedoch $\Omega(n)$, weil wir dann nur eine doppelt verkettete Liste benutzen.*

Beispiel 7.1.3 Ist $n \approx 2000$, so sind drei Vergleiche bei Suche im Mittel ok, wähle also m so, dass $\alpha \approx 3$ ist.

Bei der Divisionsmethode kann man z.B. $m = 701$ wählen (weit entfernt von 512 und 1024), d.h. $h(k) := k \mod 701$.

In diesem Beispiel *bedingt* die mittlere Anzahl der Vergleiche also die Größe der Hashtabelle. Im Allgemeinen gibt es immer einen Tradeoff zwischen Größe der Hashtabelle (Speicher) und Anzahl der Vergleiche beim Suchen (Zeit). $\diamond$

7.2 Universelles Hashing

22.06.2020

Die Idee ist, den oben angesprochenen Worst-Case zu vermeiden. Dazu stelle man sich einen böswilligen Gegenspieler (adversary) vor, welche das Hashingverfahren kennt und es durch die Auswahl spezieller Schlüssel in den Worst-Case treibt.

Wir können das verhindern, in dem wir eine (für das gesamte Verfahren gleiche) zufällige Hashfunktion benutzen.

Definition 7.2.1 (Universelle Hashfunktionenklasse)
Eine (endliche) Menge H von Hashfunktionen ist universell bezüglich

der Größe m der Hashtabelle, wenn für alle verschiedenen $x, y \in U$

$$\big|\{h \in H : h(x) = h(y)\}\big| \leqslant \frac{|H|}{m}$$

gilt.

Bemerkung 7.2.2 ist H universell für m und wählt man h zufällig und gleichverteilt aus H aus, so kollidieren x und y (d.h. $h(x) = h(y)$) mit Wahrscheinlichkeit kleiner gleich $\frac{1}{m}$.

Theorem 7.2.1: Universelles Hashing

Seien H universell für m und $h \in H$ zufällig und gleichverteilt gewählt. Füge $n \leqslant m$ Schlüssel mittels Hashing mit Verkettung ein. Dann ist für $k \in U$ die erwartete Länge der Liste an der Position $h(k)$ durch

$$\begin{cases} \alpha, & k \text{ nicht in der Liste,} \\ 1 + \alpha, & k \text{ in der Liste.} \end{cases}$$

beschränkt.

Beweis. Für verschiedene Schlüsse $k, \ell \in U$ sei $X_{k,\ell} \coloneqq \mathbb{1}_{\{h(k)=h(\ell)\}}$.

Nach Bemerkung 7.2.2 gilt $\Pr\{h(k) = h(\ell)\} \leqslant \frac{1}{m}$. Somit folgt $\mathbb{E}_h[X_{k,\ell}] \leqslant \frac{1}{m}$, wobei $\mathbb{E}_h$ der Erwartungswert über alle $h \in H$.

Sei Y_k die Zufallsvariable, welche angibt, wie viele Schlüssel außer k in T auf $h(k)$ gehasht werden. Dann gilt $Y_k = \sum_{\substack{\ell \in T \\ \ell \neq k}} X_{k,\ell}$ und somit

$$\mathbb{E}_h[Y_k] = \sum_{\substack{\ell \in T \\ \ell \neq k}} \mathbb{E}_h[X_{k,\ell}] \leqslant \sum_{\substack{\ell \in T \\ \ell \neq k}} \frac{1}{m}.$$

Der letzte Ausdruck ist die erwartete Länge der Liste zur Adresse $h(k)$, die aus Lemma 7.1.1 bekannt ist. □

Korollar 7.2.3

Bei universellem Hashing mit Verkettung benötigt man $\Theta(n)$, um eine Sequenz von n `Insert`-, `Search`- und `Delete`-Operationen zu verarbeiten, welche $\mathcal{O}(m)$ `Insert`-Operationen beinhaltet.

Beispiel 7.2.4 (Existenz universeller Hashklassen)

Sei p eine Primzahl mit $p > x$ für alle $x \in U \subset \mathbb{N}$ und $p > m$.

Für $a \in \mathbb{Z}_p^* \coloneqq \{1, \ldots, p-1\}$ und $b \in \mathbb{Z}_p \coloneqq \{0, \ldots, p-1\}$ definiere die Hashfunktion

$$h_{a,b}(x) \coloneqq ((ax + b) \mod p) \mod m$$

Wähle z.B. $U \subset \{0, \ldots, 16\}$, $m = 6$, $p = 17$, $a = 3$ und $b = 4$. Dann gilt $h_{3,4}(8) = ((24 + 4) \mod 17) \mod 6 = 11 \mod 6 = 5$.

Betrachte $H \coloneqq \{h_{a,b} \colon a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ mit $|H| = (p-1)p > m$.

Wir zeigen, dass H bezüglich m universell ist.

Beweis. Seien $k, \ell \in U \subset \mathbb{Z}_p$ zwei verschiedene Schlüssel und $(a, b) \in \mathbb{Z}_p^* \times \mathbb{Z}_p$ fest.

Seien

$$r := (ak + b) \mod p \quad \text{und} \quad s := (a\ell + b) \mod p.$$

Dann gilt

$$r - s \equiv \underbrace{a}_{\neq 0} \underbrace{(k - \ell)}_{\neq 0} \mod p.$$

und somit $r \neq s$.

Bestimme a und b aus $k, q\ell, r, s$:

$$a = (r - s)\left[(k - \ell)^{-1} \mod p\right] \mod p \quad \text{und} \quad b = (r - ak) \mod p$$

Damit ist $(a, b) \mapsto (r, s)$ bijektiv.

Für zufällige gleichverteilte $(a, b) \in \mathbb{Z}_p^* \times \mathbb{Z}_p$ sind also auch (r, s) zufällig gleichverteilt.

Somit gilt für die Kollisionswahrscheinlichkeit

$$\Pr\{h_{a,b}(k) = h_{a,b}(\ell) = \Pr\{r \equiv s \mod m\} = \left\lfloor \frac{p}{m} \right\rfloor - 1 \leqslant \frac{(p-1)}{m}.$$

Die Wahrscheinlichkeit, dass $r \equiv s \mod m$ gilt, ist höchstens $\frac{\frac{p-1}{m}}{p-1} = \frac{1}{m}$.□

7.3 Hashing mit offener Adressierung

Wir suchen nach einer alternativen Art der Kollisionsauflösung, da die Struktur Arrays von Listen kompliziert ist. Eine Möglichkeit ist, „es mit den Hashwerten nicht so genau zu nehmen“: man speichert mutwillig die Schlüssel in der Hashtabelle, komme was wolle. Anstelle von Überlauflisten wie bisher, nimmt man bei Kollision einfach den nächsten freien Platz in der Tabelle. Das ist nur eine gute Idee solange die Hashtabelle hinreichend groß ist ($m \geqslant n$).

Was ist diese nächstbeste Ersatzadresse? Wir betrachten eine modifizierte Hashfunktion

$$h\colon U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\star} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Adresse}},$$

wobei $\star$ die Anzahl der erfolglosen Versuche bei Suche nach einem freien Eintrag ist. Die Modifikation besteht darin, die Anzahl der erfolglosen Versuche miteinzubeziehen. Ist ein Hashfeld schon besetzt (d.h. erfolgloser Versuch), so soll beim nächsten Versuch ein andere Hashfeld herauskommen. Es ist $h(k, i)$ die Adresse bei $(i + 1)$-ten Versuch, eine freie Adresse für den Schlüssel k zu finden und man berechnet nacheinander die Adresse $h(k, 0)$, $h(k, 1)$, $h(k, 2)$, …, bis eine freie Adresse gefunden wurde.

Damit (bei $n \leqslant m$) immer eine freie Adresse gefunden wird, benötigen wir die Permutationsbedingung: für alle $k \in U$ definiert $i \mapsto h(k, i)$ eine Permutation von $\{0, \ldots, m-1\}$. Diese Bedingung garantiert, dass z.B. die Folge $(h(k, i))_{i=0}^{m-1}$ sind nicht wiederholt.

Permutationsbedingung

Achtung: dies erschwert das Löschen von Datensätzen, da man nicht weiß, in welchem Versuch die freie Adresse gefunden wurde. In der Praxis wird deshalb Hashing mit offener Adressierung nur verwendet, wenn man Löschen explizit verbietet.

Beispiel 7.3.1 (Lineares Sondieren)
Für eine gewöhnliche Hashfunktion h' betrachte

$$h(k,i) := (h'(k) + i) \mod m.$$

Man hat also eine konkrete Idee $h'(k)$, was die Adresse für k sein soll. Wenn diese schon besetzt ist, geht man ein Feld weiter.

Ein Vorteil ist, dass die Permutationsbedingung erfüllt ist.

Ein Nachteil ist „primäres Clustering“: es bilden sich immer größere Blöcke konsekutiver belegter Adressen. Das macht den Vorteil von Hashing, das man für viele Operationen hintereinander, eine über die Summe verteilte Laufzeit habe, die konstant pro Operation ist, zunichte. Hier degeneriert es, und es wird schnell zu einem linearer durchsuchen: sind q konsekutive Adressen $a, a+1, \ldots, (a+q-1) \mod m$ belegt, so wird eine neues k mit zufälligem und gleichverteilten $h(k)$ mit Wahrscheinlichkeit $\frac{q+1}{m}$ an der Adresse $a+q$ abgelegt. ◇

Beispiel 7.3.2 (Quadratisches Sondieren)
Betrachte wie oben

$$h(k,i) := (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m,$$

wobei $c_1, c_2 \in \mathbb{N}$ zwei Parameter sind. Die Idee ist, dass die Sprungweite quadratisch zunimmt. Ob die Permutationsbedingung gilt, hängt von c_1, c_2 und m ab.

Ein Vorteil ist, dass kein primäres Clustering auftritt. Ein Nachteil ist, das „sekundäres Clustering“ auftritt: ist $h'(k_1) = h'(k_2)$, so stimmt auch die Folge der Ersatzadressen überein. ◇

Die beste Möglichkeit, um mit offener Adressierung umzugehen, ist, dass man zwei gewöhnliche Hashfunktionen kombiniert. Dies ist in der Praxis gängig.

Beispiel 7.3.3 (Doppel-Hash)
Für gewöhnliche Hashfunktionen h_1 und h_2 betrachte

$$h(k,i) := (h_1(k) + i \cdot h_2(k)) \mod m$$

Man nennt h_1 die primäre und h_2 die sekundäre Hashfunktion. Primäres und sekundäres Clustering wird vermieden, den $h_1(k) = h_1(k')$ führt zu verschiedenen Ersatzadressen, wenn $h_2(k) \neq h_2(k')$.

Wann ist die Permutationsbedingung erfüllt? Für $m = 8$ und $h_2(k) = 2$ ist die Permutationsbedingung für k nicht erfüllt, da wenn $h_1(k)$ gerade ist, auch $h_1(k) + ih_2(k)$ immer gerade, so werden nur die geraden Adressen in der Hashtabelle ausprobiert. Ist hingegen $h_2(k') = 3$, so ist die Permutationsbedingung erfüllt. ◇

Theorem 7.3.1

Doppel-Hashing erfüllt die Permutationsbedingung genau dann wenn $\mathrm{ggT}(m, h_2(k)) = 1$ für alle $k \in U$ gilt.

Beweis. Übungsaufgabe. □

23.06.2020

Wie viele Sonderierungen müssen durchgeführt werden, bis eine freie Adresse gefunden wird? Wir beantworten diese Frage unter der Gleichverteilungsannahme: die nächste Sondierung wählt immer zufällig und gleichverteilt einer der m (Größe der Hashtabelle, n ist die Anzahl der Elemente in der Hashtabelle) Adressen.

Theorem 7.3.2: Analyse: Hashing mit offener Adressierung

Bei einer Auslastung $\alpha = \frac{n}{m} < 1$ ist unter der Gleichverteilungsannahme die erwartete Anzahl der Sondierungen beim Einfügen höchstens $\frac{1}{1-\alpha}$.

Beispiel 7.3.4 Ist $\alpha \leqslant \frac{1}{2}$, also die Tabelle höchstens nur bis zur Hälfte gefüllt, so erwartet man höchstens zwei Sondierungen.

Ist die Tabelle zu höchstens 80% gefüllt, so erwarten wir höchstens fünf Sondierungen.

Insbesondere ist $\frac{1}{1-\alpha} \xrightarrow{\alpha \nearrow 1} \infty$ und $1 \leqslant \frac{1}{1-\alpha}$ für $\alpha \in (0,1)$. ◇

Für den Beweis wird etwas Wahrscheinlichkeitstheorie benötigt, die wir hier wiederholen.

Eine diskrete Zufallsvariable X kann die Werte aus $\{x_1, \dots, x_n\}$ annehmen. Die entsprechenden Wahrscheinlichkeiten bezeichnen wird mit $p_i := \Pr[X = x_i]$ für $i \in \{1, \dots, n\}$. Sie müssten $\sum_{i=1}^{n} p_i = 1$ und $p_i \in [0,1]$ für alle $i \in \{1, \dots, n\}$ erfüllen. Der Erwartungswert der Zufallsvariable X ist $\mathbb{E}[X] := \sum_{i=1}^{n} x_i \cdot p_i$. Erwartungswert

Beispiel 7.3.5 (Fairer Würfel) Beschreibt X die Augenzahl eines fairen Würfels, so ist $\{x_1, \dots, x_n\} = \{1, \dots, 6\}$ und $p_i = \frac{1}{6}$ für alle $i \in \{1, \dots, 6\}$. Somit ist $\mathbb{E}[X] = \frac{7}{2}$. Es muss also kein i mit $x_i = \mathbb{E}[X]$ geben.

Beschreibt Y die Summe der Augenzahlen von zwei fairen Würfeln, so ist $\{x_1, \dots, x_n\} = \{2, \dots, 12\}$, $p_i = \frac{1}{36} \min(i, 12 - i)$ und $\mathbb{E}[Y] = 7$. ◇

Wir führen nun diskrete Zufallsvariablen auf einen abzählbar unendlichen Raum ein: eine Zufallsvariable X kann nun die Werte $\{x_i\}_{i \in \mathbb{N}}$ annehmen und p_i sind analog definiert und es muss $\sum_{i \in \mathbb{N}} p_i = 1$ gelten. Der Erwartungswert von X ist $\mathbb{E}[X] := \sum_{i=1}^{\infty} x_i \cdot p_i$, wenn die **absolut** Reihe konvergent ist (damit Umordnung der Reihe den Wert nicht ändert).

Beispiel 7.3.6 (Urnenmodell)
Betrachte eine Urne, in welche m Kugeln liegen, davon sind w weiß und $s := m - w$ schwarz. Für eine zufällig (gleichverteilt) gezogenen Kugel k gilt $\Pr[k \text{ weiß}] = \frac{w}{m} =: p$ und $\Pr[k \text{ schwarz}] = \frac{m-w}{m} = 1 - p$.

Sei X die Anzahl Ziehungen (mit Zurücklegen) bis eine weiße Kugel gezogen wird. Dann gilt $\Pr[X = i] = (1-p)^{i-1} \cdot p$ für $i \in \mathbb{N}$. Es folgt

$$\mathbb{E}[X] = \sum_{i \in \mathbb{N}} i(1-p)^{i-1} \cdot p = p \cdot \sum_{i \in \mathbb{N}} i(1-p)^{i-1} = \frac{1}{p} = \frac{m}{w}. \qquad \diamond$$

Beweis. (von Satz 7.3.2) Die Sonderierungen entsprechen Ziehen aus

einer Urne (freie Plätze = weiße Kugeln), also

$$\mathbb{E}[\#\text{ Ziehungen bis freier Platz gefunden}] = \frac{m}{m-n} = \frac{1}{1-\alpha}. \qquad \square$$

Wir haben gezeigt, dass das Einfügen in eine Hashtabelle mit offener Adressierung konstante Zeit benötigt, wenn die Auslastung α als eine Konstante betrachtet wird. Sofern nicht bereits Element in Einfügefolge gelöscht wurden, ist die Suche genau wie das Einfügen. Bei offener Adressierung beschränkt man sich deshalb meist auf Hashing ohne Löschen.

Summa summarum ist der Aufwand pro Operation im Mittel $O(1)$, aber im Worst-Case $\Omega(n)$ (wenn nur noch ein Eintrag frei ist, muss man die ganze Liste durchsuchen). Bei Suchbäumen benötigen wir nur eine Ordnung auf der Menge der Schlüssel, beim Hashing müssen wir auch mit den Schlüsseln rechnen können (z.B. $U \subset \mathbb{N}$). In der Praxis ist oft beides gegeben und somit können wir beide Methoden vergleichen. Bei Suchbäumen ist der Aufwand pro Operation in Worst-Case $\Omega(\log(n))$, jedoch ist auch der Aufwand im Mittel $\Omega(\log(n))$, da mindestens die Hälfte aller Knoten in Tiefe von mindestens $\Omega(\log(n))$ liegen. Man „bezahlt“ also dafür, dass man im Worst-Case schneller als linear ist damit, dass man auch im Mittel nur logarithmisch ist.

7.4 Perfektes Hashing

Ähnlich zu statisches Suchbäumen gibt es statische Hashtabellen: alle relevanten Schlüssel sind a priori bekannt z.B. reservierte Worte in `python` oder die Namen aller Dateien auf einer DVD.

Wir wollen den Worst-Case-Aufwand für die Suche von $\Omega(\log(n))$ auf $O(1)$ verbessern, ohne die $O(1)$ im Mittel zu verlieren.

Definition 7.4.1
Ein Hashing-Verfahren heißt perfekt, wen der Worst-Case-Aufwand für das Suchen konstant ist.

Eine Möglichkeit besteht darin, ein zweistufiges Verfahren zu konstruieren. Zunächst gibt es eine primäre Hash-Tabelle der Größe $m = n$ mit einer Hash-Funktion h. Die Konflikte an einer Adresse $j \in \{0, \ldots, m-1\}$ werden durch eine **kollisionsfreie** sekundäre Hash-Tabelle der Größe $m_j = n_j^2$ gelöst, wobei n_j die Anzahl der Schlüssel x mit $h(x) = j$ sind. Die Einträge der primären Hash-Tabelle sind also die sekundären Hash-Tabellen.

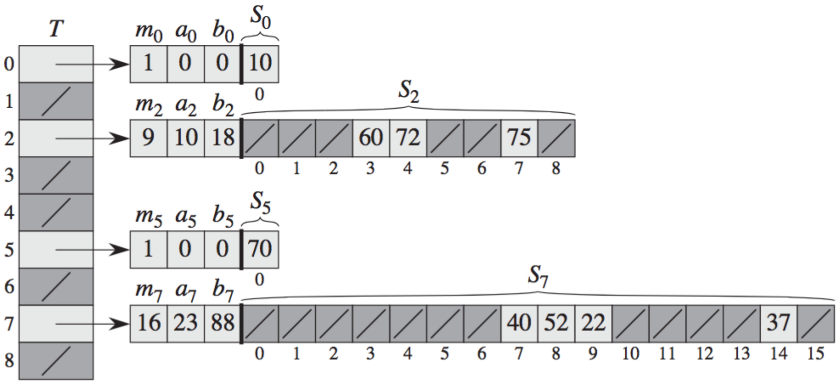


Abbildung 43: Die Größe der primären Hash-Tabelle ist $m = n = 1+3+1+4 = 9$. Es ist $h_0(10) = 0$ und $n_0 = 1$ und somit $m_0 = 1$. Die Parameter a_i und b_i codieren die Hash-Funktion: $h_i(x) = ((a_i x + b_i) \mod p) \mod m$. [Cormen]

Die Zahlen n_j werden vorab bestimmt (nach der Wahl von h), da alle Schlüssel bekannt sind. Die primäre Hash-Funktion h und die sekundären Hash-Funktionen $(h_j)_{j=0}^{m-1}$ werden aus einer universellen Klasse gewählt.

Realisierung

Sei p eine Primzahl mit $p > x$ für alle n Schlüssel $x \in \mathbb{N}$ und $m := n$. Wir wählen nun geeignete $a \in \mathbb{Z}_p^*$ und $b \in \mathbb{Z}_p$ und setze $h(x) := ((ax + b) \mod p) \mod m$. Für $j \in \{0, \ldots, m-1\}$ wähle $a_j \in \mathbb{Z}_p^*$ und $b_j \in \mathbb{Z}_p$ und setze $h_j(x) := ((a_j x + b_j) \mod p) \mod m_j$ mit $m_j := n_j^2$, wobei n_j die Anzahl der Schlüssel x ist, für die $h(x) = j$ gilt. Gilt $n_j = 1$, so wählen wir $a_j = b_j = 0$ und $m_j^2 = 1$.

Zwei mögliche Probleme

Problem A: Wie stellt man sicher, das beim sekundären Hashing keine Kollisionen auftreten?

Theorem 7.4.1

Werden n_j Schlüssel mit einer zufällig aus einer universellen Klasse gewählten Hash-Funktion h_j den $m_j := n_j^2$ Adressen zugewiesen, so ist die Wahrscheinlichkeit für das Auftreten einer Kollision kleiner als $\frac{1}{2}$.

Beweis. Wir haben n Schlüssel und somit $\binom{n_j}{2}$ Paare von Schlüsseln, die kollidieren können. Eine Kollision passiert mit Wahrscheinlichkeit $\frac{1}{m_j}$, da wir eine universelle Familie haben. Sei X ein Zufallsvariable, welche die Anzahl der Kollisionen darstellt. Dann gilt

$$\mathbb{E}[X] = \binom{n_j}{2} \cdot \frac{1}{n_j^2} = \frac{n_j^2 - n_j}{2n_j^2} < \frac{1}{2}.$$

Die Markov'sche Ungleichung besagt, dass $\Pr[X \geqslant t] \leqslant \frac{1}{t}\,\mathbb{E}[X]$ für

eine nichtnegative Zufallsvariable X und eine Zahl $t > 0$ gilt. Also folgt $\Pr[X \geqslant 1] \leqslant \mathbb{E}[X] < \frac{1}{2}$. □

Korollar 7.4.2

Durch wiederholtes zufälliges Ziehen von h_j kann man mit hoher Wahrscheinlichkeit Kollisionsfreiheit erreichen.

Produziert man eine DVD und hat die Dateinamen, so wählt man die primäre Hash-Funktion und errechnet alle Kollisionen. Nach der Berechnung der (zufälligen) passenden sekundären Hash-Funktionen. Geht das schief (mit Wahrscheinlichkeit $< \frac{1}{2}$), wählen wir eine neue Hash-Funktion und haben somit die Wahrscheinlichkeit, dass es wieder schief geht, auf kleiner als $\frac{1}{4}$ reduziert. Die Wahrscheinlichkeit, dass es beim n-ter Versuch immer noch schief geht, ist also kleiner als $\frac{1}{2^n}$, fällt also sehr schnell.

B: Wie groß ist der benötigte Speicherplatz für die primäre und die sekundären Hash-Tabellen?

Theorem 7.4.2

Werden n Schlüssel mit einer zufällig aus einer universellen Klasse gewählten Hash-Funktion h den $m := n$ Adressen zugewiesen, so ist

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n.$$

Beweis. Sei $a \in \mathbb{N}$. Dann gilt $a^2 = a + 2\binom{a}{2}$. Somit folgt

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] = \mathbb{E}\left[\sum_{j=0}^{m-1} n_j + 2\binom{n_j}{2}\right] = \underbrace{\sum_{j=0}^{m-1} \mathbb{E}[n_j]}_{=n} + 2\,\mathbb{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right].$$

Der Ausdruck $\sum_{j=0}^{m-1} \binom{n_j}{2}$ beschreibt die Anzahl der Paare der Hash-Tabelle, die kollidieren. Da die primäre Hash-Funktion aus der universellen Familie kommt ist die Kollisionswahrscheinlichkeit $\frac{1}{m}$ und somit gilt

$$\mathbb{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \leqslant \frac{1}{m}\binom{n}{2} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

da $n = m$. Es folgt

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \leqslant n + \frac{n-1}{2} = 2n - 1 < 2n.$$ □

Korollar 7.4.3

Der erwartete Speicherbedarf für das zweistufige Hashing-Verfahren ist $O(n)$.

Korollar 7.4.4

Durch wiederholtes Ziehen einer zufälligen primären Hash-Funktion kann man mit hoher Wahrscheinlichkeit erreichend, dass $\sum_{j=0}^{m-1} n_j^2 < 4n$

8 Turingmaschinen und Komplexität

29.06.2020

8.1 Berechenbarkeit

In CoMa I haben wir einen Algorithmus als eine Rechenvorschrift definiert, die die erwartete Eingabe genau spezifiziert, die auf der Eingabe durchzuführende Rechenschritte inklusive deren Reihenfolge eindeutig angibt und spezifiziert, wann der Algorithmus terminiert und was er dann ausgibt.

Diese Definition wollen wir nun präzisieren. Wir setzen dafür das (primitive, aber sehr mächtige) Rechnermodell der Turingmaschine voraus.

Definition 8.1.1 (Alphabet, Wort, Sprache)
Sei $\Sigma \neq \varnothing$ eine endliche Menge. Für $k \in \mathbb{N}$ ist Σ^k die Menge der Funktionen $f\colon \{1,\dots,k\} \to \Sigma$. Eine Funktion $f \in \Sigma^k$ kann man auch als Folge $f(1)\dots,f(k)$ schreiben und nennt es Wort oder Zeichenkette der Länge k über dem Alphabet Σ. Wort, Alphabet

Das einzige Element von Σ^0 nennt man leeres Wort. Ferner sei $\Sigma^* := \bigcup_{k=0}^{\infty} \Sigma^k$. Eine Teilmenge $L \subset \Sigma^*$ heißt Sprache über dem Alphabet Σ. Sprache

Definition 8.1.2 (Berechnungsproblem)
Sei $P \subset D \times E$ eine Relation, sodass es zu jedem $d \in D$ ein $e \in E$ mit $(d,e) \in P$ gibt. Dann heißt P Berechnungsproblem. Berechnungsproblem

- Ist $(d,e) \in P$, so ist e eine korrekte Ausgabe für das Problem P mit Eingabe d.
- Die Elemente von D sind die Instanzen des Problems.
- Ist P eine Funktion, so ist das Problem P eindeutig.
- Sind D und E Sprachen über einem endliche Alphabet Σ, so ist P ein diskretes Berechnungsproblem.
- Ist $D \subset \mathbb{R}^n$ und $E \subset \mathbb{R}^m$ mit $n,m \in \mathbb{N}_{>0}$, so ist P ein numerisches Berechnungsproblem.
- Ein eindeutiges Berechnungsproblem $P\colon D \to E$ mit $|E| = 2$ (z.B. ja oder nein) ist ein Entscheidungsproblem.

Beispiel 8.1.3
Ein Entscheidungsproblem ist es, zu einer gegebenen natürlichen Zahl zu entscheiden, ob es sich um eine Primzahl handelt. ◇

Gibt es zu jedem Berechnungsproblem einen Algorithmus, der es löst? Nein, denn es gibt mehr Problem als Algorithmen (s. u.).

Definition 8.1.4 (Mächtigkeit)
Zwei Mengen A und B heißen gleichmächtig und wir schreiben $|A| = |B|$, wenn es eine bijektive Abbildung von A nach B gibt. gleichmächtig

Eine Menge A heißt endlich, wenn es eine injektive Funktion $f\colon A \to$

$\{1, \dots, n\}$ für ein $n \in \mathbb{N}$ gibt und andernfalls unendlich.

Eine Menge A heißt abzählbar, wenn es eine injektive Funktion $f\colon A \to \mathbb{N}$ gibt, andernfalls überabzählbar.

Theorem 8.1.1

Sei Σ eine nichtleere endliche Menge. Dann ist Σ^* abzählbar.

Korollar 8.1.5

Die Menge aller `python`*-Programme ist abzählbar.*

Beweis. Betrachte das endliche Alphabet Σ der Menge der ASCII Zeichen, also Klein-und Großbuchstaben, Zahlen, Sonderzeichen etc. Für die Sprache der `python`-Programme L gilt $L \subset \Sigma^*$. □

Theorem 8.1.2

Die Menge $\{0,1\}^{\mathbb{N}}$ aller Funktionen von $\mathbb{N}$ nach $\{0,1\}$ ist überabzählbar.

Diese Menge kann als 2-adische Dezimalbrüche mit führender Ziffer Null interpretiert werden.

Korollar 8.1.6

Es gibt Entscheidungsproblem, die kein `python`*-Programm löst.* *Entscheidungsproblem*

Eines der wichtigsten Entscheidungsprobleme ist das Halteproblem. Gegeben ist ein Algorithmus (`python`-Programm) A und ein passender Input x. Terminiert A mit Input x nach endlich vielen Schritten? Halteproblem

Theorem 8.1.3

Das Halteproblem wird von keinem `python`-Programm gelöst.

Definition 8.1.7 (Sprache ↔ Entscheidungsproblem)

Sei Σ eine nichtleere endliche Menge.

- Das zu einer Sprache $L \subset \Sigma^*$ (über dem Alphabet Σ) gehörende Entscheidungsproblem ist

$$P_L\colon \Sigma^* \to \{\text{True}, \text{False}\},\ x \mapsto \begin{cases} \text{True}, & \text{wenn } x \in L, \\ \text{False}, & \text{sonst.} \end{cases}$$

- Die zu einem Entscheidungsproblem $P\colon \Sigma^* \to \{\text{True}, \text{False}\}$ gehörende Sprache ist $L_P := \{x \in \Sigma^* : P(x) = \text{True}\} \subset \Sigma^*$.

Sprachen und Entscheidungsprobleme können also synonym zu einander

verwendet werden.

8.2 Turing-Maschinen

Beispiel 8.2.1 (Intuitive Beschreibung der Turingmaschine)
Eine Turingmaschine (TM) kann man sich als Abstraktion eines Rechners vorstellen.

Der Speicher der TM ist ein unbeschränktes Band bestehend aus abzählbar unendlich viele Zellen. Jede Zelle kann genau ein Zeichen des endlichen Bandalphabets Γ. Der Lese-/Schreibkopf der TM schaut jeweils auf eine Zelle des Bandes, deren Zeichen er lesen und überschreiben kann.

Der endliche Input-String der TM steht anfangs auf dem band und der Lese-/Schreibkopf schaut auf das erste Zeichen des Inputs. Links und rechts des Inputs stehen lauter Leerzeichen $B \in \Gamma$.

Die TM befindet sich zu Beginn im Anfangszustand q_0 aus der endlichen Zustandsmenge Q. In jedem Rechenschritt der TM geschieht das Folgende: der Kopf ließt das Zeichen auf dem Band und überschreibt es und bewegt sich dann maximal eine Zelle nach links oder nach rechts (oder bleibt auf der Zelle stehen). Danach geht die TM in einen neuen Zustand aus Q über. ◇

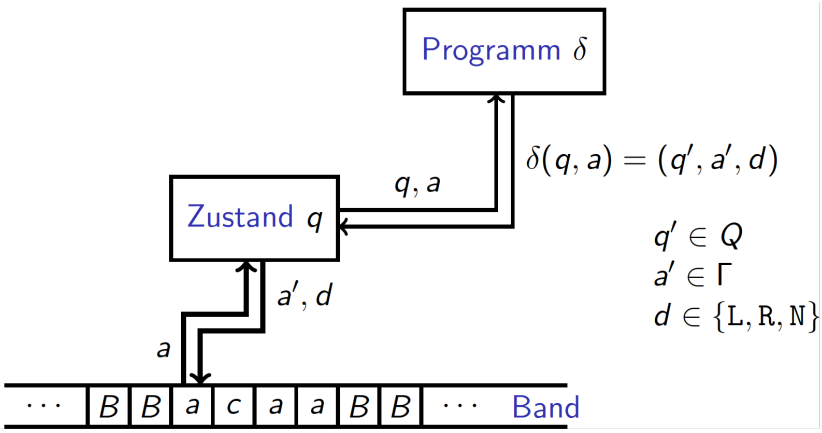


Abbildung 44: Hier ist $\Gamma = \{a, c, a', d, B, \ldots\}$.

30.06.2020

Definition 8.2.2 (Deterministische Turingmaschine)
Eine deterministische Turingmaschine (DTM) ist gegeben durch

- eine endliche Zustandsmenge Q und einen Anfangszustand $q_0 \in Q$,
- ein endliches Eingabealphabet Σ,
- ein endliches Bandalphabet $\Gamma \supset \Sigma$ und ein Leerzeichen $B \in \Gamma \backslash \Sigma$,
- ein Programm $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{R}, \mathtt{L}, \mathtt{N}\}$,
- eine Menge akzeptierender Endzustände $F \subset Q$.

Ein Zustands $q \in Q$ heißt Endzustand, falls $f(q, a) = (q, a, \mathtt{N})$ für alle $a \in \Gamma$ gilt. Ist DTM im Zustand $q \in Q$ und liest das Zeichen $a \in \Gamma$ mit

$\delta(q,a) = (q,a,\mathtt{N})$, so terminiert sie. Terminiert die DTM, so zeigt ihr Kopf auf das erste Zeichen des Outputs. Der Output endet vor dem ersten Leerzeichen rechts des Kopfes. Terminiert die DTM in einem Zustand $q \in F$, so akzeptiert sie den Input. Anfangs zeigt der Lese-/Schreibkopf auf das erste Zeichen des Inputs. Neben dem Input stehen auf dem Band nur Leerzeichen.

Beispiel 8.2.3 (Erste Turing-Maschine)
Seien $Q := \{q_0, q_1, \ldots, q_6\}$, $\Sigma := \{0,1\}$, $\Gamma := \Sigma \cup \{B\}$ und $F := \{q_6\}$ sowie δ gegeben durch

δ	0	1	B
q_0	$(q_0, 0, \mathtt{R})$	$(q_1, 1, \mathtt{R})$	–
q_1	–	$(q_1, 1, \mathtt{R})$	$(q_2, B, \mathtt{L})$
q_2	–	$(q_3, B, \mathtt{L})$	–
q_3	$(q_3, 0, \mathtt{L})$	$(q_3, 1, \mathtt{L})$	$(q_4, B, \mathtt{R})$
q_4	$(q_5, B, \mathtt{R})$	–	–
q_5	$(q_0, 0, \mathtt{N})$	–	$(q_6, B, \mathtt{N})$
q_6	–	–	–

Hier bedeutet – in Zeile q und Spalte a den Eintrag $(q, a, \mathtt{N})$. ◇

Lemma 8.2.4
Die obige DTM entscheidet die Sprache $L := \{0^n 1^n : n \geqslant 1\}$, d.h. sie terminiert nach endlich vielen Schritten und akzeptiert den Input genau dann, wenn er in L ist.

Betrachte z.B. die Eingabe 01, d.h. das Band sieht am Anfang so aus: $\ldots Bq_0 01B \ldots$, wobei der Kopf stets rechts von dem Zustandszeichen q_k steht. Anwenden von δ wird mit $\mapsto$ abgekürzt. Dann ist die Konfigurationsfolge

$$\begin{aligned} Bq_0 01B &\vdash B0q_0 1B \vdash B01q_1 B \vdash B0q_2 1B \vdash Bq_3 0B \vdash Bq_3 B0B \\ &\vdash Bq_4 0B \vdash Bq_5 B \vdash Bq_6 B, \end{aligned}$$

also wird die Eingabe akzeptiert.

Zum Beweis: Man ist nur im Zustand q_0, wenn nur Nullen gelesen worden sind. Sobald man eine eins liest, gelangt man in Zustand q_1. Man kommt also nur in Zustand 1, wenn man erst nur Nullen liest und dann eine Eins liest. Der Zustand q_2 zeigt: die gesamte Eingabe besteht aus nur Nullen und dann nur Einsen. Der Zustand ist wie q_2, aber löscht die letzte Eins. Der Zustand wie q_4 ist wie q_3, aber der Kopf steht auf B links der Eingabe. Der Zustand q_5 ist q_4, aber erste Null gelöscht. Wir haben also genau so viel Einsen wie Nullen gelöscht. Der Zustand q_6 signalisiert, dass die Eingabe akzeptiert ist.

Definition 8.2.5
Eine Funktion $f\colon D \to E$ heißt berechenbar (oder total rekursiv), wenn es eine DTM gibt, die bei Input $x \in D$ nach endlich vielen Schritten terminiert und den Output $f(x)$ liefert. berechenbar

Eine Sprache $L \subset \Sigma^*$ heißt entscheidbar (oder rekursiv), wenn das entscheidbar

zugehörige Entscheidungsproblem P_L berechenbar ist.

Eine Sprache $L \subset \Sigma^*$ heißt semi-entscheidbar (rekursiv aufzählbar), wenn es eine DTM gibt, die bei Input $x \in \Sigma^*$ genau dann nach endlich vielen Schritten im akzeptierten Endzustand terminiert, wen $x \in L$ gilt. semi-entscheidbar

In der Definition der Semi-Entscheidbarkeit wird nichts ausgesagt, für den Fall, das $x \notin L$. Insbesondere ist es erlaubt, das die DTM nicht terminiert, wenn $x \in L$. Entscheidbare Sprachen sind semi-entscheidbar.

Beispiel 8.2.6 (Weiter Turing-Maschine)
Seien $Q := \{q_0, q_1, \ldots, q_3\}$, $\Sigma := \{0, 1\}$, $\Gamma := \Sigma \cup \{B\}$ und $F := \varnothing$ sowie δ gegeben durch

δ	0	1	B
q_0	$(q_0, B, \mathtt{R})$	$(q_1, 1, \mathtt{R})$	–
q_1	$(q_1, 0, \mathtt{R})$	$(q_1, 1, \mathtt{R})$	$(q_2, B, \mathtt{L})$
q_2	$(q_2, 1, \mathtt{L})$	$(q_3, 0, \mathtt{L})$	–
q_3	$(q_3, 0, \mathtt{L})$	$(q_3, 1, \mathtt{L})$	$(q_0, B, \mathtt{R})$

Hier bedeutet – in Zeile q und Spalte a den Eintrag $(q, a, \mathtt{N})$. ◇

Die DTM startet in Zustand q_0 und löscht und geht nach rechts, so lange sie Nullen liest. Ist die Eingabe verbraucht, sind wir in eine Endzustand. Liest die DTM eine Eins, so geht sie in Zustand q_1. **Rest selber überlegen**

Definition 8.2.7 (Konfiguration einer DTM)
Eine DTM befinden sich in der Konfiguration aqb, wobei $q \in Q$ und $a, b \in \Gamma^*$, wenn der Zustand der DTM q ist, auf dem Band der String B^*abB^* steht und der Lese-/Schreibkopf auf das erste Symbol von b zeigt. Konfiguration

Die Konfiguration $a'q'b'$ ist direkte Nachfolgekonfiguration von aqb, wenn die DTM in einem Schritt von aqb nach $a'q'b'$ gelangt. Wir schreiben dann $aqb \vdash a'q'b'$. direkte Nachfolgekonfiguration

Die Konfiguration $a''q''b''$ ist Nachfolgekonfiguration von aqb, wenn die DTM in endlich vielen Schritten von aqb nach $a''q''b''$ gelangt. Wir schreiben dann $aqb \vdash^* a''q''b''$.

Für die erste DTM gilt also

$$\begin{aligned} Bq_00011 &\vdash 0q_0011 \vdash 00q_011 \vdash 001q_11 \vdash 0011q_1B \vdash 001q_21 \vdash 00q_31 \\ &\vdash 0q_301 \vdash Bq_3001 \vdash Bq_3B001 \vdash Bq_4001 \vdash Bq_501 \vdash Bq_001 \\ &\vdash^* Bq_6B \end{aligned}$$

Eine wichtige Technik zur Programmierung von TM ist die Daten aus einer endlichen Menge A (z.B. $A \subseteq \Gamma^k$) durch Integration in die Zustandsmenge Q zu speichern: setze $Q_{\text{neu}} = Q \times A$. Ein Zustandswechsel beinhaltet dann immer eine Änderung des eigentlichen Zustands und des Speichers, der Speicher wird also permanent gelesen.

Eine andere Technik ist die Verwendung eine mehrspuringen TM. Das Band eine k-Spur-TM besteht aus k Spuren, sodass in jeder Zelle des

Bandes k Zeichen stehen, d.h. ein Element aus Γ^k. Für den Input kann beispielsweise die erste Spur genutzt werden und für den Output die k-te. Eine k-Spur-TM kann als herkömmlich TM mit Bandalphabet Γ^k aufgefasst werden. Insbesondere sind k-Spur-TM nicht mächtiger als gewöhnlich TM.

Die Lese-/Schreibköpfe der k Bänder können sich unabhängig bewegen. Die TM liest und schreibt in jedem Schritt auf allen k Bändern, d.h. $\delta\colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{L}, \mathtt{R}, \mathtt{N}\}^k$.

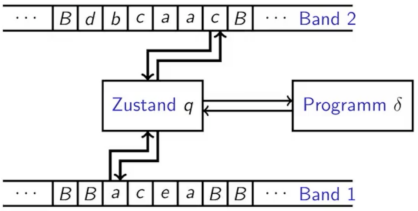


Abbildung 45: Schematische Darstellung einer 2-Band-TM.

Beispiel 8.2.8 (Zwei-Band TM)
Seien $Q := \{q_0, q_1, \ldots, q_3\}$, $\Sigma := \{0, 1\}$, $\Gamma := \Sigma \cup \{B\}$ und $F := \{q_3\}$ sowie δ gegeben durch

δ	q_0	q_1	q_2	q_3
00	$(q_0, 00, \mathtt{R})$	$(q_0, 10, \mathtt{R})$	$(q_2, 00, \mathtt{L})$	–
01	$(q_0, 11, \mathtt{R})$	$(q_1, 01, \mathtt{R})$	$(q_2, 01, \mathtt{L})$	–
0B	$(q_0, 0B, \mathtt{R})$	$(q_0, 1B, \mathtt{R})$	$(q_2, 0B, \mathtt{L})$	–
10	$(q_0, 10, \mathtt{R})$	$(q_1, 00, \mathtt{R})$	$(q_2, 10, \mathtt{L})$	–
11	$(q_1, 01, \mathtt{R})$	$(q_1, 11, \mathtt{R})$	$(q_2, 11, \mathtt{L})$	–
1B	$(q_0, 1B, \mathtt{R})$	$(q_1, 0B, \mathtt{R})$	$(q_2, 1B, \mathtt{L})$	–
B0	$(q_0, 00, \mathtt{R})$	$(q_0, 10, \mathtt{R})$	$(q_2, B0, \mathtt{L})$	–
B1	$(q_0, 11, \mathtt{R})$	$(q_1, 01, \mathtt{R})$	$(q_2, B1, \mathtt{L})$	–
BB	$(q_2, BB, \mathtt{L})$	$(q_0, 1B, \mathtt{R})$	$(q_3, BB, \mathtt{L})$	–

Eine weitere Technik ist die Verwendung von Unterprogrammen, welche in einem bestimmten Zustand aufgerufen werden und gewissen Daten übergeben bekommen. Diese Daten kann man auf eine eigene Spur schreiben, mit welcher das Unterprogramm dann arbeitet. Eine Teilmenge der Zustände wir nur für das Unterprogramm verwendet.

Eine letzte Technik sind Schleifen, die man als spezielle Unterprogramme mit bestimmten Abbruchkriterien realisiert. Für `for`-Schleifen kann man einen Zähler verwenden (s.o.).

06.07.2020

Beispiel 8.2.9 (Noch eine 2-Band TM)
Seien $Q := \{q_0, \ldots, q_3\}$, $\Sigma := \{0, 1\}$, $\Gamma := \Sigma \cup \{B\}$, $F := \{q_3\}$ und δ gegeben durch

δ	q_0	q_1	q_2	q_3
0 0	–	–	$(q_2, 0B, \texttt{L R})$	–
0 1	–	–	–	–
0 B	$(q_0, 00, \texttt{R R})$	–	–	–
1 0	–	–	–	–
1 1	–	–	$(q_2, 1B, \texttt{L R})$	–
1 B	$(q_0, 11, \texttt{R R})$	–	–	–
B 0	–	$(q_1, B0, \texttt{N L})$	–	–
B 1	–	$(q_1, B1, \texttt{N L})$	–	–
B B	$(q_1, BB, \texttt{N L})$	$(q_2, BB, \texttt{L R})$	$(q_3, BB, \texttt{R N})$	–

Hierbei besteht zu Beginn das erste Band aus der eigentliche Eingabe und das zweite Band ist leer.

Welche Inputs auf Band Eins akzeptiert diese DTM? Wird im Input eine Null (oder eine Eins) gelesen, so wird diese auf das zweite Band kopiert und beide Köpf bewegen sich simultan nach rechts, und der nächste Zustand ist immer noch q_0. Der Zustand ändert sich, wenn der Kopf an das Ende der Eingabe gelangt und B B liest. Der Kopf des ersten Bands („erster Kopf“) bleibt stehen und der zweite Kopf wird nach links bewegt, also auf das letzte Zeichen der kopierten Eingabe und die TM befindet sich nun im Zustand q_1, in welchem nur die Köpfe bewegt werden, aber das Band unverändert bleibt. Im Zustand q_1 wird der zweite Kopf an den Anfang der Eingabe bewegt. Dort angekommen, geht die TM in Zustand q_2 über, und der erste Kopf zeigt auf das letzte Zeichen der Eingabe und der zweite Kopf auf das erste.

In q_2 passiert nur etwas nennenswertes, wenn die Bandeinträge identisch sind. Ist das der Fall, wird auf dem zweiten Band gelöst und die Köpfe „über Kreuz“ weiterbewegt. Stehen zwei verschieden Einträge auf den Bändern, wird die Eingabe nicht akzeptiert.

Somit sind die akzeptierten Eingaben genau die Palindrome. ◇

Beispiel 8.2.10 (Multiplikation zweier n-Bit-Binärzahlen)
Man schreibt die beiden Faktoren auf zwei unterschiedliche Bänder und wendet das Prinzip der schriftlichen Multiplikation an. Auf einem dritten Band addiert man dann die n Summanden. ◇

Hat man vier oder fünf Zahlen, die in dem Programm eine Rolle spielen, kann es sinnvoll sein, für jede Zahl ein eigenes Band zu benutzen. Interessante Programme sind so gestrickt, dass man nicht weiß, wie groß die Eingabe ist, z.B. bei Sortierverfahren. Die (endliche!) Anzahl der Bänder muss vor Beginn des Algorithmus festgelegt werden. In diesem Fall kann man Datenblöcke hintereinander auf ein Band schreiben und durch ein „Sonderzeichen“ trennen. Durch Zählen der Sonderzeichen können bestimmte Speicheradressen gefunden werden. Wird ein Datenblock durch einen längeren ersetzt, muss eventuell der gesamte Bandinhalt nach rechts verschoben werden.

Definition 8.2.11 (Laufzeit und Speicherplatz)
Die Rechenzeit einer DTM für einen bestimmten Eingabestring ist die Anzahl der Zustandsübergänge, bis die Maschine stoppt.

Rechenzeit

Die Worst-Case-Rechenzeit $t(n)$ ist die maximale Anzahl der Rechenschritte, welche die Maschine auf Inputs aus Σ^n durchführt.

Der Speicherplatz einer DTM für einen bestimmten Eingabestring ist die Anzahl der Zellen, auf die der Kopf der Maschine jemals zeigt.

Der Worst-Case-Speicherplatz $s(n)$ ist der maximale Speicherplatz, den die Maschine auf Inputs aus Σ^n nutzt.

Beispiel 8.2.12
Für die DTM aus Beispiel 8.2.3 ist $t(n) \in \Theta(n^2)$ und $s(n) = n + 2$. Für die DTM aus Beispiel 8.2.6 ist $t(n) \in \Theta(2^n)$ und $s(n) = n + 2$. Für die DTM aus Beispiel 8.2.9 ist $t(n) \in \Theta(n)$ und $s(n) = 2n + 4$. ◇

Beweis. (Für das letzte Beispiel) Für eine Eingabe der Länge n benötigen wir weitere n Zellen für die kopierte Eingabe auf Band zwei und jeweils die Leerzeichen links und rechts der Eingabe. Somit ist $s(n) = 2(n + 2)$.

Die Laufzeit ist linear in der Länge der Eingabe, da die DTM sich genau $n + 1$ mal im Zustand q_0 befindet, dann $n + 1$ mal im Zustand q_1 und dann n mal im Zustand q_2. □

8.3 Universelle Turing-Maschinen

Bislang haben wir TM als „special purpose Rechner“ konzipiert, d.h. als für eine feste Aufgabe programmiert. Jetzt wollen wir die TM als „general purpose Rechner“, der Programm und speziellen Input enthält, auf der das Programm abgearbeitet wird.

Betrachte eine TM M mit $\Sigma := \{0, 1\}$, $\Gamma := \{0, 1, B\} = \{x_1, x_2, x_3\}$, $D_1 := \mathtt{L}$, $D_2 := \mathtt{R}$, $D_3 := \mathtt{N}$, $Q := \{q_1, \ldots, q_t\}$, einem Anfangszustand q_1 und $F := \{q_2\}$. Es ist keine einschränkende Annahme, dass es nur einen akzeptierenden Zustand gibt, da man sonst in konstanter Laufzeit und mit konstantem Speicherplatz eine Zustand hinzufügen kann, der von allen akzeptierenden Zuständen in eine neuen Zustand übergeht. Kodiere nun die z-te Zeile, wobei $1 \leqslant z \leqslant s := |Q| \cdot |\Gamma|$, des Programms δ von M wie folgt:

$$\delta(q_i, x_j) = (q_k, x_\ell, D_m) \longleftrightarrow 0^i 1 0^j 1 0^k 1 0^\ell 1 0^m = \text{code}(z),$$

wobei $k \in \{1, \ldots, t\}$ und $k, \ell \in \{1, 2, 3\}$ sind und die Eins als Trennzeichen benutzt wird.

Definition 8.3.1 (Gödel-Nummer)
Die Gödel-Nummer der TM M mit s-zeiligem Programm δ ist

$$\langle M \rangle := 111\text{code}(1)11\text{code}(2)11 \ldots 11\text{code}(s)111$$

Hierbei wird 11 als Trennzeichen verwendet.

Korollar 8.3.2
Es gibt nur abzählbar viele Programme.

Beweis. Die Gödel-Nummer kann als natürliche Zahl gelesen werden. □

Definition 8.3.3
Eine universelle TM U erhält als Inputpaar $\langle M \rangle$ und w, wobei M eine TM und $w \in \{0,1\}^*$ ist. U simuliert M auf Input w.

universelle TM

Man kann U als 3-Band-TM realisieren.

Beobachtung Für festes M ist die Rechenzeit von U auf dem Input $\langle M \rangle, w$ nur um einen konstanten Faktor größer als die Rechenzeit von M auf w.

Beispiel 8.3.4 (Universelle TM als 3-Band-TM)

1. Test die Eingabe auf Korrektheit, d.h. $\langle M \rangle$ muss z.B. von der Form $1110^i10^j10^k10^\ell10^m11\ldots$ mit $i,k \geqslant 1$ und $j,\ell,m \in \{1,2,3\}$ sein. Keine zwei Codes dürfen mit demselben Präfix 0^i10^j beginnen.
2. Kopiere $\langle M \rangle$ auf Band zwei, überschreibe es auf Band eins mit Leerzeichen.
3. Schreibe Zustand q_i von M auf Band drei, kodiert als 0^i.
4. In jedem Schritt:
   - Lies x_j auf Band eins, suche $110^i10^j1\ldots$ auf Band zwei, wobei 0^i auf Band drei steht.
   - Akzeptiere, falls $i = 2$.
   - Andernfalls lies $0^k10^\ell10^m11$ auf Band zwei, schreibe 0^k auf Band drei, schreibe x_ℓ auf Band 1, bewege Kopf von Band eins in Richtung D_m. ◇

8.4 Die Church'sche These und Registermaschinen

Die Church'sche These ist „Die durch die formale Definition der TM erfasste Klasse berechenbarer Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein."

Bemerkung 8.4.1 (Zur Church'schen These)
Die Interpretation ist, das das TM-Modell mindestens genau so mächtig ist, wie jedes andere bekannte Berechnungsmodell. Außerdem erfasst die TM Rechenzeiten im Vergleich zu anderen Modellen bis auf polynomielle Faktoren richtig (z.B. Registermaschine (s.u.)). Die Church'sche These ist prinzipiell nicht beweisbar, da der Begriff „intuitiv berechenbar" nicht exakt formalisiert werden kann. Sie könnte jedoch falsifiziert werden, indem nicht Turing-berechenbare Funktionen als berechenbar „nachgewiesen" werden.

07.07.2020

Zur Untermauerung der Church'schen These betrachten wir ein weiteres formales Rechnermodell, welches näher an reale Rechner angelehnt ist.

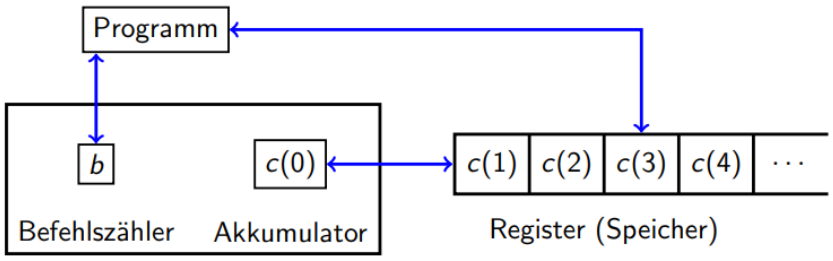


Abbildung 46: Schematische Darstellung einer Registermaschine.

Beispiel 8.4.2 (Random Access Machine - RAM)
Die Registermaschine hat, wie die TM, einen unendlich langen Speicher. Hier ist das Band jedoch nur in eine Richtung unendlich. Die große Box (entspricht heute ca. dem CPU) besteht aus dem Befehlszähler und dem Akkumulator. Ein Programm ist eine durchnummerierte Folge von Zeilen (Befehlen) und der Befehlszähler startet mit dem Wert 1 und gibt später die aktuelle Zeile an, die ausgeführt wird. Der Akkumulator ist die Stelle, an der gerechnet werden kann.

Der Befehlszähler, der Akkumulator und das Register enthalten natürliche Zahlen. In den ersten Register steht zu Beginn die Eingabe (cf. TM); in den weiteren Registern sowie im Akkumulator 0. Die Anzahl der Register ist unbeschränkt, also können wir beliebige Eingabelängen realisieren. ⋄

Möglich Befehle sind

LOAD i:	$c(0) := c(i);\ b := b+1;$
STORE i:	$c(i) := c(0);\ b := b+1;$
ADD i:	$c(0) := c(0)+c(i);\ b := b+1;$
SUB i:	$c(0) := \max\{c(0)-c(i),0\};\ b := b+1;$
MULT i:	$c(0) := c(0)\cdot c(i);\ b := b+1;$
DIV i:	$c(0) := \lfloor c(0)/c(i) \rfloor;\ b := b+1;$
GO TO j:	$b := j;$
IF $c(0)\,?\,\ell$ GO TO j:	$b := j$ falls $c(0)\,?\,\ell$ wahr;
$(?\in \{=,<,\leq,\geq,>\})$	$b := b+1$ sonst;
END:	$b := b;$

Man beachte, dass nicht alle Subtraktionen erlaubt sind, da die Registereinträge natürliche Zahlen sind.

Weitere Befehlsvarianten sind CLOAD, CADD, CSUB, CMULT, CDIV: ersetzt bei LOAD, ADD, SUB, MULT, DIV auf der rechten Seite $c(i)$ durch i. Genau so gibt es indirekte Befehlsvarianten wie INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT, INDIV: ersetze in LOAD, STORE. ADD, SUB, MULT, DIV $c(i)$ mit $c(c(i))$. Wie bei der Mehrband-TM können diese praktischer weiteren Befehle durch die oben angegebenen simuliert werden.

Bei der Messung der Rechenzeit und des Speicherplatzes unterscheiden wir zwei Kostenmodelle.

Uniformes Kostenmaß. Jeder Rechenschritt benötigt eine Zeiteinheit und der Speicherplatz entspricht der Anzahl genutzter Register.

Jedoch ist das Bandalphabet einer TM endlich, also können wir in jeder Speicherzelle nur endlich viele verschiedene Zustände speichern. Die Register sind mächtiger: da Register beliebig große Zahlen enthalten können, ist das uniforme Kostenmaß in der Praxis oft unrealistisch. Daher ein alternatives Kostenmaß.

Logarithmisches Kostenmaß. Die Zeit für einen Schritt ist proportional zur Länge (in Binärkodierung) der Zahlen in den daran beteiligten Registern und der Speicherplatz eine Registers entspricht der Länge der gespeicherten Zahl.

Im Folgenden beziehen wir uns meist auf das logarithmische Kostenmaß. Das folgende Theorem untermauert die Church'sche These.

Theorem 8.4.1

Jede $t(n)$-zeitbeschränkte Registermaschine kann für ein Polynom q durch eine $O(q(n + t(n)))$-zeitbeschränkte TM simuliert werden. Jede $t(n)$-zeitbeschränkte TM kann durch eine Registermaschine simuliert werden, die uniform $O(n + t(n))$ und logarithmisch $O\left((n + t(n)) \log(n + t(n))\right)$ zeitbeschränkt ist.

8.5 Zurück zur Berechenbarkeit

Definition 8.5.1 (Kanonisch Ordnung auf Σ^*)
Sei Σ ein endliches Alphabet mit linearer Ordnung $\prec$. Für $v, w \in \Sigma^*$ schreiben wir $v \leqslant w$, wenn v kürzer als w ist oder wenn beide gleich lang sind und v bezüglich $\prec$ lexikographisch kleiner ist als w. Die dadurch definiert totale Ordnung auf Σ^* heißt kanonische Ordnung.

kanonische Ordnung

Definition 8.5.2 (Diagonalsprache)
Sei M_i eine TM, deren Gödel-Nummer $\langle M_i \rangle \in \{0,1\}^*$ an der i-ten Stelle der kanonischen Ordnung aller Gödel-Nummern steht. Sei $w_i \in \Sigma^*$ ein Wort an der i-ten Stelle der kanonischen Ordnung von Σ^*. Die Diagonalsprache ist $D := \{w_i : M_i \text{ akzeptiert } w_i \text{ nicht}\}$.

Diagonalsprache

Theorem 8.5.1

Die Diagonalsprache ist nicht rekursiv.

Beweis. Angenommen, es existiert eine TM M, welche D entscheidet. Dann existiert ein $j \in \mathbb{N}$, sodass $M = M_j$ gilt. Ist $w_j \in D$, so akzeptiert M w_j. Aber nach Definition von D wird w_j nicht von M_j akzeptiert, was einen Widerspruch darstellt. Ist $w_j \notin D$, so akzeptiert M w_j nicht. Aber dann müsste $w_j \in D$ sein, was eine Widerspruch darstellt. □

Korollar 8.5.3
Das Komplement der Diagonalsprache $\overline{D} := \Sigma^* \backslash D$ ist nicht rekursiv.

Beweis. Allgemein ist eine Sprache L genau dann rekursiv, wenn $\overline{L}$ rekursiv ist, da man die akzeptierenden und die nicht-akzeptierenden Endzustände der zugehörigen TM vertauschen kann. □

Definition 8.5.4 (Halteproblem)
Das Halteproblem H ist durch die Sprache

$$\{\langle M \rangle w : M \text{ terminiert auf Input } w \text{ nach endlich vielen Schritten}\}$$

definiert.

Halteproblem

Theorem 8.5.2

Das Halteproblem ist nicht rekursiv.

Beweis. Angenommen, es existiert eine TM M, welche H entscheidet. Wir konstruieren nun aus M eine TM M', welche $\overline{D}$ entscheidet. M' entscheidet für $w \in \Sigma^*$ ob $w \in \overline{D}$ wie folgt:

1. Berechne i mit $w = w_i$ und berechne $\langle M_i \rangle$.
2. Wende M auf $\langle M_i \rangle w$ an. Ist $\langle M_i \rangle w \notin H$, so ist $w \notin \overline{D}$.
3. Andernfalls ist $\langle M_i \rangle w \in H$. Simuliert man nun M_i auf der Eingabe w (mit eine universellen TM), so akzeptiert M_i w_i genau dann wenn $w \in \overline{D}$. Dann wäre aber $\overline{D}$ entscheidbar, was ein Widerspruch zu Korollar 8.5.3 ist. □

Definition 8.5.5
Die universelle Sprache ist

$$U := \{\langle M \rangle w : M \text{ akzeptiert den Input } w\}.$$

universelle Sprache

Theorem 8.5.3

Die universelle Sprache ist nicht rekursiv.

Beweis. Angenommen, es existiert eine TM M', welche U entscheidet. Wir konstruieren aus M' ein TM M'', welche $\overline{D}$ entscheidet. M'' entscheidet für $w \in \Sigma^*$ ob $w \in \overline{D}$ wie folgt:

1. Berechne i mit $w = w_i$ und berechne $\langle M_i \rangle$.
2. Wende M' auf $\langle M_i \rangle w$ an und entscheide damit, ob $\langle M_i \rangle w_i \in U$, d.h. ob $w \in \overline{D}$, was ein Widerspruch zu Korollar 8.5.3 ist. □

Theorem 8.5.4

Die universelle Sprache ist rekursiv abzählbar.

Beweis. Wende die universelle TM auf $\langle M \rangle w$ an. Akzeptiert M den Input w, dann stellt man das nach endlich vielen Schritten fest. □

Definition 8.5.6 (Spezielles Halteproblem)
Sie $\varepsilon \in \Sigma^0$ das leere Wort. Das spezielle Halteproblem H_ε ist

$$\{\langle M \rangle : M \text{ terminiert auf Input } \varepsilon \text{ nach endlich vielen Schritten}\}.$$

spezielle Halteproblem

Theorem 8.5.5

Das spezielle Halteproblem ist nicht rekursiv.

Beweis. Angenommen, es existiert ein TM M', welche H_ε entscheidet. Wir konstruieren aus M' ein TM M'', welche H entscheidet. M'' entscheidet für $\langle M \rangle w$ ob $\langle M \rangle w \in H$ wie folgt:

1. Berechne die Gödel-Nummer $\langle M_w^* \rangle$ der DTM M_w^*, die wie folgt arbeitet: Schreibe w auf das Band und simuliere dann M auf Input w.
2. Simuliere M' auf $\langle M_w^* \rangle$.

Dann akzeptiert M'' $\langle M \rangle w$ genau dann, wenn M_w* auf ε terminiert und das passiert genau dann, wenn M auf w terminiert. □

Animationen aus CoMa I

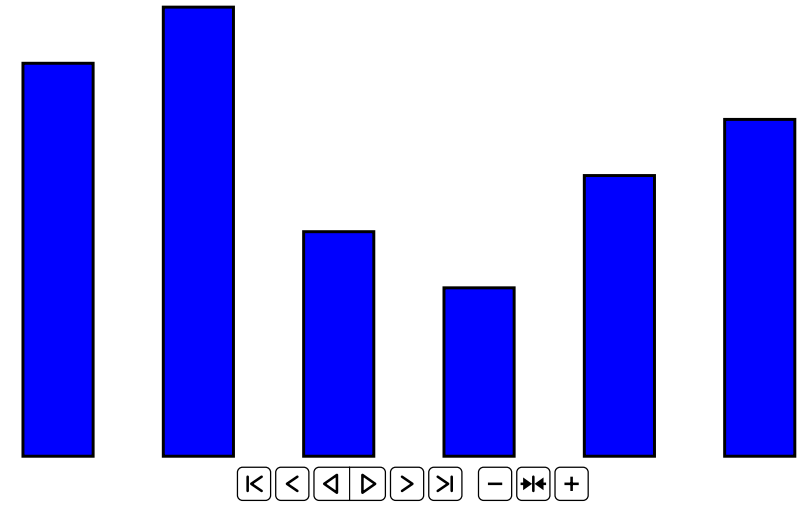


Abbildung 47: Selectionsort

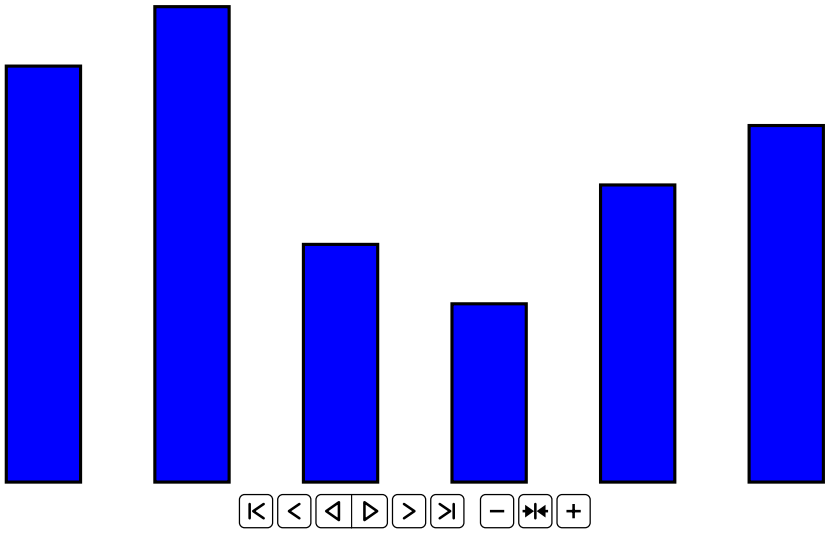


Abbildung 48: Insertionsort

Index

W

Z