ADS Study Guide*

2. Juli 2015

Inhaltsverzeichnis

1 Einführung – Algorithmen **2**

2 Paradigmen **3**
- 2.1 Greedy . . . 3
- 2.2 Divide-and-Conquer . . . 3
- 2.3 Dynamic Programming . . . 3

3 Analyse und Bewertung von Algorithmen **4**
- 3.1 Laufzeitkomplexität – Ordnungsnotation . . . 4
- 3.2 Laufzeitanalyse . . . 5
- 3.3 Beispiele . . . 6

4 Vektoren **8**
- 4.1 Dictionary – Einführung . . . 8
- 4.2 Separate Chaining . . . 9
- 4.3 Double Hashing . . . 10
  - 4.3.1 Linear Probing . . . 11
- 4.4 Linear Hashing . . . 11
- 4.5 Extendible Hashing . . . 12
- 4.6 Cuckoo Hashing . . . 13
- 4.7 Coalesced Hashing . . . 14

5 Sortieralgorithmen **14**
- 5.1 Selection Sort . . . 14
- 5.2 Insertionsort . . . 15
- 5.3 Quicksort . . . 16
- 5.4 Mergesort . . . 17
- 5.5 Heapsort . . . 18
- 5.6 Counting Sort . . . 20
- 5.7 Bucket Sort . . . 21
- 5.8 Radixsort . . . 21

*ADS S2015 (Wanek, Schikuta, Henzinger)

6 Listen 22
6.1 singly-linked list 22
6.2 Stack 24
6.3 Queue 26
6.4 Spezielle Listen 27

7 Bäume 27
7.1 Eigenschaften 27
7.2 Binärer Suchbaum 28
7.3 inorder, preorder, postorder und levelorder Traversierung an einem Beispiel 33
7.4 Laufzeit von DFS-Traversierung beim binären Suchbaum 33
7.5 Mehrwegbäume 34
7.6 2-3-4 Baum 34
7.7 B^+-Baum 35
7.7.1 Eigenschaften 35
7.7.2 Balanzierung 36
7.8 Trie 38
7.9 Priority Queue 38
7.9.1 Heap 39

8 Graphen 41
8.1 Eigenschaften 41
8.2 Topologisches Sortieren 42
8.3 Traversierung 43
8.3.1 Depth-First-Search 43
8.3.2 Breadth-First Search 44
8.3.3 Ein Beispiel traversiert mit BFS und DFS 44
8.4 Dijsktras Algorithmus für kürzeste Wege 47
8.5 Bellman-Ford Algorithmus für kürzeste Wege 49
8.6 Floyd-Warshall Algorithmus für die transitive Hülle 50
8.7 Floyd-Warshall Algorithmus für kürzeste Wege 50
8.8 Minimale Spannende Bäume 50
8.8.1 Algorithmus von Kruskal 51
8.8.2 Algorithmus von Prim 54

A Big-O Tabellen 56
A.1 Datenstrukturen 56
A.2 Sortieralgorithmen 57
A.3 Graphen 57

1 Einführung – Algorithmen

Ein Algorithmus kann auch als ein zielführendes Rezept beschrieben werden – er definiert eine Abfolge von Schritten, die ein gewünschtes Ergebnis liefern. Algorithmen lösen also ein bestimmtes Problem auf eine bestimmte Weise. Leider werden Algorithmen nicht gleich geboren; so haben Bubblesort und Quicksort zwar das selbe

Ziel, nämlich das Sortieren von Werten, klaffen in der Performance jedoch stark auseinander. Ein Ziel der Informatik ist demnach, Algorithmen zu (er)finden, die vordefinierte Probleme effizienter lösen. Um Daten zu organisieren, sodass sie durch Algorithmen verarbeitbar sind, sind Datenstrukturen notwendig, die wiederum auf Effizienz optimiert werden müssen.

2 Paradigmen

2.1 Greedy

In jedem Schritt eines greedy Algorithmus wird die Möglichkeit gewählt, die unmittelbar (in diesem Schritt) den optimalsten Wert bezüglich der Zielfunktion liefert.

Pro: findet oft gute Lösungen, schnell
Contra: manchmal doch nicht die optimalste Lösung

Beispiel: Münzwechselmaschine, Dijkstra, Prim, Kruskal

2.2 Divide-and-Conquer

Bei Divide-and-Conquer-Ansätzen wird das Problem zuerst so lange verkleinert, bis die Subprobleme lösbar werden (oft als trivialer Fall einer Rekursion), darauf werden die Teillösungen wieder vereinigt und bilden so die Lösung des großen Ganzen.

Problem size division
Zerlegung des Problems in eine endliche Anzahl von Teilproblemen

Step division
Aufteilen einer Aufgabe in eine Sequenz von individuellen Aufgaben (bspw. Funktionen)

Case division
Identifikation von Spezialfällen

Beispiel: Mergesort, Quicksort
Bei Quicksort ist der Merge-Schritt (das Zusammenführen von Teillösungen) trivial, bei Mergesort ist es genau umgekehrt – das Teilen ist trivial.

Nicht zu Verwechseln mit Mulitply-and-Surrender.

2.3 Dynamic Programming

Bei Dynamic Programming Ansätzen werden Probleme nicht aufgeteilt, sondern die Lösungen aufgebaut – man startet mit einer Lösung für den trivialen Fall, wie etwa die erste Fibonacci-Zahl, und kombiniert alle darauffolgenden Lösungen aufgrund der vorhergehenden Lösungen. Dies verursacht ggf. einen höheren Speicherbedarf.

3 Analyse und Bewertung von Algorithmen

3.1 Laufzeitkomplexität – Ordnungsnotation

$Big - O - Notation$

Eine Funktion $f(n)$ ist von der Ordnung $O(g(n))$, wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \leq c_0 g(n)$ für alle $n > n_0$ gilt. Big-O liefert eine *Obergrenze*, wie "Ich habe nicht *mehr als* 10 Millionen Euro in meiner Geldbörse."

e.g. $17n^2 \in O(n^2)$, aber auch $17n^2 \in O(n!)$

$Big - \Omega - Notation$

Eine Funktion $f(n)$ ist von der Ordnung $\Omega(g(n))$, wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \geq c_0 g(n)$ für alle $n > n_0$ gilt. Big-Ω liefert eine *Untergrenze*, wie "Ich habe nicht *weniger als* 5 Euro in meiner Geldbörse."

e.g. $n^{37} \in \Omega(n^2)$, aber auch $n^{37} \in \Omega(1)$

$Big - \Theta - Notation$

Eine Funktion $f(n)$ ist von der Ordnung $\Theta(g(n))$, wenn drei Konstanten c_0, c_1 und n_0 existieren, sodass $c_0 g(n) \leq f(n) \leq c_1 g(n)$ für alle $n > n_0$ gilt. Big-Θ liefert also ein Abgrenzung nach oben und nach unten – eine *asymptotisch scharfe Schranke*. Es gilt sowohl $f \in O(g)$ als auch $g \in O(f)$ (und selbiges mit Ω!). Es handelt sich also um eine exakte Beschreibung der Laufzeitkomplexität, ohne viel Interpretationsspielraum.

Zusammenfassend in Sprache:

$f(n) \in O(g(n))$

$f(n)$ wächst asymptotisch nicht schneller als $g(n)$

$f(n) \in \Omega(g(n))$

$f(n)$ wächst asymptotisch nicht langsamer als $g(n)$

$f(n) \in \Theta(g(n))$

$f(n)$ wächst asymptotisch gleich wie $g(n)$, $f(n)$ ist also sowohl $O(g(n))$ als auch $\Omega(g(n))$

Zur Veranschaulichung harte Zahlen (keine Messwerte):

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Abbildung 1: Wachstumsraten von häufig vorkommenden Funktionen [1]

3.2 Laufzeitanalyse

Die mathematische Analyse der Laufzeit ist nicht trivial. Generell überlegt man sich sowohl die Ober- als auch die Untergrenze und leitet daraus ggf. die Θ-Ordnung ab (wenn O und Ω übereinstimmen ist das eindeutig möglich) – man betrachtet also die bestmögliche Laufeit (z.B. bei Quicksort, der im besten Fall von der Ordnung $n \log n$ ist, also $quick(n) \in \Omega(n \log n)$) und die schlechtmöglichste Laufzeit (die bei Quicksort, bei vorsortierten Folgen und tollpatschiger Pivotwahl von der Ordnung n^2 ist, also $quick(n) \in O(n^2)$). In diesem Fall wäre die Bestimmung der durchschnittlichen Laufzeitkomplexität nicht einfach; es heißt aber generell, dass diese Randfälle so selten vorkommen, dass die abartigen Entartungen von Quicksort oft unter den Tisch fallen und er als $O(n \log n)$-Algorithmus gehandelt wird, was bei cleverer Pivotwahl auch zutrifft.

Ignoriere konstante Faktoren

$T(n) = 99n \implies O(n)$

Konstante Faktoren ändern sich bezüglich n nicht direkt (evt. indirekt aufgrund von Cache/Speicher, das wird aber in der theoretischen Form nicht beachtet!)

Merke nur höchste Potenzen

$T(n) = n^3 + n^2 \implies O(n^3)$

Verglichen mit der höchsten Potenz sind andere Potenzen bei großen n vernachlässigbar. Wenn wir zirka 500 Jahre auf das Sortieren einer Eingabe warten müssen, ist es auch schon egal ob es in Wahrheit 532,9 Jahre sind.

Wichtige Rekurrenzen:

$T(n) = T(n-1) + n \Rightarrow T(n) = n\,\frac{n+1}{2} \implies O(n^2)$

$T(n) = T(\frac{n}{2}) + 1 \Rightarrow T(n) = log_2\, n \implies O(log\, n)$

$T(n) = T(\frac{n}{2}) + n \Rightarrow T(n) = 2n \implies O(n)$

$T(n) = 2T(\frac{n}{2}) + n \Rightarrow T(n) = n\, log_2\, n \implies O(n \log n)$

$T(n) = 2T(\frac{n}{2}) + 1 \Rightarrow T(n) = 2n - 1 \implies O(n)$

Master Theorem

$T(n) = aT(\frac{n}{b}) + \Theta(n^c)$; wobei $a \geq 1,\ b > 1,\ c \geq 0$

Unterscheidung in 3 Fälle:

1. wenn $c < log_b\, a$, dann $T(n) \in \Theta(n^{log_b\, a})$
2. wenn $c = log_b\, a$, dann $T(n) \in \Theta(n^c \log n)$
3. wenn $c > log_b\, a$, dann $T(n) \in \Theta(n^c)$

Anders gesagt: $T(n) \in \Theta(n^{max\ \{\log_b a,\ c\}})$. Der Fall, dass $c = \log_b a$, ist ein Sonderfall.

[1] Grafik aus Steven S. Skiena: The Algorithm Design Manual Second Edition, 2008

3.3 Beispiele

```
void g(int n) {
  if(n == 0) return;
  for(int i = 2; i < 3*n; ++i);
  g(n/2);
}
```

Die zugehörige Rekurrenz hat die Form

$$T(n) = T(\frac{n}{2}) + n$$

Mittels Mastermethode können wir mit $a = 1$, $b = 2$ und $c = 1$ die Laufzeitkomplexität als von der Ordnung $\Theta(n)$ bestimmen, da $\log_b a < c$ – damit ist Fall 3 wahr, das Ergebnis ist $\Theta(n^c) \equiv \Theta(n)$. Alternativ kann man sich das Ergebnis als Folge vorstellen:

n	+	n / 2	+	n / 4	+	n / 8	+	n / 16	+	n / 32	+	...	+	1

Abbildung 2: Rekurrenz mit der Lösung $2n$.

$2n$ ist von der Ordnung $\Theta(n)$. Damit gilt die Lösung.

```
void h(int n, int digit = 6) {
  if(n == 0) return;
  for(int i = 0; i < n; ++i)
    for(int i = n; i > 0; i -= 3);
  for(int i = 0; i < digit; ++i) h(n / 3, digit);
}
```

Die äußere Schleife läuft n mal. Die innere Schleife läuft $\lceil \frac{n}{3} \rceil$ mal. Zusammen ergibt das eine Laufzeit von $\Theta(n^2)$. Die Schleife, in der die Rekursion passiert, läuft 6 mal, daher gibt es 6 rekursive Aufrufe pro Aufruf.
Die zugehörige Rekurrenz ist

$$T(n) = 6T(\frac{n}{3}) + n^2$$

mit $a = 6$, $b = 3$ und $c = 2$. Da $\log_b a < c$ ist Fall 3 wahr, wir erhalten das Ergebnis $\Theta(n^c) \equiv \Theta(n^2)$.
(Anmerkung: für *digit* war ein Teil der Matrikelnummer anzugeben, daher konstant.)

> Schreiben Sie **eine** Funktion in C++ mit einem Parameter n (vom Typ `int`), deren Laufzeitkomplexität **gleichzeitig** die Ordnungen $O(n^3)$, $\Omega(n)$ und $\Theta(n^2)$ hat.

Die wirklich wichtige Information ist die Laufzeitkomplexität in Θ-Notation.

```
void quadratischer_kuckkuck(int n) {
  for(size_t i = 0; i < n; ++)
    for(size_t j = n; j; --j)
      std::cout << "kuckkuck!\n";
}
```

$O(n^3)$ legt eine Oberschranke fest, unsere Funktion darf also nicht schneller als eine Funktion der Ordnung $\Theta(n^3)$ wachsen. Das tut sie nicht.

$\Omega(n)$ legt eine Unterschranke fest, unsere Funktion darf also nicht langsamer als eine Funktion der Ordnung $\Theta(n)$ wachsen. Das tut sie nicht.

$\Theta(n^2)$ legt eine asymptotisch scharfe Schranke fest, unsere Funktion wächst also etwa gleich wie eine andere Funktion der Ordnung $\Theta(n^2)$. Rein zufällig gilt das für unsere Funktion, denn wir haben zwei ineinander geschachtelte Schleifen, die beide n-mal laufen und eine konstante Operation ausführen.

Demnach sind alle drei Ordnungen erfüllt. Juhu!

```
1  int x = 19;
2  int y = 34;
3  int z = 18;
4
5  void g(int i, int n) {
6    if(i > 1) {
7      for(int j = 0; j > n; j += 2)   // wird nie wahr
8        g(i - 1, n);
9    }
10 }
11
12 void f(int n) {
13   if(!n) return;
14   g(x % 5, n);   // 4
15   f(n / (z % 10 + 2));   // 10
16   g(x % 5, n);   // 4
17   for(int i = 0; i < y % 10; i += 2)   // 4
18     f(n / (z % 10 + 2));   // 10
19   g(x % 5, n);   // 4
20 }
```

Funktion g ist ein Blindgänger, da $n < 0$ nie wahr werden kann. Damit ist $g \in \Theta(1)$. Die Schleife in f läuft 2-mal plus 1 mal außerhalb der Schleife macht das 3 rekursive Aufrufe mit einer verminderten Problemgröße von $\frac{n}{10}$ jeweils. Die Rekurrenz ist somit

$$T(n) = 3T(\frac{n}{10}) + \Theta(n^0)$$

Die Mastermethode liefert uns für $a = 3$, $b = 10$ und $c = 0$ mit $\log_b a > c$ den Fall 1, also ist $f(n) \in \Theta(n^{\log 3})$ oder $f(n) \in \Theta(3^{\log n})$.

```
1  int x = 19;
2  int y = 34;
3  int z = 18;
4
5  void g(int i, int n) {
6    if(i > 1) {
7      for(int j = 0; j < n; j += 2)
8        g(i - 1, n);
9    }
10 }
11
12 void f(int n) {
13   if(!n) return;
```

```
14   g(x % 5, n);   // 4
15   f(n / (z % 10 + 2));   // 10
16   g(x % 5, n);   // 4
17   for(int i = 0; i < y % 10; i += 2)   // 4
18     f(n / (z % 10 + 2));   // 10
19   g(x % 5, n);   // 4
20 }
```

Die Schleife läuft jeweils $\lceil \frac{n}{2} \rceil$. In dieser Version ist $g \in \Theta(n^3)$, da die Rekursion 3-mal läuft und das 3 ineinander geschachtelten Schleifen entspricht. Dies muss durch Rückwärtseinsetzen bewiesen werden:

$$T(n) = n * T(n) = \ldots = n * n * n * 1$$

Die Rekurrenz ist daher

$$T(n) = 3T(\frac{n}{10}) + \Theta(n^3)$$

Damit ist f, da $c = 3 > \log 3$, von der Ordnung $\Theta(n^3)$.

> Schreiben Sie **eine rekursive** Funktion in C++ mit einem Parameter n (vom Typ `int`), deren Laufzeitkomplexität **gleichzeitig** die Ordnungen $O(n^3)$, $\Omega(n)$ und $\Theta(n^2)$ hat. Zeigen Sie mithilfe des Mastertheorems, dass die Laufzeitkomplexität Ihrer Funktion die gewünschten Ordnungen hat.

```
1 void f(int n) {
2   if(!n) return;
3   std::cout << "hallo!\n";
4   for(int i = 0; i < 4; ++i) f(n/2);
5 }
```

Die dazugehörige Rekurrenz sieht so aus:

$$T(n) = 4T(\frac{n}{2}) + \Theta(n^0)$$

Wir erhalten also $a = 4$, $b = 2$ und $c = 0$. Damit gilt Fall 1, daher $f \in \Theta(n^2)$. Aufgrund der Definition der O-Notation als Obergrenze und der Ω-Notation als Untergrenze gilt auch $f \in O(n^3)$ und $f \in \Omega(n)$, da unsere Funktion nicht schneller als n^3 und nicht langsamer als n wächst.

4 Vektoren

4.1 Dictionary – Einführung

Dictionaries – zu Deutsch *assoziative Datenfelder* – sind Datenstrukturen, die *Schlüssel* (meistens Zeichenketten) verwenden, um enthaltene Elemente zu adressieren (*key-value pairs*). Dies ist effizient, da statt linearem Suchen eine Transformationsfunktion verwendet wird, die in $O(1)$ berechnet werden kann und (im Idealfall) die eindeutige Position des Wertes liefert. Dictionaries sind in so gut wie allen Programmiersprachen eingebaut in unterschiedlichen Varianten vorhanden. Python beispielsweise verwendet eine dynamisch wachsende Version von Double Hashing mit Randomized Probing[2],

[2] svn.python.org/projects/python/tags/r32b1/Objects/dictobject.c

Ruby verwendet Separate Chaining, wobei das Datenfeld vergrößert wird, sobald die Wertedichte pro Bucket 5 erreicht[3]. C++ verwendet zumeist Bäume, ist implementationsabhängig.
Alle Hashtabellen funktionieren irgendwie gleich – es gibt eine Hashfunktion, die Werten eine Position zuordnet. Was den Unterschied ausmacht, ist die Resolvierung einer Hashkollision, also was passiert, wenn zwei (oder mehr) Werte an die selbe Position gehasht werden.

4.2 Separate Chaining

Separate Chaining resolviert Kollisionen dadurch, dass die Werte einfach eingehängt werden. Dies ist durch verkettete Listen realisiert, die arbiträr lang werden können. Separate Chaining Hashtabellen können also ein Vielfaches ihrer Größe an Werten aufnehmen, was aber mit extremen Performanceeinbußen einhergeht.

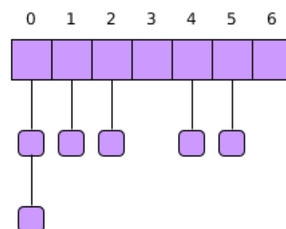

Abbildung 3: Separate Chaining Hashtabelle der Größe 7 mit 6 Werten und einer Kollision auf 0

Üblicherweise ist Separate Chaining ein *statisches* Hashverfahren, der Datenbereich wird also nicht nach Bedarf vergrößert. In einigen high-level Sprachen (und auch in der high-level Übung) wird jedoch eine modifizierte Version von Separate Chaining eingesetzt, die nach erreichen einer bestimmten Auslastung (oder nach dem Erreichen einer gewissen Wertedichte, mit welcher man beispielsweise Clustering erkennen kann) alle Werte in einen vergrößerten Datenbereich rehasht. Wichtig: Die Tabellengröße muss und soll nicht prim sein. Primzahlen suchen ist ein aufwändiges Hobby und stört daher die Performance. Eine einfache Struktur könnte so aussehen:

```
template < class T, size_t N = 11 >
class sep_chaining {
  private:
    size_t table_size{ N };
    linked_list< T >* table{ new linked_list< T >[table_size] };
    size_t entries{ 0 };
    double max_factor{ 0.666 }; // float of the beast
  public:
    ...
};
```

Einfügen, Entfernen und Suchen wird alles über die Liste an der entsprechenden Position realisiert. Implementiert man für diese einen Iterator hat man eine wunderschöne, für ihre Einfachheit sehr effiziente, Hashtabelle.

[3]patshaughnessy.net/Ruby-Under-a-Microscope-Rough-Draft-May.pdf

4.3 Double Hashing

Double Hashing verwendet eine zweite Hashfunktion, die bei Kollision zur ursprünglich errechneten Position hinzuaddiert wird. Dies wird wiederholt bis eine freie Stelle gefunden wird, oder man an die ursprüngliche Position gelangt – was bedeutet, dass der Wert nicht eingefügt werden kann. Ist die Tabellengröße eine Primzahl wird jeder Slot betrachtet, bevor man an den ursprünglichen Slot gelangt.

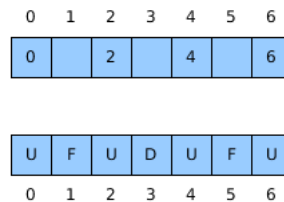

Abbildung 4: Double Hashing Hashtabelle der Größe 7, mit Statusfeld

Im Gegensatz zu Separate Chaining werden die Werte direkt in der Tabelle gespeichert, daher können keine Annahmen über den Inhalt per se getroffen werden, sondern es muss eine Statustabelle angelegt werden, die genaue Auskunft über den Zustand des Feldes gibt. Wenn ein Wert freigegeben wird, also entfernt, wird der Status an dieser Position auf `deleted` gesetzt, sodass der Kollisionspfad, der beim Suchen anderer Elemente verfolgt wird, nicht gestört wird. Würde man dies nicht tun und den Status auf `free` setzten, würde die Hashtabelle fälschlicherweise vermuten, dass ein Wert nicht vorhanden wäre, da er sonst an die erste freie Stelle geschrieben worden wäre – diese freie Stelle war zum Zeitpunkt des Einfügens aber nicht frei!

Eine einfache Struktur könnte so aussehen:

```
template < class T, size_t N = 11 >
class double_hash {
  private:
    size_t table_size{ next_prime(N) };
    enum class Status : char { free, occupado, deleted }* status{ new Status[table_size]() };
    T* table{ new T[table_size]() };
    double max_factor{ 0.666 };
  public:
    ...
};
```

Bei Double Hashing als Kollisionsresolutionsverfahren ist eine Primzahl als Tabellengröße nicht notwendig, aber sehr empfehlenswert. Die Theorie dahinter ist die relative Primheit zwischen Schrittgröße (die durch die zweite „Hashfunktion“ gegeben ist) und Tabellengröße. Besitzen Schrittlänge und Tabellengröße einen gemeinsamen Teiler, wie etwa 2, werden beim Verfolgen des Kollisionspfades Indize ausgelassen. Dies kann zu falschen Annahmen bezüglich des Status führen – die Hashtabelle würde hierbei melden, dass sie voll beziehungsweise ein Einfügen unmöglich ist, weil ein Zyklus erkannt wurde. In Wahrheit könnte die Hashtabelle aber auch nur 50% oder weniger gefüllt sein! Es existieren komplexere Algorithmen, wie Randomized Probing, die dieses Problem beheben. Primzahlen suchen ist aufwändig!

4.3.1 Linear Probing

Linear Probing ist ein Sonderfall von Double Hashing, wobei die zweite Funktion $g(x) = 1$ ist; es wird also linear der nächste freie Platz gesucht. Hier entfällt auch der Zwang zu Primzahlen, da jedenfalls alle Indize besucht werden.

4.4 Linear Hashing

Das Ziel von Linear Hashing ist lineares Wachstum. Dies wird dadurch erreicht, dass die Hashtabelle pro Durchlauf um einen Bucket wächst. Überläufe werden durch Überlaufbuckets gehandhabt. Die Hashfunktionen ändern sich nach jedem vollständigen Durchlauf.

Zu Beginn besteht die Hashtabelle aus m Buckets (nummeriert 0 bis $m-1$) und einer Hashfunktion $h_0(x) = f(x) \% m$. Sobald der erste Überlauf passiert, wird der Bucket an der Position 0 gespalten. An den überlaufenden Bucket (Der keinesfalls der Bucket sein muss, der gerade zur Spaltung ansteht! Die Spaltung erfolgt strikt deterministisch!) wird ein Überlaufbucket angehängt, um den Überlauf unterzubringen. Weiters wird die Tabelle um einen Bucket (Nummer m) erweitert und die Werte aus dem Bucket 0 gerehasht unter Verwendung der zweiten Hashfunktion $h_1(x) = f(x) \% 2m$. Beim nächsten Überlauf wird Bucket 1 gespalten und ein Bucket $m + 1$ hinzugefügt, usw. usf.
Nach einer vollständigen Runde (nachdem Bucket $m - 1$ gespalten wurde) beginnt die nächste Runde. `p`, das bisher den nächst zu spaltenden Bucket angab, wird auf 0 zurückgesetzt, die originale Hashfunktion h_0 wird verworfen und durch h_1 ersetzt. Die neue zweite Hashfunktion ist $h_2(x) = f(x) \% 2^2m$. Zu jedem Zeitpunkt verwendet Linear Hashing zwei Hashfunktionen: h_i und h_{i+1}, wobei $h_i(x) = f(x) \% 2^im$.

Zu jedem Zeitpunkt verfügt Linear Hashing über folgende Komponenten:

- einen Wert `i`, der die derzeitige Rundennummer angibt
- einen Wert `p` $\in [0 \dots 2^im)$, der den als nächstes zu spaltenden Bucket angibt
- ein Total von $2^im + p$ Buckets, die jeweils wieder Overflow-Buckets haben können.

Beim Suchen eines Wertes wird der korrekte Bucket folgendermaßen bestimmt:

$$pos(x) = \begin{cases} h_i(x) & \text{wenn } h_i(x) \geq p \\ h_{i+1}(x) & \text{sonst} \end{cases}$$

Wenn die berechnete Positon größer oder gleich der als nächstes zu spaltenden Position ist, wurde dieser Bucket in der momentanen Runde noch nicht gespalten, ergo muss die zweite Hashfunktion nicht berechnet werden. Anderfalls muss die zweite Funktion hinzugezogen werden, da der Wert nach einem Split entweder im Bucket j oder im Bucket $j + 2^im$ liegt. Folgende zwei Abbildungen zeigen die lineare Vergrößerung und den Rehash-Vorgang.

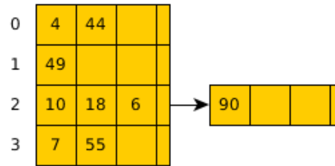

Abbildung 5: Überlauf im Bucket 2, Bucket 0 wird gespalten.

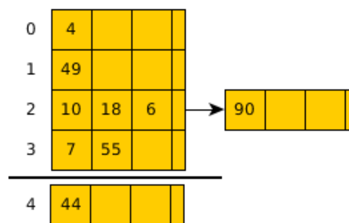

Abbildung 6: Nach der Spaltung. Overflow ist nach wie vor vorhanden und wird erst im übernächsten Splitting „behoben". Der Balken soll nur neu dazu gekommene Buckets markieren. `p` ist nun 1.

Sollte der Wert 44 nun gesucht werden, so würde $h_0(44) = 0 < p$ gelten, damit würde h_1 aufgerufen werden, die auch die korrekte Position 4 liefert.
Da bei jedem auftretenden Überlauf ein Splitting und die verschwenderische Vergrößerung um nur ein Bucket ein etwas dezenter Overkill ist, ist eine gute mögliche Optimierung von Linear Hashing die Verwendung von der Formel von Salzberg, die besagt, dass erst nachdem $L * b$ Datensätze eingefügt wurden ein Splitting erfolgen soll. b ist die Blockgröße und $L = \frac{Rohdatenvolumen}{Volumen\ der\ Datenstruktur}$, ist also der Belegungsfaktor. Weiters können bereits größere Felder allokiert werden statt einer graduellen Vergrößerung von einem Bucket pro Split und/oder Pointer auf die Buckets verwendet werden, da das Kopieren von Pointern trivial ist.

4.5 Extendible Hashing

Bei Extendible Hashing wird versucht, den Rehash-Aufwand möglichst gering zu halten. Um dies zu erreichen, wird ein Index aus Pointern auf Buckets, die eine vorgegebene Größe haben, verwendet. Der Index verfügt über eine globale Tiefe, die die Größe festlegt. Jeder Bucket verfügt über eine lokale Tiefe.
Die globale Tiefe gibt außerdem (vordergründig) Auskunft über die Anzahl an Bits, die von dem von der Hashfunktion berechneten Wert gerade verwendet werden. Sollte die globale Tiefe gerade 3 sein, so werden nur die (in dieser Variante) letzten 3 Bits betrachtet. Die lokale Tiefe der Buckets gibt ebenfalls Auskunft über die Anzahl der Bits, die bei den in dem Bucket enthaltenen Werten übereinstimmen. Es zeigen genau 2^{gd-ld} Pointer auf einen Bucket – dies hat zur Folge, dass die Werte $3 \equiv 0011$ und $7 \equiv 0111$ in einem Bucket der lokalen Tiefe 2 (die letzten 2 Bits stimmen überein!) zusammen wohnen. Sollte die globale Tiefe 4 sein, so zeigen $2^{4-2} = 4$ Pointer auf diesen, das sind alle Einträge, die in den letzten 2 Bits 11 sind. Folgendes Bild illustriert einen gültigen Zustand.

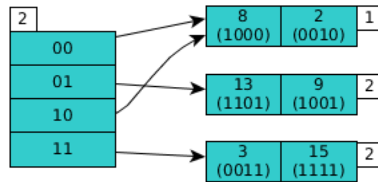

Abbildung 7: Extendible Hashing mit globaler Tiefe 2, zwei Buckets mit lokaler Tiefe 2 und einem Bucket mit lokaler Tiefe 1

Sollte ein Bucket überlaufen, werden zwei Fälle unterschieden:

1. Die lokale Tiefe des Buckets ist kleiner der globalen Tiefe: das bedeutet, dass 2^n mit $n \geq 1$ Pointer auf diesen Bucket zeigen. In diesem Fall wird der Bucket *gespalten* – alle Werte werden entnommen, ein zweiter Bucket alloziiert und die Hälfte aller Pointer (all jene, die sich im dazugekommenen Bit unterscheiden) werden auf den neuen Bucket gesetzt. Danach werden alle Werte rekursiv wieder hinzugefügt und darauf gehofft, dass der Überlauf damit erledigt ist. Die lokale Tiefe beider Buckets hat sich um 1 erhöht.

2. Sollte die lokale Tiefe des Buckets gleich der globalen Tiefe sein, so reichen die durch die globale Tiefe gegebene Anzahl an Bits nicht aus, um alle Werte in zugehörige Buckets unterzubringen. Der Index wird verdoppelt – eine relativ billige Operation, da nur Pointer zweifach kopiert werden müssen. Die globale Tiefe wird erhöht. Damit zeigen auf den betroffenen Bucket nun zwei Pointer und er kann gespalten werden.

Extendible Hashing leidet ggf. unter dem exponentiellen Wachstum des Index, garantiert aber, sollte der Index in den Hauptspeicher passen, immer nur einen Index $\rightarrow$ Bucket-Zugriff pro Anfrage, da keine Überlaufbereiche verwendet werden. Weiters wäre in dieser Version das Vorhandensein von Duplikaten fatal – sind beispielsweise 6 Duplikate vorhanden, die Bucketgröße aber nur 4, würde die Hashtabelle eine Spaltungs-Endschlossschleife durchlaufen. In Realität werden auch bei Extendible Hashing Überlaufbereiche angelegt, wenn die Anzahl an Spaltungen beziehungsweise Verdoppelungen eine gewisse Schranke überschreiten.

4.6 Cuckoo Hashing

Komisches Ding. Es sind zwei Tabellen vorhanden (die gleich groß sein können) und es werden zwei Hashfunktionen verwendet. Beim Einfügen wird zuerst die erste Tabelle mit der ersten Hashfunktion betrachtet – sollte die berechnete Position besetzt sein, wird der einzufügende Wert mit dem dort stehenden Wert vertauscht und versucht, dieses Element in die zweite Tabelle (unter Verwendung der zweiten Hashfunktion) einzufügen. Falls dort Platz ist, wird das Element einfach dort hingeschrieben, ansonsten wird wieder getauscht und das Prozedere wiederholt. Ein Zyklus ließe sich erkennen, wenn das ursprüngliche Element (also das Element, mit dem die `add`-Methode aufgerufen wurde) zum dritten Mal auftaucht, in der Praxis gibt es aber eine Obergrenze an Schleifendurchläufen. Gäbe es diese nicht wäre hier ggf. eine Endlosschleife! Sollte

diese Obergrenze erreicht werden, werden zufällig zwei neue Hashfunktionen gewählt und alle Elemente gerehasht. Speicherschonend lässt sich das durchführen, indem man beide Tabellen durchläuft und alle Werte, die falsch stehen, entfernt und neu hinzufügt.

Für Cuckoo wird eine fast universelle Hashfunktionenfamilie verwendet, die gegeben ist durch

$$h_a(x) = (a * x \ \% \ 2^w) \ div \ 2^{w-q}$$

wobei a eine zufällig gewählte Zahl $0 < a < 2^w$, w gleich der Breite des verwendeten Datentyps (32 oder 64 Bit) und q der Exponent der momentanen Tabellengröße als Zweierpotenz ist. Die Modulo-Operation ist nicht notwendig, da die maximale Breite eines Zahlentyps ohnehin nicht überschritten werden kann.

4.7 Coalesced Hashing

Coalesced Hashing ist das uneheliche Kind von Double Hashing und Separate Chaining. Anstatt bei Kollision immer neuen Speicher anzufordern (Sep. Ch.), werden innerhalb der Tabelle Ketten gebildet. Dies kann einerseits durch Pointer oder auch durch Markierung mit speziellen Werten passieren.
Sollte beim Einfügen eine Kollision auftreten, wird der Wert in die letzte freie Position eingetragen und die Position in die Liste, die von der ursprünglich berechneten Position ausgeht, eingehängt.

Als Optimierung bietet sich die Verwendung eines Kellerspeichers an. Dabei wird das Array größer dimensioniert als nötig und der überschüssige Platz als Keller verwendet. Weiters empfiehlt sich die Verwendung eines Zeigers, der auf die nächste freie Position in der Tabelle zeigt.

Soll ein Element gelöscht werden, das nicht Teil einer Kette ist, so ist dies ohne weitere Umstände möglich. Sollte sich das Element im Keller befinden und Teil einer Kette sein, so können die Pointer einfach umgeleitet werden, da der Keller nicht von der Hashfunktion abgedeckt wird, also niemals ein Wert an eine Kellerposition gehasht werden kann. Andernfalls muss das nächste Element der Kette gefunden werden und an die Position des zu löschenden Elementes vorgezogen werden, ohne die Kette dabei zu verändern. Die soeben frei gewordene Position muss dann auf die selbe Art behandelt werden, d.h. der Löschvorgang propagiert sich ggf. durch die gesamte Kette.

5 Sortieralgorithmen

5.1 Selection Sort

```
1 template <class T>
2 void selectionsort(std::vector<T>& vec) {
3   size_t left = 0;
4   do {
5     T min = left;
6     for(size_t i = left + 1; i < n; i++)
7       if(vec[i] < vec[min])
```

```
8          min = i;
9      std::swap(vec[min], vec[left]);
10     left++;
11   } while(left < vec.size());
12 }
```

Selection Sort, oder Sortieren durch Minimumsuche, findet zuerst das kleinste Element, und vertauscht es dann mit dem ersten Wert, womit es effektiv an die richtige Stelle befördert wird. Danach sucht er das nächst kleinste und vertauscht es mit dem zweiten Wert, etc.
Die Rekurrenz lässt sich so beschreiben:

$$T(n) = T(n-1) + n = \frac{n(n-1)}{2}$$

Die Lösung dieser Rekurrenz ist $\Theta(n^2)$.

5.2 Insertionsort

```
1  template <class T>
2  void insertionsort(std::vector<T>& vec) {
3    for(size_t i = 1; i < vec.size(); i++) {
4      T current = vec[i];
5      size_t insertion_point = i;
6      while(insertion_point && vec[insertion_point - 1] > current) {
7        vec[insertion_point] = vec[insertion_point - 1];
8        insertion_point--;
9      }
10     vec[insertion_point] = current;
11   }
12 }
```

Insertionsort nimmt ein Element, findet seinen rechtmäßigen Platz und platziert es dort. Diese Operation beinhaltet das Verschieben von n Elementen, um Platz für das derzeitige Element zu schaffen. Im Gegensatz zu Selection Sort muss sich Insertionsort nicht alle Werte anschauen, sondern bricht ab, sobald der richtige Platz gefunden wurde. Im besten Fall ist Insertionsort von der Ordnung $O(n)$, da jedes Element nur mit seinem Nachbarn verglichen wird.
Insertionsort wird oft als Hilfssortieralgorithmus verwendet, wie etwa bei Bucketsort, wo die Elemente nach dem Ablauf des Bucketsortalgorithmus bereits grob vorsortiert sind und sich maximal k Positionen von ihrer endgültigen Position entfernt befinden. Insertionsort würde also höchstens k mal iterieren, wäre also von der Komplexität $O(nk)$ und somit linear.

Es existiert eine weitere Variante dieses Algorithmus, namentlich Shell Sort. Shell Sort nutzt die gerade erwähnte Eigenschaft Insertionsorts in dem es über größere Distanzen (die je nach Arraygröße selten iterieren) vorsortiert. Die Distanzen werden immer kleiner, ehe sie 1 werden – bei Distanz 1 ist Shell Sort identisch zu Insertionsort. Da das Array bereits grob vorsortiert ist, ist der letzte Durchlauf von linearer Komplexität.

```
1  template <class T>
2  void shellsort(std::vector<T>& vec) {
3    /* Ciura Gap-Sequence [1..701] erweitert via g(k) = 2.25*g(k - 1) */
```

```
4   auto gap_seq = { 5243285, 2330349, 1035711, 460316, 204585, 90927, 40412,
5                    17961, 7983, 3548, 1577, 701, 301, 132, 57, 23, 10, 4, 1 };
6   for(auto const& gap : gap_seq) {
7     for(size_t i = gap; i < vec.size(); i++) {
8       T current = vec[i];
9       size_t j;
10      for(j = i; j >= gap && vec[j - gap] > current; j -= gap)
11        vec[j] = vec[j - gap];
12      vec[j]= current;
13    }
14  }
15 }
```

5.3 Quicksort

```
1  template <class T> // von den folien
2  void quicksort(std::vector<T>& a, int const l, int const r) {
3    int i, j;
4    T pivot;
5    if(r > l) {
6      pivot = a[r]; i = l - 1; j = r;
7      while("juhu schleife") {
8        while(a[++i] < pivot);
9        while(a[--j] > pivot) if(j == l) break;
10       if(i >= j) break; // verlaesst aeussere schleife
11       std::swap(a[i], a[j]);
12     }
13     std::swap(a[i], a[r]); // pivot steht nun korrekt
14     quicksort(a, l, i - 1);
15     quicksort(a, i + 1, r);
16   }
17 }
```

Quicksort ist, wie auch Mergesort, Vertreter der *Divide-and-Conquer*-Methode, kann aber nicht so schön graphisch dargestellt werden, da sein Ablauf von der Struktur der Eingabedaten abhängt. Generell wählt Quicksort zu Beginn eines Aufrufes ein *Pivotelement*, das Basis für die darauf folgenden Verlgeiche ist. In unserem Beispiel muss das rechteste Element als Pivot hinhalten, es könnte aber auch der Median oder ein aus 3 zufällig gewählten Werten berechneter Mittelwert sein. Links vom Pivot sollen nur Elemente stehen, die kleiner als der Pivot sind. Rechts vom Pivot sollen nur Elemente stehen, die größer gleich dem Pivot sind. Quicksort läuft nun also von links das gerade betrachtete Subarray durch und prüft für jedes Element, ob die obige Bedingung gilt. Wenn nicht, bricht die `while`-Schleife dort ab und lässt den Index dorst stehen. Dann beginnt Quicksort von rechts und überprüft wiederum, ob jedes betrachtete Element größer dem Pivot ist. Wenn nein, bricht er dort ab. Sobald beide Schleifen beendet wurden, wird geprüft ob sich die Indize nicht gegenseitig überlaufen haben. Sollte dies nicht der Fall sein, werden die zwei Elemente getauscht, die Bedingung ist somit für diese Werte wahr, und die Schleifen beginnen wieder zu laufen. Sollte dies der Fall sein, wird die äußerste `while`-Schleife verlassen und der Pivot wird mit dem durch `i` (`i` kommt von links) markierten Wert vertauscht. Der Pivot steht somit an seiner endgültigen Position.

Das alles wird für die durch `i` markierten Subarrays rekursiv wiederholt, bis schließlich der triviale Fall, dass Subarrays der Größe 1 sortiert sind, eintrifft und sich die

Rekursion beendet. Quicksort hat eine mittlere Laufzeit von $\Theta(n \log n)$, kann aber unter ungünstigen Umständen (umgekehrt sortierte Eingabe, unglückliche Pivotwahl) zu $O(n^2)$ entarten. Diese Fälle sind in der Praxis aber sehr rar.

Hier in 6 Zeilen, dafür suboptimal in der Geschwindigkeit und vorallem nicht mehr in-place!

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  let smaller_or_equal = [a | a <- xs, a <= x]
      larger = [a | a <- xs, a > x]
  in quicksort smaller_or_equal ++ [x] ++ quicksort larger
```

Wir machen zwei Listen, `smaller_or_equal` und `larger`, die jeweils Elemente kleinergleich beziehungsweise größer dem Pivotelement, das mit dem Ausdruck `(x:xs)` als erstes Element der Liste gewählt wird, enthalten. Wir wiederholen dies für alle Subarrays, bis wir den Base-Case der leeren Liste erreichen. Danach klatschen wir den Pivot zwischen die Teillisten und sind fertig.

5.4 Mergesort

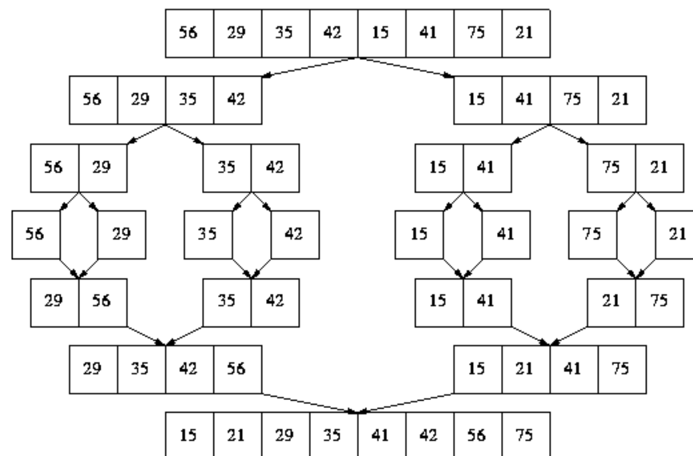

Abbildung 8: Mergesort graphisch dargestellt

Mergesort lässt sich am besten graphisch verstehen. Der Vollständigkeit halber aber hier auch Sourcecode, der von den Folien übernommen wurde.

```cpp
template <class T> // von den folien
void mergesort(std::vector<T>& vec, std::vector<T>& aux, size_t left,
     size_t right) {
  if(right > left) {
    size_t i, j, k, m;
    m = (right + left)/2;
    mergesort(vec, aux, left, m);
    mergesort(vec, aux, m+1, right);

    for(i = m + 1; i > left; i--)
      aux[i-1] = vec[i-1];
```

```
12      for(j = m; j < right; j++)
13        aux[right+m-j] = vec[j+1];
14      for(k = left; k <= right; k++)
15        vec[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
16    }
17  }
18
19  template <class T>
20  void mergesort(std::vector<T>& vec) {
21    std::vector<T> aux(vec.size());
22    mergesort(vec, aux, 0, vec.size() - 1);
23  }
```

Mergesort halbiert das Array stur so lange, bis es Größe 1 erreicht. Dann werden die Subarrays unter Rücksicht der Reihenfolge wieder miteinander verschmolzen. Das Ergebnis ist ein sortiertes Array.
Mergesort ist immer von der Komplexität $\Theta(n \log n)$. Die Eingabedaten sind für Mergesort irrelevant. Mergesort operiert exsitu, hat also eine Speicherkomplexität von $O(n)$ und einen höheren Overhead als Quicksort, entartet dafür aber nie.

5.5 Heapsort

Für die allgemeine Beschreibung eines Heaps siehe 7.9.1 Heap (Seite 39).

```
1   template <class T> // ueberprueft heap-bedingung und stellt diese ggf her
2   void shift_right(std::vector<T>& vec, size_t low, size_t high) {
3     size_t root = low;
4     while((root*2)+1 <= high) {
5       size_t left_child = (root * 2) + 1;
6       size_t right_child = left_child + 1;
7       size_t swap_index = root;
8       if(vec[swap_index] < vec[left_child])
9         swap_index = left_child;
10      if(right_child <= high && vec[swap_index] < vec[right_child])
11        swap_index = right_child;
12      if(swap_index != root) {
13        std::swap(vec[root], vec[swap_index]);
14        root = swap_index;
15      } else break;
16    }
17  }
18
19  template <class T> // baut heap
20  void heapify(std::vector<T>& vec, size_t low, size_t high) {
21    int mid_index = (high - low - 1)/2;
22    while(mid_index >= 0) {
23      shift_right(vec, mid_index, high);
24      mid_index--;
25    }
26  }
27
28  template <class T>
29  void heapsort(std::vector<T>& vec) {
30    size_t high = vec.size() - 1;
31    heapify(vec, 0, high);
32    while(high) {
33      std::swap(vec[0], vec[high--]);
```

```
34      shift_right(vec, 0, high);
35    }
36  }
```

`heapify` arrangiert zuerst die Elemente im Input so, dass die Max-Heap Bedingung erfüllt ist. Danach wird das Wurzelelement immer "entfernt" und an die letzte Stelle geschrieben. `high` wird dekrementiert und die Heap-Bedingung wieder hergestellt.

Diese Prozedur entspricht dem simplen wiederholtem Entfernen aus einem Max-Heap, wodurch natürlich das entstehende Array sortiert ist.
Da jede Heap-Operation (hier also nur das Entfernen und Wiederherstellen der Heap-Bedingung) von der Ordnung $O(\log n)$ ist und dies für alle n Elemente getan werden muss, ist Heapsort von der Ordnung $\Theta(n \log n)$. Heapsort kann nicht entarten, ist aber zu modernen Cache-Architekturen nicht so freundlich wie Quicksort oder Mergesort, von daher immer etwas langsamer. Heapsort ist nicht stabil, da jegliche Information über die ursprüngliche Ordnung der Elemente in der `heapify`-Phase ggf. zerstört wurde.
Um die Vorgehensweise von Heapsort zu verstehen betrachten wir folgedes Minimalbeispiel.

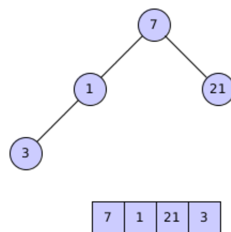

Abbildung 9: Eingabearray in Feld und Baumdarstellung

Nun wird `heapify` aufgerufen, das die Elemente so arrangiert, dass die Max-Heap-Bedingung gilt. Diese ist für die Blätter des Baumes bereits trivial erfüllt.

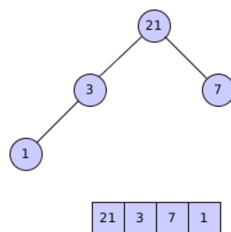

Abbildung 10: Nach dem Ausführen von `heapify`. Wir können nun damit beginnen, das Wurzelelement fortlaufend zu entfernen.

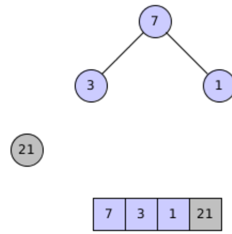

Abbildung 11: Nach dem Entfernen des Wurzelements, also dem Vertauschen mit dem letzten Elements des Heaps. Danach Wiederherstellung der Heap-Bedingung.

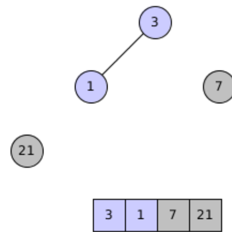

Abbildung 12: Wiederholtes Entfernen, Tauschen und Wiederherstellen.

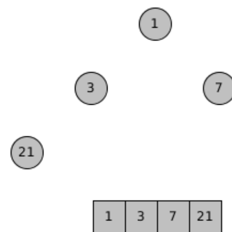

Abbildung 13: Fertig.

5.6 Counting Sort

Counting Sort zählt zuerst in einem Hilfsarray wieviele Elemente von einem bestimmten Wert vorhanden sind. Dann wird die Anzahl des Wertes `i` mit der Anzahl des Wertes `i-1` addiert und somit ermittelt, an welche Stelle der Wert i stehen muss. Schließlich wird das Originalarray von hinten durchgegangen und die Werte an die richtige Position geschrieben. Dabei ist wichtig, dass die Anzahl im Zähl-Hilfsarray dekrementiert wird, da möglicherweise Duplikate im Array vorhanden sind. Counting Sort ist nur für kleine n gebräuchlich, da der Speicheraufwand unnötig groß ist (möchte man etwa das Array `[1 000 001, 5]` via Counting Sort sortieren, so muss ein Hilfsvektor der Größe 1 000 002 allokiert werden). Weiters können negative Werte mit der vanilla Version nicht sortiert werden, da es keine negativen Indize gibt.

5.7 Bucket Sort

Bucketsort ist mit Counting Sort verwandt. Die Werte werden hierbei aufgrund von einigen berechneten Statistiken (dafür benötigt man den maximalen und minimalen Wert) in zugehörige Buckets verteilt, die dann mit einem für kleine Problemgrößen möglichst schnellen Algorithmus (das ist zumeist Insertionsort) sortiert werden. Unter der Annahme, dass der Sortieralgorithmus für kleine Problemgrößen lineare Laufzeit hat, ist Bucketsort von der Ordnung $O(n * k)$, wobei k die Anzahl an Buckets bezeichnet.
Eine gute und einfach umzusetzende Optimierung ist das Sortieren des Arrays *nach* dem Abschließen der Gather-Phase. Mit Insertionsort sind hier gute Zugewinne erreichbar. Außerdem lohnt es sich ggf. mit der Größe von k herumzuspielen...

```
1  template <class T>
2  void bucketsort(std::vector<T>& vec) {
3    using bucket = std::vector<T>;
4    T const minimum = min(vec), maximum = max(vec);
5    size_t const bucket_count = vec.size();
6    unsigned const bucket_range = (maxumum - minimum + 1)/bucket_count + 1;
7    bucket* buckets = new bucket[bucket_count];
8
9    for(size_t i = 0; i < vec.size(); ++i) //scatter
10     buckets[size_t(vec[i] - minimum)/bucket_range].push_back(vec[i]);
11
12   //sort
13   for(size_t i = 0; i < bucket_count; ++i) insertionsort(buckets[i]);
14
15   size_t i = 0;
16   for(size_t i = 0; i < bucket_count; ++i) // gather
17     for(auto const& elem : buckets[i]) vec[i++] = elem
18
19   delete[] buckets;
20 }
```

5.8 Radixsort

Radixsort ist mit Bucketsort und Countingsort verwandt. Hierbei werden die Werte in nach den jeweils betrachteten Stellen in Buckets verteilt, dann wieder eingesammelt und nach der nächsten Stelle wiederum verteilt. Am Ende ist das Array sortiert. Das entspricht dem Vorgehen von Lochkartensortierern. Die Laufzeit ist $\Theta(m * n)$ wobei m die Länge des Schlüssels bezeichnet. Sollte diese konstant sein (wie etwa bei 32-bit Ganzzahlen) ist Radixsort de facto linear.
Hier eine oktale Implementation, die sich jeweils ein Byte pro Durchlauf ansieht. Von allen betrachteten Implementationen ist diese die schnellste. Gewöhnlich nimmt man als Radix 10.

```
1  template <class T>
2  void octal_radixsort(std::vector<T>& vec) {
3    std::array<std::vector<T>, 256> buckets;
4    for(size_t i = 0; i < sizeof(T); i++) {
5      for(auto const& vec_elem : vec) // scatter
6        buckets[(vec_elem >> (i << 3)) & 0xFF].push_back(vec_elem);
7      auto vec_iter = vec.begin();
8      for(auto& bucket : buckets) { // gather
```

```
9        for(auto const& bucket_elem : bucket)
10         *(vec_iter++) = bucket_elem;
11       bucket.clear();
12     }
13   }
14 }
```

Diese Implementation ist natürlich nur für `uint` geeignet. Negative Zahlen werden nicht korrekt sortiert. Das ließe sich durch ein Verschieben des Zahlenbereiches beheben. Andere Datentypen bieten keine bitweisen Operatoren an, und sind damit out.

6 Listen

Der Begriff „Liste“ ist uneindeutig. Wir reden hier über „scattered memory“ Listen, also nicht über fortlaufende (Array) Konstrukte.

Listen sind weit verbreitete dynamische Datenstrukturen. Im Gegensatz zu Arrays ermöglichen sie das Einfügen an einer beliebigen Stelle (ohne dem Verschieben von Werten) und können unbegrenzt wachsen. Ein Nachteil gegenüber Arrays ist jedoch die abhanden gekommene Speicherlokalität, die bedeutet, dass jeder Zugriff einen Cache-Miss zur Folge hat, da die einzelnen Elemente einer Liste auf der Heap verstreut sind und daher die Annahme, dass ein Lesen die umgebenden Elemente in den Cache lädt ungültig ist. Lineares Suchen in Listen ist damit qualvoll langsam. Generell ist die Wahl von dynamisch wachsenden Arrays (`std::vector<T>`), sollte man an irgendeinem Zeitpunkt die Datenstruktur traversieren müssen, immer eine bessere. In einigen Grenzfällen (wie etwa wenn man bereits weiß, dass die verwendeten Listen klein bleiben werden, und die Zeitverluste damit tolerierbar sind) können natürlich Listen verwendet werden, wie in Seperate Chaining. Weiters haben Listen einen gewissen Memory-Overhead (jedes Element verfügt über zusätzlich $1 - 2$ Pointer, um das vorhergehende und/oder das nachfolgende Element zu referenzieren), der durch das Verwenden von fortlaufenden (*contiguous*) Datenstrukturen nicht gegeben ist.[4] Listen sind dennoch wichtige Datenstrukturen.

6.1 singly-linked list

Struktur

```
1  template < class T >
2  class linked_list {
3    private:
4      struct element {
5        T value;
6        element* next{ nullptr };
7        element(T const& val) : value{ val } {}
8      }* head{ nullptr };
9    public:
10     ...
11 };
```

[4] Bjarne Stroustrup: Are Lists evil? – `isocpp.org/blog/2014/06/stroustrup-lists`

Die Liste besteht aus einzelnen Elementen (der Einfachheit halber als `struct` implementiert), die jeweils einen Wert speichern und einen Pointer auf das nächste Element haben. Das letzte Element zeigt auf `nullptr`. Die Liste selbst merkt sich nur das erste Element in der Instanzvariable `head`. Es existieren auch Erweiterungen, wie die **doubly-linked list**, deren Elemente sowohl über `next`- als auch `prev`-Pointer verfügen, was das Entfernen von Elementen einfacher macht, oder auch Listen, die sich sowohl `head` als auch `tail` merken, sodass man sowohl am Kopf als auch am Ende der Liste effizient einfügen und entfernen kann.

insert

```
1 template < class T >
2 void linked_list< T >::insert(T const& val) {
3   element* ptr = new element(val);
4   ptr->next = head;
5   head = ptr;
6 }
```

Wir fügen am Kopf der Liste an, was sehr effizient ist – hier läuft es in $O(1)$, ein Vektor hätte für selbige Operation eine Laufzeit von $O(n)$, da alle bereits enthaltenen Elemente um 1 Feld verschoben werden müssten. Folgendes Bild illustriert den Vorgang:

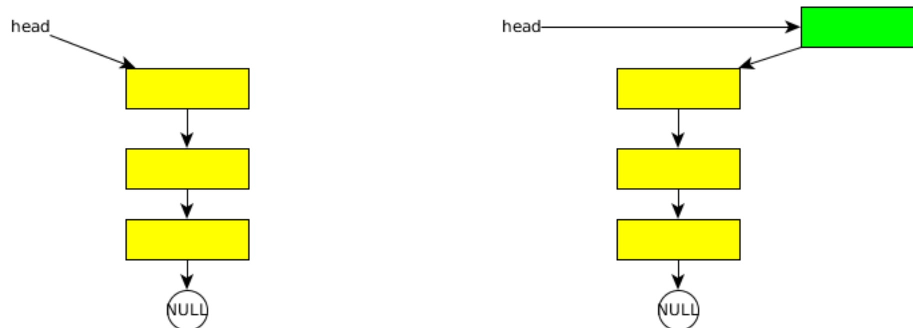

Abbildung 14: Operation `insert`

remove

```
1 template < class T >
2 void linked_list< T >::remove(T const& val) {
3   element* tmp = head;
4   element* prev = nullptr;
5   while(tmp) {
6     if(tmp->value == val) {
7       if(prev) prev->next = tmp->next;
8       else head = tmp->next;
9       delete tmp;
10      return;
11    }
12    prev = tmp;
```

```
13      tmp = tmp->next;
14    }
15  }
```

Da wir eine einfach verkettete Liste (*singly-linked list*) verwenden, ist das Entfernen etwas haarig. Zuerst müssen wir das angestrebte Element finden – wir iterieren dafür mittels `while`-Schleife durch die Liste und merken uns im `prev`-Pointer die jeweils vorhergehenden Elemente, um bei Sucherfolg die Pointer richtig anpassen zu können. Sollte `prev == nullptr`, so möchten wir gerade das Kopfelement entfernen. Ist der `next`-Pointer von `head` null, ist die Liste danach leer und `head` zeigt wieder auf `nullptr`. Folgendes Bild illustriert den Vorgang (den Fall, dass das zu entfernende Element nicht Kopfelement ist):

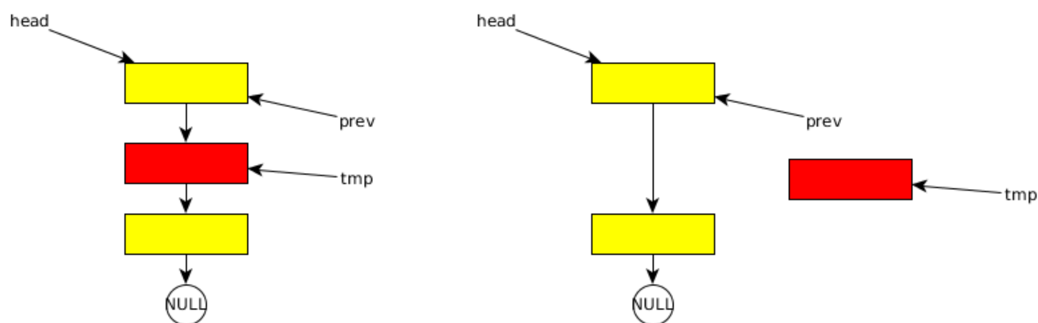

Abbildung 15: Die Operation `remove()`

contains

```
1  template < class T >
2  bool linked_list< T >::contains(T const& val) {
3    element* tmp = head;
4    while(tmp) {
5      if(tmp->value == val) return true;
6      tmp = tmp->next;
7    }
8    return false;
9  }
```

Der Inklusionstest ist selbsterklärend – wir iterieren über die Liste, bis wir entweder ans Ende gelangen oder das gesuchte Element finden. Je nach Situation returnieren wir `true` oder `false`.

6.2 Stack

Da der Stack in seinen Operationen sehr limitiert ist, lässt er sich einfach als Liste implementieren, oder auch als dynamisch wachsendes Array, sofern man zuversichtlich gegenüber dem Wachstum des Stacks ist. Bei einem Stack werden die Elemente in umgekehrter Reihenfolge entfernt, in der sie eingefügt wurden, also `s, t, a, c, k` ⇒ `k, c, a, t, s`. Hier ist ein Stack als Liste gezeigt.

Struktur

```
template < class T >
class stack {
  private:
    struct frame {
      T value;
      frame* next{ nullptr };
      frame(T const& val) : value{ val } {}
    }* top_{ nullptr };
  public:
    ...
};
```

Wie eine einfach verkettete Liste.

push

```
template < class T >
void stack< T >::push(T const& val) {
  frame* ptr = new frame(val);
  ptr->next = top_;
  top_ = ptr;
}
```

Wie `linked_list< T >::insert(val)`.

pop

```
template < class T >
T stack< T >::pop() {
  if(!top) throw empty_error();
  T ret = top_->value;
  frame* del = top_;
  top_ = top_->next;
  delete del;
  return ret;
}
```

`pop` returniert den `top`-Wert und löscht `top`. Falls der Stack leer ist, können wir nicht korrekt fortfahren, daher werfen wir eine Exception. Sonst merken wir uns erst den zu returnierenden Wert in `ret` und die Adresse vom alten `top` in `del`. Wir setzen `top` auf den Nachfolger, löschen `del` und returnieren `ret`.

top

```
template < class T >
T const& stack< T >::top() const {
  return top_->value;
}
```

Echt wild.

6.3 Queue

Eine Queue ist wie ein Stack, nur anders – nämlich FIFO anstatt LIFO. Konkret bedeutet dass, dass die Elemente in der Reihenfolge entfernt werden, in der sie auch eingefügt wurden, also `q, u, e, u, e` ⇒ `q, u, e, u, e`. Wir implementieren diese hier als doppelt verkettete Liste mit zwei Instanzvariablen, namentlich `head` und `tail` für `enqueue(val)` und `dequeue()` respektive. Unsere Queue wächst von unten nach oben, i.e. `front()` liefert das `tail`-Element, das ist arbiträr. Diese Implementation weicht von den Folien ab (ist effizienter – $O(1)$ für alle Operationen), keine Ahnung warum das auf den Folien so umständlich gestaltet ist.

Struktur

```
template < class T >
class queue {
  private:
    struct element {
      T value;
      element* prev{ nullptr };
      element* next{ nullptr };
      element(T const& val) : value{ val } {}
    }* head{ nullptr },* tail{ nullptr };
  public:
    ...
};
```

enqueue

```
template < class T >
void queue< T >::enqueue(T const& val) {
  element* ptr = new element(val);
  if(!head) head = tail = ptr;
  else {
    head->prev = ptr;
    ptr->next = head;
    head = ptr;
  }
}
```

Selber Algorithmus wie beim Einfügen bei der einfach verketteten Liste, mit dem Unterschied, dass das aller erste eingefügte Element (wenn `head == nullptr`) auch `tail` wird.

dequeue

```
template < class T >
T queue< T >::dequeue() {
  if(!tail) throw empty_error();
  T ret = tail->value;
  element* del = tail;
  if(tail->prev) {
    tail = tail->prev;
```

```
    tail->next = nullptr;
  } else head = tail = nullptr;
  delete del;
  return ret;
}
```

Etwas komplizierter. Wenn die Queue leer ist können wir nicht fortfahren, daher werfen wir eine Exception. Wir merken uns zuerst den `tail`-Wert, den wir returnieren wollen, in `ret` und die Adresse des `tail`-Elementes in `del`. Wenn `tail` einen Vorgänger hat, setzen wir `tail` auf diesen, sonst ist `tail` auch gleichzeitig `head` und wir setzen beide auf `nullptr`. Dann löschen wir `del` und returnieren `ret`. Folgendes Bild beschreibt den Vorgang:

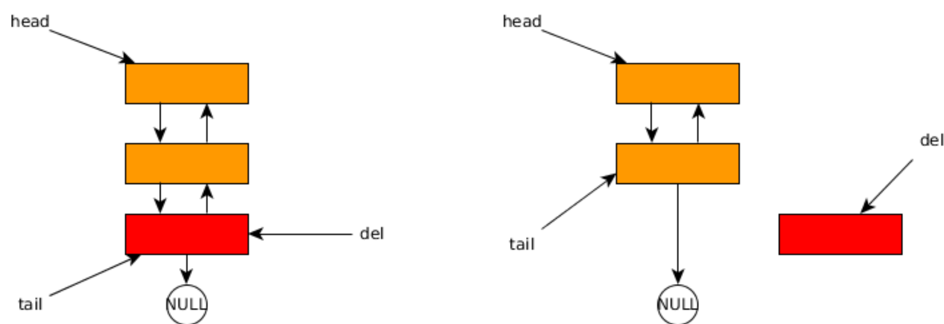

Abbildung 16: Operation `dequeue`

front

```
template < class T >
T const& queue< T >::front() const {
  return tail->value;
}
```

Wiederum sehr wild. Ich weiß garnicht wie ich das erklären soll.

6.4 Spezielle Listen

- Doubly linked list (→ unsere Queue)
- Circular list
- Ordered list
- Double ended list (→ unsere Queue)

7 Bäume

7.1 Eigenschaften

Bäume sind rekursive Datenstrukturen, die unbegrenzt dynamisch wachsen können. Ein Baumknoten verfügt über n Pointer, die auf seine *Kindsknoten* zeigen. Wenn

ein Knoten keine Kindsknoten hat, nennt man ihn *Blatt*. Der oberste Knoten eines Baumes heißt Wurzel.

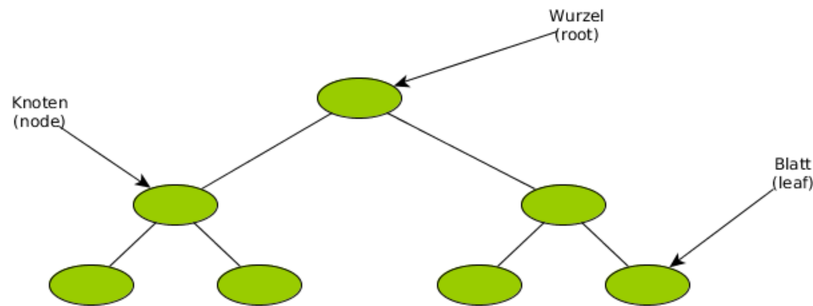

Abbildung 17: Binärer Baum

Die Anzahl an maximalen Kindsknoten, die ein Knoten haben kann, heißt *Grad* des Baumes. Die Höhe eines perfekt balanzierten Baumes ist definiert als $\lfloor \log_g n \rfloor$, im Falle des binären Baumes, der Grad 2 hat, wäre die Höhe also der Logarithmus dualis der Anzahl an Knoten. Der Baum in der Abbildung hat 7 Knoten, die Höhe muss also 2 sein.

7.2 Binärer Suchbaum

In einem binären Suchbaum ist das linke Kind eines Knotens immer kleiner und das rechte immer größer gleich dem Elternknoten.

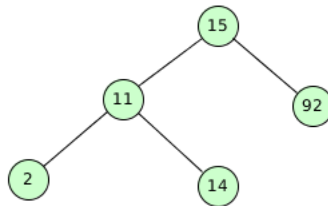

Abbildung 18: Binärer Suchbaum, der 5 Werte speichert.

Struktur

```
template < class T >
class binary_search_tree {
  private:
    struct node {
      T value;
      node* left;
      node* right;
    }* root;
  public:
    ...
```

```
};
```

Wir müssen uns nur merken, wo der Wurzelknoten ist. Alle anderen Operationen erfolgen über den Wurzelknoten – daraus folgt auch, dass alle Baumoperationen mit Ausnahme des Traversierens $O(\log n)$ sind, da in jedem Fall ein Abstieg zur gesuchten Stelle geschieht. Ein Ausnahmefall ist natürlich der zur linearen Liste entarteter Baum, der für alle Operationen eine Komplexität von $O(n)$ hat.

insert

```
template < class T >
void binary_search_tree::insert(node* const ptr, T val) {
  if(!root) {
    root = new node(val);
    return;
  }
  if(val < ptr->value) {
    if(ptr->left) {
      insert(ptr->left, val);
    } else {
      ptr->left = new node(val);
    }
  } else {
    if(ptr->right) {
      insert(ptr->right, val);
    } else {
      ptr->right = new node(val);
    }
  }
}
```

Rekursiv implementiert. Wenn kein Wurzelknoten vorhanden ist, allokieren wir diesen und returnieren sofort. (Diese Anweisung wäre besser in der aufrufenden Funktion aufgehoben!) Ansonsten geschieht zuerst ein Abstieg, wobei immer wieder der einzufügende Wert mit dem Wert des derzeit betrachteten Knoten verglichen wird, um den richtigen Pfad zu finden. Sobald man am `nullptr` angelangt ist, wird dort ein Knoten mit dem entsprechenden Wert allokiert.

```
template < class T >
void bst< T >::add(T const& val) {
  if(!root) {
    root = new node(val);
    return;
  }
  node* tmp = root;
  while("look mum, no recursion!") {
    if(val < tmp->value)
      if(tmp->left) tmp = tmp->left;
      else {
        tmp->left = new node(val);
        tmp->left->parent = tmp;
        return;
      }
    else
      if(tmp->right) tmp = tmp->right;
      else {
```

```
19        tmp->right = new node(val);
20        tmp->right->parent = tmp;
21        return;
22      }
23  }
24 }
```

Iterativ implementiert. Wie zuvor – wenn keine Wurzel, machen wir eine Wurzel und sind fertig. Ansonsten traversieren wir den entsprechenden Pfad, der durch die Relation zwischen den Werten, die schon im Baum sind und dem einzufügenden Wert gegeben ist, bis wir den `nullptr` finden und dort den Wert allokieren. Außerdem setzen wir hier noch den `parent`-Pointer entsprechend. Das haben wir dazu genommen, um `remove` ein bisschen angenehmer zu gestalten.

remove

```
1  template < class T >
2  void bst< T >::replace_self_in_parent(node* node_, node* successor) {
3    if(successor) successor->parent = node_->parent;
4    if(!node_->parent) {
5      root = successor;
6    } else {
7      if(node_ == node_->parent->left) node_->parent->left = successor;
8      else node_->parent->right = successor;
9    }
10 }
11
12 template < class T >
13 void bst< T >::remove(T const& val) {
14   node* del = find_node(val);
15   if(!del) return; // nicht gefunden
16   if(del->left && del->right) {
17     T min = find_min(del->right);
18     remove(min);
19     del->value = min;
20     return;
21   }
22   if(del->left) replace_self_in_parent(del, del->left);
23   else if(del->right) replace_self_in_parent(del, del->right);
24   else replace_self_in_parent(del, nullptr);
25   delete del;
26 }
```

Iterativ implementiert. `find_node` sucht, ähnlich wie `insert`, in einer `while`-Schleife den Wert und returniert einen Pointer auf den entsprechenden Knoten. Bei erfolgloser Suche returniert sie `nullptr`. Sollte der Knoten zwei Kinder haben, finden wir das Minimum des rechten Teilbaumes und ersetzen den Wert in dem zu löschenden Knoten durch das Minimum. Stattdessen entfernen wir das Minimum. Sollte der Knoten ein Kind haben, so merken wir uns den Knoten und passen mittels `replace_self_in_parent` die Pointer an. Wenn keine Kinder vorhanden sind, setzen wir die entsprechenden Pointer `null`. Der Wurzelknoten ist der einzige Knoten, dessen `parent`-Pointer `null` sein kann. Am Ende hauen wir den `del`-Pointer weg.

traverse

inorder

```
template < class T >
void binary_search_tree< T >::inorder_traverse(node* node) {
  if(!node) return;
  inorder_traverse(node->left);
  std::cout << node->value << ' ';
  inorder_traverse(node->right);
}
```

Iterativ:

```
template < class T >
void bst< T >::itraverse_inorder() {
  stack< node* > stack;
  node* current = root;

  while("juhu baum") {
    while(current) {
      stack.push(current);
      current = current->left;
    }
    if(!stack.empty()) {
      current = stack.pop();
      std::cout << current->value << ' ';
      current = current->right;
    } else return;
  }
}
```

preorder

```
template < class T >
void binary_search_tree< T >::preorder_traverse(node* node) {
  if(!node) return;
  std::cout << node->value << ' ';
  preorder_traverse(node->left);
  preorder_traverse(node->right);
}
```

Iterativ:

```
template < class T >
void bst< T >::itraverse_preorder() {
  stack< node* > stack;
  stack.push(root);
  node* current = nullptr;

  while(!stack.empty()) {
    current = stack.pop();
    std::cout << current->value << ' ';
    if(current->left) stack.push(current->left);
    if(current->right) stack.push(current->right);
  }
}
```

postorder

```
template < class T >
void binary_search_tree< T >::postorder_traverse(node* node) {
  if(!node) return;
  postorder_traverse(node->left);
  postorder_traverse(node->right);
  std::cout << node->value << ' ';
}
```

Iterativ:

```
template < class T >
void bst< T >::itraverse_postorder() {
  stack< node* > first_stk;
  stack< node* > second_stk;
  first_stk.push(root);
  node* current = nullptr;

  while(!first_stk.empty()) {
    current = first_stk.pop();
    second_stk.push(current);
    if(current->left) first_stk.push(current->left);
    if(current->right) first_stk.push(current->right);
  }

  while(!second_stk.empty()) {
    current = second_stk.pop();
    std::cout << current->value << ' ';
  }
}
```

Dafür gibt es auch eine nicht-rekursive Version, die mit nur einem Stack auskommt. Die ist aber etwas komplexer (ein paar Randfälle).

bfs

```
template < class T >
void bst< T >::traverse_bfs() const {
  queue< node* > queue;
  queue.enqueue(root);
  node* current = nullptr;

  while(!queue.empty()) {
    current = queue.dequeue();
    std::cout << current->value << ' ';
    if(current->left) queue.enqueue(current->left);
    if(current->right) queue.enqueue(current->right);
  }
}
```

7.3 inorder, preorder, postorder und levelorder Traversierung an einem Beispiel

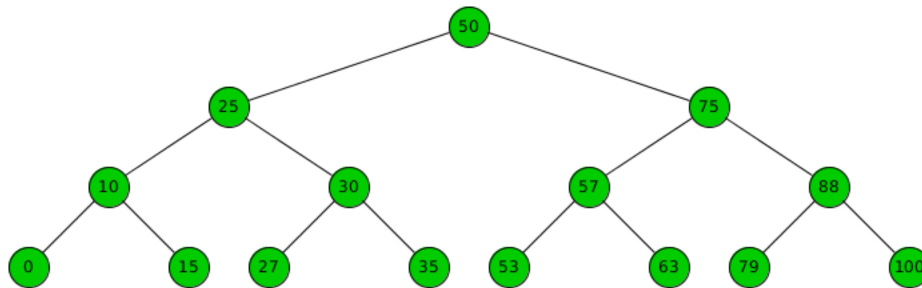

Abbildung 19: Binärsuchbaum

inorder
Die inorder-Traversierung entspricht einer Ausgabe in sortierter Reihenfolge. Zuerst erfolgt der Abstieg in den linkesten Knoten, der den Wert 0 hält. Dieser wird ausgegeben, ein Stackframe fällt weg, und der Elterknoten wird betrachtet. Danach wird das rechte Kind betrachtet. Dies setzt sich fort, bis der gesamte linke Teilbaum des Wurzelknoten abgearbeitet wurde. Danach geschieht dasselbe mit dem rechten Unterbaum. Die Ausgabe wäre: **0 10 15 25 27 30 35 50 53 57 63 75 79 88 100**.

preorder
Zuerst (prä) wird der Elterknoten betrachtet, dann *alle* Kindsknoten (zuerst links, dann rechts). Die Ausgabe wäre **50 25 10 0 15 30 27 35 75 57 53 63 88 79 100**.

postorder
Zuerst werden *alle* Kindsknoten (zuerst links, dann rechts) betrachtet, danach (post) der Elterknoten. Die Ausgabe wäre **0 15 10 27 35 30 25 53 63 57 79 100 88 75 50**.

levelorder (bfs)
Wir traversieren den Baum „stockweise“, also besuchen zuerst den Wurzelknoten, dann seine zwei Kinder, dann jeweils die Kinder der Kinder, etc. Dazu benötigen wir eine Hilfsdatenstruktur, nämlich unsere wunderschöne Queue aus 6.3. Die Ausgabe wäre **50 25 75 10 30 57 88 0 15 27 35 53 63 79 100**.

7.4 Laufzeit von DFS-Traversierung beim binären Suchbaum

Jeder Funktionsaufruf beinhaltet zwei rekursive Aufrufe, die, unter der Annahme, dass der Baum perfekt balanziert ist, nurnoch halb soviele Werte bearbeiten. Weiters ist die Ausgabefunktion konstant. Die Rekurrenz ist daher in allen 3 Fällen:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n^0)$$

Damit erhalten wir via Mastermethode $a = 2$, $b = 2$ und $c = 0$. $\log_2 2 = 1$, also ist Fall 1 wahr, dessen Ergebnis $\Theta(n^{\log_b a})$ ist. Somit ist die Laufzeitkomplexität von DFS bei Bäumen $\Theta(n)$. Jeder Knoten wird einmal besucht.

7.5 Mehrwegbäume

Mehrwegbäume sind Bäume, die mehr als 2 Kinder besitzen können.

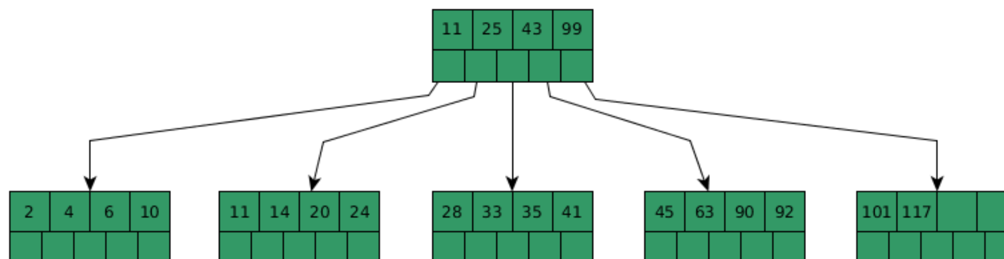

Abbildung 20: Mehrwegbaum des Grades 5.

Jeder Knoten speichert maximal $g - 1$ Schlüsselwerte und verweist auf g weitere Knoten (oder `nil`). Die Werte können entweder mit ihren Schlüsseln in ihren Knoten oder als externe Knoten gespeichert werden. Die Referenzen legen Intervalle fest, sodass der Wert 35 im Knoten gefunden wird, der von `nptrs[2]` referenziert wird, da `keys[1]` $\leq 35 <$ `keys[2]`. Folgende Funktion erledigt die Wegfindung für das Suchen eines Wertes:

```
1 template < class T >
2 size_t get_ptr_index(T value) {
3   if(value < keys[0]) return 0;
4   for(size_t i = 0; i < key_count - 2; ++i)
5     if(key[i] <= value and value < key[i + 1]) return i + 1;
6   return key_count;
7 }
```

Es wird die entsprechende Indexposition des zugehörigen Pointers zurückgegeben. Es ist absolut notwendig, dass die Schlüsselwerte aufsteigend sortiert sind! Die Werte selbst können in externen Knoten in beliebiger Reihenfolge angebracht sein, zumeist sind aber auch diese sortiert (beispielsweise im B^+-Baum).

7.6 2-3-4 Baum

Jeder Knoten eines 2-3-4-Baums hat mindestens 2 und maximal 4 Kinder (daher der Name). Alle externen Knoten besitzen die selbe Tiefe. Ein Überlauf eines Knotens bedeutet das Splitten in einen 2-Knoten und einen 3-Knoten. Es wird darauf versucht, einen neuen Schlüssel in den Elterknoten einzufügen, was ggf. dort zu einem Überlauf führt, etc – schlimmstenfalls propagiert sich das Splitting bis zur Wurzel. Beim Löschen wird ein Schlüssel in einem Indexknoten mit weiteren Indexkindern durch seinen inorder Vorgänger oder Nachfolger ersetzt (wie bei binären Suchbäumen). Sollte der betroffene Schlüssel in einem Knoten leben, der nur externe Kindsknoten hat, wird der Schlüssel einfach entfernt.

Durch das Entfernen von Schlüsseln kann es zu einem Unterlauf kommen, was entweder das Verschmelzen von Knoten oder dem Verschieben von Werten zur Folge hat.

Verschmelzen
Wenn alle unmittelbar Benachbarten Knoten 2-Knoten sind, kann der betroffene Knoten mit seinem Nachbarn zusammen gelegt werden. Dabei wird der nicht mehr benötigte Schlüsselwert aus dem Indexknoten in den soeben verschmolzenen Knoten verschoben. Wie beim Splitten kann auch das Mergen sich bis zur Wurzel nach oben propagieren.

Verschieben
Wenn ein unmittelbar adjazenter Knoten ein 3- oder 4-Knoten ist, verschiebt man ein Kind zu dem unterlaufenen Knoten, schiebt den Schlüssel dazu in den betroffenen Indexknoten und schiebt einen anderen Schlüssel aus dem ehemaligen 3- oder 4-Knoten in den darüberliegenden Knoten. Dies kann keine Split- oder Merge-Operation auslösen, da Werte nur verschoben werden und nicht neu dazu kommen oder entfernt werden.

7.7 B^+-Baum

Ein B^+-Baum ist ein externspeicherorientierter, höhenbalanzierter Mehrwegbaum. Der Aufwand für einen Zugriff ist durch die Zusicherung, dass der Weg zu allen Daten gleich lang ist, von der Ordnung $\Theta(\log n)$. Um diese Eigenschaften aufrecht zu erhalten sind Algorithmen zur Balanzierung der Last bei Einfügen/Entfernen notwendig (Spalten bei Überlauf, Verschmelzen bei Unterlauf).

7.7.1 Eigenschaften

Die Knoten eines B^+-Baums der Ordnung k haben mindestens 2 und maximal $2k + 1$ Kinder. Eine Ausnahme ist der Wurzelknoten, der auch nur ein Kind haben kann. Alle internen Knoten haben mindestens k und maximal $2k$ Schlüsselwerte und folglich mindestens $k + 1$ Kinder und maximal $2k + 1$ Kinder. Ausnahme ist die Wurzel, die auch nur einen Schlüsselwert (und damit 2 Kinder) haben kann. Alle Daten werden in externen Knoten gespeichert, Kopien der Schlüssel sind in internen Knoten gespeichert und geben den Pfad zum externen Knoten an. Im linken Unterbaum sind nur Werte gespeichert, die kleiner dem dazugehörigen Schlüsselwert sind, und im rechten Unterbaum sind nur Werte gespeichert, die größer oder gleich dem entsprechendem Schlüssel sind. Die Schlüssel stecken also Intervalle ab und bilden diese auf die Pointer auf die Kindsknoten ab.

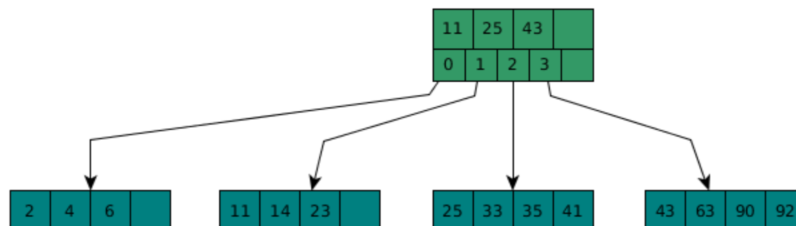

Abbildung 21: B^+-Baum der Ordnung 2

7.7.2 Balanzierung

Beim Einfügen

Wollen wir den Wert 42 einfügen, so wissen wir, dass er in den Knoten 2 kommen muss, da $25 \leq 42 < 43$. Dieser ist aber bereits bei voller Kapazität $2k$, wir müssen ihn also spalten. Glücklicherweise ist noch Platz im Wurzelknoten. Wir allokieren also einen neuen Knoten, schieben die Hälfte der Werte in diesen, kopieren den kleinsten Schlüssel an die richtige Stelle im Wurzelknoten und fügen den Wert 42 ein. Folgende Abbildung zeigt den Baum nach diesem Vorgang.

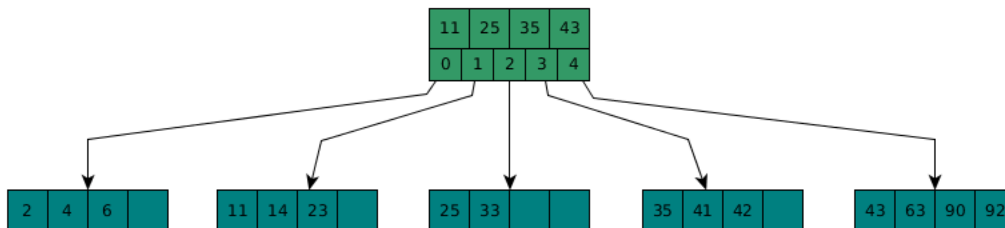

Abbildung 22: Unser B^+-Baum nach dem Einfügen von 42

Jetzt wollen wir aber noch den Wert 99 einfügen. Wir wissen, dass dieser Wert in den Knoten 4 wandern muss, der aber auch schon voll ist. Die Wurzel ist ebenfalls bereits voll. Das ist echt bitter. Wir müssen also zuerst den Wurzelknoten spalten und dann den Knoten 4.

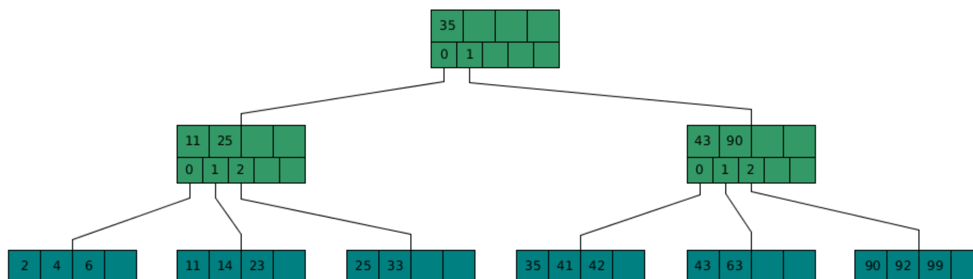

Abbildung 23: Unser B^+-Baum nach dem Einfügen von 99

Wir nahmen den mittleren Schlüsselwert (Position 2) und versuchten diesen in den darüber liegenden Indexknoten zu schieben. Da es diesen nicht gibt, mussten wir eine neue Wurzel anlegen und dann diese Operation nochmal ausführen. Knoten 4 wurde gespalten und ein neuer Schlüsselwert wurde in den darüber liegenden Indexknoten eingetragen.

Anmerken muss man, dass sich diese Aktion ggf. von weit unten bis zur Wurzel hoch propagieren kann, was viel Aufwand bedeutet.

Beim Löschen
Wir haben genug von dem Wert 33 und entfernen diesen. Nach dem Entfernen ist der Knoten nur noch mit einem Wert besetzt, es gilt also $1 < k$; damit gibt es einen Unterlauf. Wir müssen den Knoten wenn möglich mit einem seiner Nachbarn verschmelzen. Glücklicherweise ist dort Platz für den Wert 25.

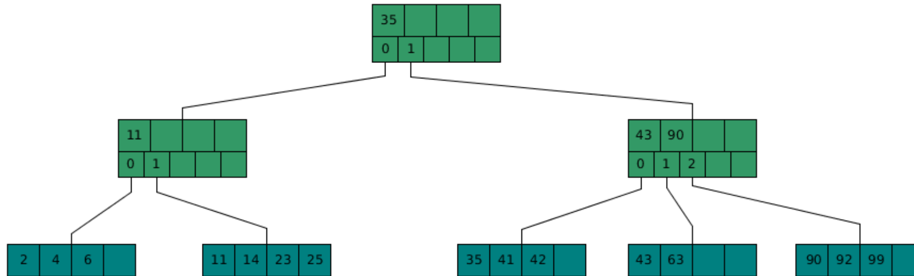

Abbildung 24: Löschen von Wert 33: Teil 1

Halt, stopp! Jetzt ist jedoch der Indexknoten, der den Schlüssel 11 hält, unterbesetzt. Wir müssen nun auch diesen verschmelzen.

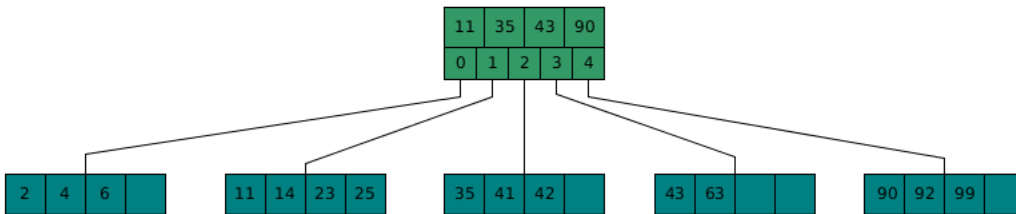

Abbildung 25: Löschen von Wert 33: Teil 2 – *Return of the Segfault*

Wir sind jetzt wieder schlank und schön. Es ist auch möglich, dass das Verschmelzen mit einem Nachbarn nicht möglich ist, da das zu einem Überlauf im Nachbarn führen würde. In diesem Fall wird dieser Überlauf willentlich in Kauf genommen und der Knoten mit allen eventuellen Komplikationen gespalten. Dies hat eine bessere Verteilung der Werte zur Folge.
Sollte ein Wert gelöscht werden, der auch als Schlüssel verwendet wird, so bleibt der Wert als Schlüssel vorhanden bis der dazugehörige Knoten gelöscht wird.

Außerdem werden oft die Bedingungen für das Verschmelzen von Knoten etwas lockerer gesehen, da dem Mergen zweier Knoten oft gleich nach der nächsten Einfügeoperation ein Split folgt. Dies nennt man *Lazy Deletion*. Das ist vorallem bei nebenläufigen Systemen wichtig. Knoten können bei diesem Ansatz komplett leer werden, was das Entfernen vereinfacht und unnötiges Splitten/Mergen unterbindet. Dies öffnet allerdings dem Risiko, dass der Baum entartet, Tür und Tor – wenn etwa nur kleine Schlüssel entfernt werden, könnte der Baum letztlich fast ausschließlich aus leeren Knoten bestehen. Dieser Fall ist in der Realität allerdings vernachlässigbar. Die

meisten modernen Datenbanksysteme verwenden Lazy Deletion.[5] Weiters kann man bei Unterbelegung von Blättern Werte von Nachbarn ausborgen und so ein Splitting vermeiden. Dabei müssen aber die Schlüsselwerte im darüber liegenden Indexknoten aktualisiert werden.

7.8 Trie

Ein Trie ist ein Mehrwegbaum, dessen Schlüssel den Pfad definieren. Sollte beispielsweise das Wort `key` gesucht werden, so würden die Kanten `k, e, y` gewählt werden, sofern diese existieren. Tries werden beispielsweise für Wörterbuchimplementationen verwendet. Weiters ist Extendible Hashing vom Konzept her ein geplättetes Binär-Trie.

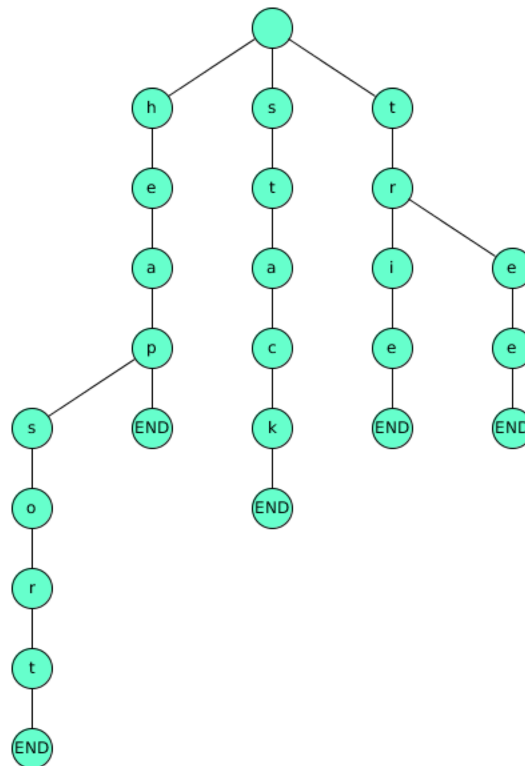

Abbildung 26: Trie, das die Wörter `heap, heapsort, stack, trie` und `tree` speichert.

7.9 Priority Queue

Eine Priority Queue speichert Werte nach ihrer Priorität geordnet. Ein `dequeue`-Aufruf liefert also den Wert mit der größten (oder kleinsten) Priorität. Sollten zwei Werte die gleiche Priorität haben, erhält der Wert der zuerst hinzugefügt wurde Vorrang (FIFO) – wie beim Song Contest. Priority Queues werden zum Beispiel für *process scheduling* verwendet. Es gibt unterschiedliche Implementationsansätze für PQs, der bekannsteste ist der Heap.

[5] ilpubs.stanford.edu:8090/85/1/1995-19.pdf – Implementing Deletion in B^+-Trees: Jan Jannink, Stanford University

7.9.1 Heap

Ein Heap ist als ungeordneter Binärbaum zu verstehen. Die einzige Einschränkung ist, dass alle Kinder eines Knotens größer gleich (*Min-Heap*) oder kleiner gleich (*Max-Heap*) dem Elterknoten sein müssen (Heap-Bedingung). Durch diese Eigenschaft lassen sie sich effizient für HIFO-Strukturen verwenden, da der Zugriff auf das Element mit höchster Prioriät konstant ist, da es immer das Wurzelelement ist. Weiters lässt sich ein Heap speichereffizient als schlichtes Array darstellen, wie folgende Abbildungen illustrieren.

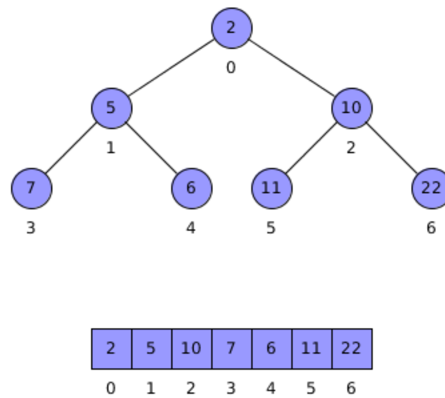

Abbildung 27: Ein Min-Heap in Baum- und Felddarstellung

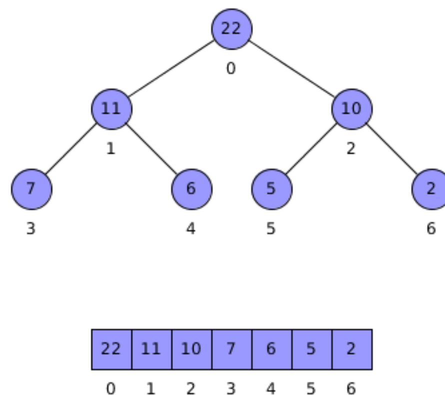

Abbildung 28: Selbiges als Max-Heap

Das linke Kind befindet sich immer am Index `parent * 2 + 1`. Das rechte ist am Index `parent * 2 + 2`, also eins weiter, zu finden.

insert

```
1 template < class T >
2 void bin_heap< T >::insert(T value) {
```

```
  heap[++last] = value;
  int child = last;
  int parent = (child - 1)/2;
  while(child != parent) {
    if(heap[parent] > heap[child]) {
      std::swap(heap[parent], heap[child]);
      child = parent;
      parent = (parent - 1)/2;
    } else break;
  }
}
```

Wir schreiben neue Werte immer an den letzten Index. Danach prüfen wir, ob die Heap-Bedingung verletzt ist – wenn ja, vertauschen wir den Wert mit seinem Elterknoten. Im Min-Heap prüfen wir also, ob der Elterknoten **größer** als der gerade eingefügte Wert ist. Falls dies nicht zutrifft, gilt die Min-Heap-Bedingung nicht, was durch Tauschen behoben wird. Dies propagiert sich im schlimmsten Fall bis zur Wurzel, da es sich beim Heap um eine Baumstruktur handelt ist die Operation also von $O(\log n)$.

remove_min

```
template < class T >
T bin_heap< T >::remove_min() {
  T value = heap[0];
  int parent = 0;
  int child = 1;
  a[0] = a[last--];
  while(child <= last) {
    if(heap[child] > heap[child + 1]) ++child;
    if(heap[child] < heap[parent + 1]) {
      std::swap(heap[parent], heap[child]);
      parent = child;
      child = 2 * child + 1;
    } else break;
  }
  return value;
}
```

Wir überschreiben den Wurzelwert mit dem letzten und überprüfen darauf wieder, ob die Heap-Bedingung verletzt wurde. Wenn ja, vertauschen wir wiederum Werte mit ihren Elterknoten. Dies propagiert sich im schlimmsten Fall bis zu den Blättern, ist daher also wieder $O(\log n)$.

Für ein Max-Heap sind alle entsprechenden Relationen umzudrehen.

bfs-Traversierung

Die bfs-Traversierung eines Heaps in Arraydarstellung ist trivial und besteht aus einer simplen Schleife. In Baumdarstellung muss eine Queue als Hilfsdatenstruktur eingesetzt werden. Siehe 7.3 (levelorder).

8 Graphen

Graphen sind eine integrale Datenstruktur der Informatik. Viele Probleme lassen sich durch Graphen beschreiben und mit Graphenalgorithmen lösen. Viele der schwierigsten Probleme der Menschheit lassen sich als Graphprobleme beschreiben (aber nicht lösen ... weil schwierig).

8.1 Eigenschaften

Graphen bestehen aus Knoten (*vertex*) und Kanten (*edge*). Knoten können durch Kanten miteinander verbunden sein. Man unterscheidet zwischen gerichteten Kanten und ungerichteten Kanten beziehungsweise zwischen gerichteten und ungerichteten Graphen.
Formal beschrieben ist ein Graph ein Zwei-Tupel $G = (V, E)$, wobei V die Menge aus Knoten $V = \{v_1, v_2, v_3, \ldots, v_i\}$ und E die Menge aus Kanten $E = \{e_1, e_2, e_3, \ldots, e_j\}$ bezeichnet. Knoten verfügen über einen **Grad**, der bei ungerichteten Graphen die Anzahl an Kanten, die von einem Knoten ausgehen beziehungsweise in diesem münden (bei ungerichteten Graphen sind die beiden äquivalent). Bei gerichteten Graphen gibt es pro Knoten ein *out-degree* und ein *in-degree*, die die Anzahl von ausgehenden und eingehenden Kanten respektive angibt.

Eine **Kantenfolge** ist eine Folge von Kanten $[v_1, v_2], [v_2, v_3], \ldots, [v_{i-1}, v_i]$. Eine Kantenfolge wird Pfad oder Weg genannt, wenn alle Knoten in dieser unterschiedlich sind.
Ein Kreis oder **Zyklus** ist eine Kantenfolge, bei der alle Kanten, alle Knoten unterschiedlich sind, und $v_1 = v_n$ gilt. Ein Graph heißt verbunden oder zusammenhängend, wenn für alle möglichen Knotenkombinationen ein Pfad existiert. Ein Baum ist ein zusammenhängender, azyklischer Graph.
Ein Graph G' ist Teilgraph von G wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. Ein Teilgraph G' ist spannender Baum von Graph G wenn $V' = V$ gilt und G' einen Baum bildet. Jeder bezüglich der Kantenanzahl maximale Teilgraph heißt **Komponente**.

Speicherung

Adjazenzmatrixdarstellung

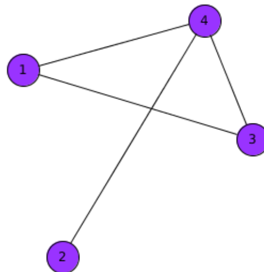

Abbildung 29: Ein kleiner Graph namens Gundulf

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Abbildung 30: Adjazensmatrix für Gundulf

- gut für kleine Graphen, oder Graphen mit vielen Kanten
- Test auf Adjazenz $\in O(1)$
- Speicherkomplexität $O(|V|^2)$
- nicht optimal für DFS

Adjazenzlistendarstellung

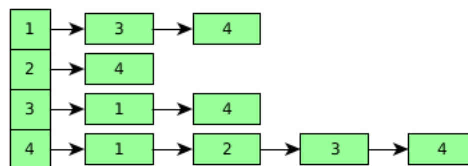

Abbildung 31: Adjazenzlistendarstellung für Gundulf

- gut für Graphen mit wenig Kanten
- Speicherkomplexität $O(|V| + |E|)$
- gut für DFS (linearer Aufwand)
- Test auf Adjazenz $\in O(|V|)$

8.2 Topologisches Sortieren

Anhand einer binären Beziehung („A muss abgeschlossen sein bevor B beginnen kann“) soll eine Reihenfolge gefunden werden, sodass die Beziehungen nicht verletzt werden. Nur azyklische Graphen können topologisch sortiert werden. Gibt es in einem Graph einen Zyklus kann dieser **nicht** topologisch sortiert werden!

Die notwendige Prozedur in Pseudopseudocode:

```
1 running_counter = 0
2 while(node exists where in_degree(node) == 0) do:
3   running_counter += 1
4   node.id = running_counter
5   delete node from graph
6 if(running_counter < |V|): print "no topological order exists!"
```

8.3 Traversierung

Durch Traversierung lassen sich bereits eine Vielzahl von Graphenproblemen lösen. Prinzipiell gibt es zwei Ansätze.

Breadth-First Search
: Geht zuerst in die Breite, besucht also zunächst alle benachbarten Knoten, bevor es weiter in die Tiefe geht.

Depth-First Search
: Geht zuerst in die Tiefe, sucht also soweit, bis es nicht mehr weiter geht und kehrt dann ggf. zurück, um Nachbarn zu besuchen.

8.3.1 Depth-First-Search

DFS wird mit einem Stack als Hilfsdatenstruktur realisiert – entweder implizit als Callstack bei einer rekursiven oder explizit bei einer iterativen Implementierung. DFS kann auf ungerichtete und gerichtete Graphen angewendet werden.

Generell müssen bereits besuchte Knoten als *besucht* markiert werden, sodass Knoten nicht mehrfach besucht werden (durch etwaige Zyklen im Graph). Man vergleiche dies mit den DFS-Ansätzen aus dem Kapitel Bäume, wo solch Markierungen nicht notwendig waren, da ein Baum mit Zyklen kein Baum ist. preorder-Traversierung bei Bäumen ist equivalent zu DFS bei allgemeinen Graphen.
Hierfür verwendet man die Adjazenzlistendarstellung für Graphen. Wir implementieren diese in den folgenden Implementationsbeispielen als einfaches Konstrukt aus Dictionary und Liste. (Van Rossum würde das geil finden.)

Depth-First Search (rekursiv)

```python
def dfs(graph, start, visited=None):
    if visited is None: visited = set()
    if start not in visited:
        print(start)
        visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited
```

Wenn `visited` noch nicht existiert, machen wir dieses. Wenn der gerade betrachtete Knoten noch nicht besucht wurde, geben wir diesen aus und markieren ihn als besucht, in dem wir ihn in das `visited`-Set hinzufügen. Nun rufen wir für jeden Nachbarn, der in der Adjazenzliste für den gerade betrachteten Knoten steht und nicht bereits als besucht markiert ist die Funktion erneut rekursiv auf.
Diese Version besucht nur alle Knoten in einer Zusammenhangskomponente. Dies lässt sich leicht ändern, in dem man noch zusätzlich (in der aufrufenden Funktion) über `graph.keys()` iteriert.

Depth-First Search (iterativ)

```
def dfs(graph, start):
    visited = set()
    stack   = [start]
    while stack:
        vertex = stack.pop()
        if vertex in visited: continue
        print(vertex)
        visited.add(vertex)
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                stack.append(neighbor)
    return visited
```

Wir simulieren wieder den Callstack durch einen Stack. Zuerst fügen wir den Startknoten hinzu und iterieren dann solange der Stack nicht leer ist. In jeder Iteration poppen wir einen Knoten vom Stack runter und betrachten ihn, geben ihn also aus und markieren ihn als besucht. Dann pushen wir alle Nachbarn des betrachteten Knoten, die noch nicht besucht wurden, auf den Stack.

8.3.2 Breadth-First Search

Bei der Breitensuche werden alle möglichen Alternativen auf einmal erforscht, es werden also zuerst alle augehenden Kanten eines Knotens untersucht, bevor zum nächsten Knoten weitergegangen wird (wie eine Welle).
Generell besucht BFS zuerst alle Knoten, deren kürzester Pfad zu v Länge 1 hat, dann alle Knoten, deren kürzester Pfad zu v Länge 2 hat, etc. $\rightarrow$ stockweises Traversieren beim Baum!

Breadth-First Search (iterativ)

Der einzige Unterschied besteht hierin, dass wir eine Queue anstatt eines Stacks verwenden. Das äußert sich in einem FIFO-Verhalten.

```
def bfs(graph, start):
    visited = set()
    queue   = [start]
    while queue:
        vertex = queue.pop(0)
        if vertex in visited: continue
        print(vertex)
        visited.add(vertex)
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
    return visited
```

8.3.3 Ein Beispiel traversiert mit BFS und DFS

Wir wollen folgenden Graphen einmal depth-first und einmal breadth-first traversieren.

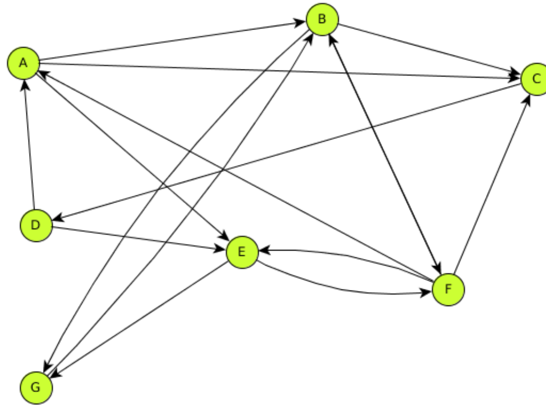

Abbildung 32: Unser Versuchskaninchen, Kevin, mit Zyklen

Die Adjazenzlistendarstellung ist folgendermaßen gegegeben:

```
graph = {'A': ['B', 'C', 'E'],
         'B': ['C', 'F', 'G'],
         'C': ['D'],
         'D': ['A', 'E'],
         'E': ['F', 'G'],
         'F': ['A', 'B', 'C', 'E'],
         'G': ['B']}
```

Wir verwenden hierzu die beiden iterativen Implementierungen, die zuvor gezeigt wurden, als Grundlage und erhalten folgende Ergebnisse. Wir beginnen bei Knoten A und arbeiten Nachbarsknoten alphabetisch ab.

DFS

1. Wir schreiben A auf den Stack.
   $stack = \{A\}$
   $visited = \{\ \}$
2. Wir betrachten A und markieren A als besucht. Wir schreiben B, C, E auf den Stack.
   $stack = \{B, C, E\}$
   $visited = \{A\}$
3. Wir nehmen E vom Stack und betrachten ihn. Wir schreiben F, G auf den Stack.
   $stack = \{B, C, F, G\}$
   $visited = \{A, E\}$
4. Wir nehmen G vom Stack und betrachten ihn. Wir schreiben B auf den Stack.
   $stack = \{B, C, F, B\}$
   $visited = \{A, E, G\}$
5. Wir nehmen B vom Stack und betrachten ihn. Wir schreiben C, F auf den Stack, da G schon besucht wurde.

$stack = \{B, C, F, C, F\}$
$visited = \{A, E, G, B\}$

6. Wir nehmen F vom Stack und betrachten ihn. Wir schreiben C auf den Stack, da A, B, E schon besucht wurden.
$stack = \{B, C, F, C, C\}$
$visited = \{A, E, G, B, F\}$
7. Wir nehmen C vom Stack und betrachten ihn. Wir schreiben D auf den Stack.
$stack = \{B, C, F, C, D\}$
$visited = \{A, E, G, B, F, C\}$
8. Wir nehmen D vom Stack. Wir schreiben nichts auf den Stack.
$stack = \{B, C, F, C\}$
$visited = \{A, E, G, B, F, C, D\}$
9. Alle Knoten, die jetzt noch am Stack sind, wurden bereits besucht. Sie werden nun vom Stack entfernt, aber nicht bearbeitet.
$stack = \{\ \}$
$visited = \{A, E, G, B, F, C, D\}$

BFS

1. Wir schreiben A in die Queue.
$queue = \{A\}$
$visited = \{\ \}$
2. Wir nehmen A aus der Queue und betrachten ihn. Wir schreiben B, C, E in die Queue. Wir markieren A als besucht.
$queue = \{B, C, E\}$
$visited = \{A\}$
3. Wir nehmen B aus der Queue und betrachten ihn. Wir schreiben C, F, G in die Queue.
$queue = \{C, E, C, F, G\}$
$visited = \{A, B\}$
4. Wir nehmen C aus der Queue und betrachten ihn. Wir schreiben D in die Queue.
$queue = \{E, C, F, G, D\}$
$visited = \{A, B, C\}$
5. Wir nehmen E aus der Queue und betrachten ihn. Wir schreiben F, G in die Queue.
$queue = \{C, F, G, D, F, G\}$
$visited = \{A, B, C, E\}$
6. Wir nehmen C aus der Queue und tun nichts damit.
$queue = \{F, G, D, F, G\}$
$visited = \{A, B, C, E\}$
7. Wir nehmen F aus der Queue und betrachten ihn. Wir schreiben nichts in die Queue.

$queue = \{G, D, F, G\}$
$visited = \{A, B, C, E, F\}$

8. Wir nehmen G aus der Queue und betrachten ihn. Wir schreiben nichts in die Queue, da B schon besucht wurde.
$queue = \{D, F, G\}$
$visited = \{A, B, C, E, F, G\}$
9. Wir nehmen D aus der Queue und betrachten ihn. Wir schreiben nichts in die Queue.
$queue = \{F, G\}$
$visited = \{A, B, C, E, F, G, D\}$
10. F und G wurden bereits besucht, werden also von der Queue entfernt und nicht bearbeitet.
$queue = \{\ \}$
$visited = \{A, B, C, E, F, G, D\}$

Beide Ergebnisse stimmen mit den iterativen Python-Programmen überein. Die rekursive DFS-Version liefert eine andere Reihenfolge, was auch OK ist. Alles wird gut.

8.4 Dijsktras Algorithmus für kürzeste Wege

Die Idee hinter Dijkstras Algorithmus ist, dass man von einem Knoten weg immer den kürzesten Pfad zum nächsten nimmt. Als solches ist er Vertreter des Greedy-Paradigmas. Man erreicht das Ziel, in dem man Kantengewichte aufsummiert, sich die Vorgänger merkt und in jedem Schritt den globalst (im Sinne von allen bereits betrachteten Knoten) kürzesten Pfad wählt. Am Ende wurde nicht nur der kürzeste Pfad $A \rightarrow Z$ sondern der kürzeste Pfad zu allen Knoten des Graphen die von A erreichbar sind bestimmt. In jeder Iteration wird ein Pfad bestimmt. Das Speichern von allen gefundenen Pfaden lässt zu, weitere Anfragen sehr schnell (durch Nachschauen) zu beantworten. Dijkstras Algoritmus ist von der Ordnung $O(|V|^2)$. Folgend ist Pseudopseudocode für eine Prozedur, die Dijkstras Algorithmus durchführt. `dist` wäre zu Beginn mit ∞ oder `MAX_INT` für jeden Knoten zu initialisieren.

```
1 dijkstra(start):
2   dist[start] = 0
3   while not all vertices have been marked as done
4     v = unvisited vertex with smallest distance
5     mark v as done
6     for each w adjacent to v and not yet marked as done do
7       dist[w] = min(dist[w], dist[v] + cost(v, w))
```

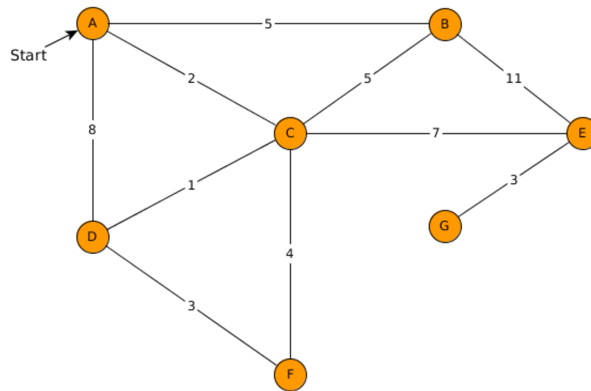

Abbildung 33: Ein Beispielgraph (namens Brunhilde)

Folgende Tabelle illustriert den Vorgang des Algorithmus. Wir beginnen mit einem Knoten und setzen die Kosten für alle benachbarten Knoten gleich den Kosten der Kanten zu diesen Knoten. Weiters merken wir uns den jeweiligen Vorgänger im Subskript. Sollte ein Knoten nicht direkt von unserem Startknoten erreichbar sein, so setzen wir die Kosten für diesen symbolisch auf ∞.

	A	B	C	D	E	F	G
A	0_A	5_A	2_A	8_A	∞	∞	∞
C		5_A	2_A	3_C	9_C	6_C	∞
D		5_A		3_C	9_C	6_C	∞
B		5_A			9_C	6_C	∞
F					9_C	6_C	∞
E					9_C		12_E
G							12_E

Tabelle 1: Brunhilde tabellarisch gelöst

Wir haben in dieser Tabelle alle kürzesten Wege von A nach N berechnet, wobei $N \in \{A, B, C, D, E, F, G\}$. Wollen wir nun beispielsweise den kürzesten Weg von A nach G bestimmen, so machen wir uns einen Stack und pushen G, also die Destination, auf diesen. Der Vorgänger des letzten (dunkelgrünen) Eintrages in Spalte G ist E, also pushen wir E auf den Stack und wechseln in Spalte E. Der Vorgänger dort ist C. Der Vorgänger von C ist A. Der Pfad $A \rightarrow G$ ist also

$$AG = \{[A, C], [C, E], [E, G]\}$$

Ein weiteres, etwas größeres Beispiel:

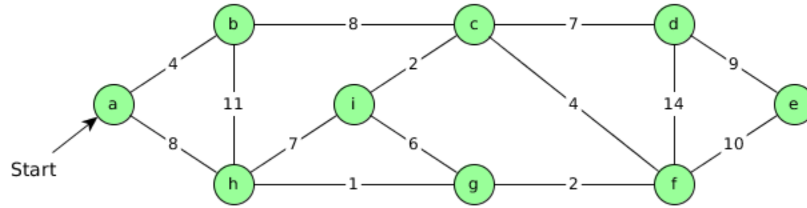

Abbildung 34: Beispielgraph Bertram (aus CLRS)

	a	b	c	d	e	f	g	h	i
a	0_a	4_a	∞	∞	∞	∞	∞	8_a	∞
b		4_a	12_b	∞	∞	∞	∞	8_a	∞
h			12_b	∞	∞	∞	9_h	8_a	15_h
g			12_b	∞	∞	11_g	9_h		15_h
f			12_b	25_f	21_f	11_g			15_h
c			12_b	18_c	21_f				14_c
i				18_c	21_f				14_c
d				18_c	21_f				
e					21_f				

Tabelle 2: Bertram tabellarisch gelöst

8.5 Bellman-Ford Algorithmus für kürzeste Wege

ACHTUNG: Ansich nicht Stoff der Vorlesung. Trotzdem gut zu wissen.

Bellman-Ford funktioniert auch mit negativ gewichteten Kanten und kann zur Findung von negativen Zyklen verwendet werden. Sollte ein Graph negative Zyklen beinhalten, kann kein kürzester Pfad gefunden werden. Die Laufzeitkomplexität von Bellman-Ford ist $O(n * m)$, wobei n die Anzahl an Knoten und m die Anzahl an Kanten bezeichnet. Damit ist Bellman-Ford langsamer als Dijkstra, außer in einem Graphen sind gleich viele Kanten wie Knoten. Weiters lässt sich Dijkstras Algorithmus durch die Verwendung von Priority Queues und Adjazenzlistendarstellung bis auf $O(m \log n)$ optimieren. Die Verwendung einer Fibonacci-Heap reduziert die Laufzeit von Dijkstra weiter auf $O(m + n \log n)$, da die Laufzeit für `decrease_key` bei Fibonacci-Heaps konstant ist.

```
1 bellman_ford(start):
2   dist[start] = 0
3   for i = 1 .. |V|-1 do
4     for all edges (u, v) do
5       dist[v] = min(dist[v], dist[u] + cost(u, v))
```

Der Algorithmus geht so vor, dass er $|V|-1$ mal alle Kanten des Graphes durchgeht und so iterativ die Minimalkosten aufsummiert. Die Verwendung einer Adjazenzmatrix bietet sich an – die Kosten können dort gleich in-place eingetragen werden. Die Verwendung einer Adjazenzliste bietet keine Vorteile, die Verwendung eines Heaps ist hier sinnlos, da jedenfalls über alle Kanten iteriert werden muss.

8.6 Floyd-Warshall Algorithmus für die transitive Hülle

Der Graph $G'(V, E')$ ist transitive und reflexive Hülle des Graphen $G(V, E)$, wenn gilt $(v, w) \in E'$, also wenn es einen Pfad von v nach w gibt.

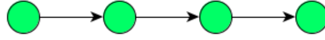

Abbildung 35: Graph Gerold

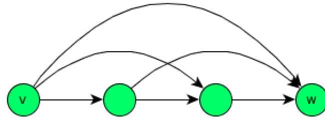

Abbildung 36: Transitive Hülle Gerolds

```
1 transitive-closure(graph):
2   for k = 0 ... N do
3     for i = 0 ... N do
4       for j = 0 ... N do
5         graph[i][j] = a[i][j] | (a[i][k] & a[k][j])
```

Dieser Algorithmus prüft für alle Knotenpaare (i, j) ob es einen Weg von i nach j durch k gibt. In jeder Iteration k entsprechen die bis dato gefundenen Kanten den Wegen, deren innere Knoten alle $\leq k$ sind. Dementsprechend wurden im Durchlauf $|V| - 1$ alle existenten Wege gefunden.

8.7 Floyd-Warshall Algorithmus für kürzeste Wege

Die Vermutung liegt nahe, dass wir diesen Algorithmus zur Findung von kürzesten Wegen missbrauchen können. Tatsächlich müssen wir nur beachten, dass die Kantenkosten minimal bleiben.

```
1 for k = 0 ... N do
2   for i = 0 ... N do
3     for j = 0 ... N do
4       cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j])
```

Sollte der Graph einen negativen Zyklus enthalten enthält die Kostenmatrix nach Ablauf des Algorithmus einen oder mehr Knoten mit Kosten < 0 in der Diagonale.

8.8 Minimale Spannende Bäume

Ein spannender Baum ist ein Teilgraph T' wobei $V_{T'} = V_G$, also alle im Originalgraphen enthaltene Knoten sind auch im Teilgraphen enthalten, und T' ein Baum ist, also keine Zyklen enthält und verbunden ist. Ein minimal spannender Baum ist zuzüglich minimal bezüglich der Kostensumme aller Kanten.

8.8.1 Algorithmus von Kruskal

Der Algorithmus von Kruskal verwendet als Datenstruktur das *Union-Find*. Es besteht ein Set, das zunächst leer ist, in das graduell alle Kanten, die vom Algorithmus als „sicher“, also minimal und nicht zyklusbildend, befunden wurden, hinzugefügt werden. Gleichzeitig werden die Mengen (Wälder) im Union-Find miteinander vereinigt, was dem Zusammenschluss von Komponenten entspricht. Am Ende liefert der Algorithmus eine Menge (Set) von Kanten, die gemeinsam den minimalen spannenden Baum bilden. Die Kanten müssen zuerst sortiert werden.

```
1  kruskal(graph):
2    MSB = set()
3    for each vertex v do
4      make-set(v)
5    E = graph.edges
6    sort(E)
7    for each edge (u, v) in E do
8      if(u and v belong to different sets)
9        add (u, v) to MSB
10       union(u, v)
11   return MSB
```

Union-Find

Union-Find ist eine Datenstruktur zur Verwaltung von disjunkten Mengen. Folgende Operationen werden unterstützt:

- `make-set(x)`: macht eine Singleton-Menge
- `find-set(x)`: liefert die Repräsentante für ein Element, determiniert also, zu welcher Menge `x` gehört
- `union(x, y)`: vereinigt zwei Mengen romantisch miteinander

Zur Darstellung eignet sich ein simples Array (wenn Knoten als Arrayindize dargestellt werden können) oder ein Dictionary, welche die jeweiligen Elterknoten speichern. Ein Union-Find verfügt also, ähnlich wie ein Heap, über eine implizite Baumstruktur, wird aber durch ein Feld dargestellt. Hier ein Beispiel. Das parent-Array ist mit $A, B, C, \ldots, F$ indiziert.

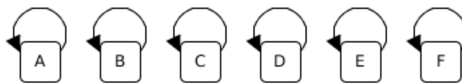

Abbildung 37: Ausgangszustand nach Aufruf von `make-set` für alle Knoten
`parent: [A, B, C, D, E, F]`

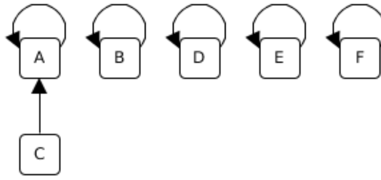

Abbildung 38: Nach `union(A, C)`
`parent: [A, B, A, D, E, F]`

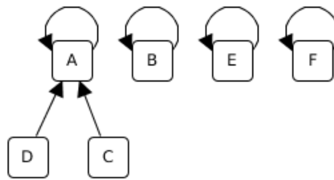

Abbildung 39: Nach `union(A, D)`
`parent: [A, B, A, A, E, F]`

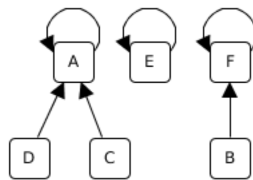

Abbildung 40: Nach `union(F, B)`
`parent: [A, F, A, A, E, F]`

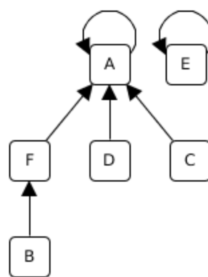

Abbildung 41: Nach `union(D, B)`
`parent: [A, F, A, A, E, A]`

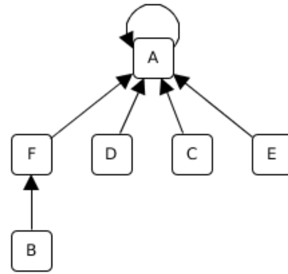

Abbildung 42: Nach `union(C, E)`
`parent: [A, F, A, A, A, A]`

Das hier gezeigte Beispiel ist ein schöner Fall, es muss jedoch beachtet werden, dass die Union-Find-Bäume im schlimmsten Fall (der recht leicht eintritt, wenn man immer wieder an Singletons anhängt) zu Listen der Länge n entarten, was die Perfomance drückt. Um dies zu vermeiden, gibt es zwei leichte, aber extrem effiziente Abwandlungen die man in `union` und `find-set` machen kann.

Union by Rank

Hierbei verwendet man eine zusätzliche Variable `rank`, die die Tiefe eines Elterknotens angibt. Die Wurzeln im Union-Find-Wald haben jeweils `rank = 0`, die unmittelbaren Nachfahren haben `rank = 1`, etc. Damit erreicht man, dass der kleinere Baum Kind des größeren wird – man vermeidet also ein unregelmäßiges Wachstum.

```
link(x,y):
  if rank[y] > rank[x]
    parent[x] = y
  else
    parent[y] = x
    if rank[x] == rank[y]
      rank[x]++
```

Path compression

Path compression bezeichnet das Hochziehen von Knoten, sodass sie direkte Kinder der Wurzel werden.

```
find-set(x):
  if x != p[x]
    p[x] = find-set(p[x])
  return p[x]
```

Beide dieser Verbesserungen führen dazu, dass Kruskals Algorithmus nun nurnoch von der Ordnung $O(|E| \log |E|)$ ist, was dem Sortieraufwand der Kanten entspricht.

Ein Beispielgraph bearbeitet mit Kruskals Algorithmus

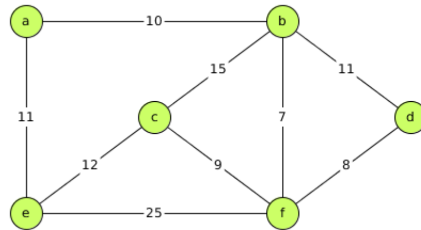

Abbildung 43: Beispielgraph Bartholomäus

Zuerst sortieren wir die Kanten des Graphen.

```
E = [(f, b, 7),
     (d, f, 8),
     (c, f, 9),
     (a, b, 10),
     (a, c, 11),
     (b, d, 11),
     (e, c, 12),
     (b, c, 15),
     (f, c, 25)]
```

Nun bearbeiten wir die Kanten aufsteigend. Wir fügen die Kanten hinzu, wenn die beiden beteiligten Knoten noch nicht in der selben Menge (Union-Find) enthalten sind. Dann vereinigen wir die beiden Mengen mittels Union-Find. Am Ende erhalten wir die MSB-Menge

$$MSB = \{\ (f, b, 7),\ (d, f, 8),\ (c, f, 9),\ (a, b, 10),\ (a, e, 11)\ \}$$

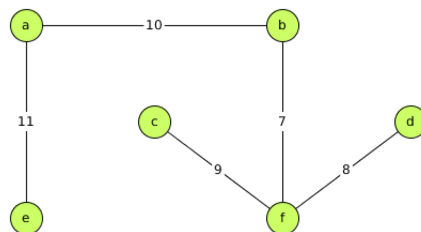

Abbildung 44: MSB von Bartholomäus

8.8.2 Algorithmus von Prim

Prims Algorithmus ist ähnlich dem von Kruskal, jedoch ist es nicht notwendig, die Kanten zu sortieren. Die minimalste Kante wird mittels dem Einsatz einer Priority Queue ermittelt.

```
prim(graph, start):
  MSB = set()
  queue = graph.vertices
  initialze dist[u] = MAX for all u in graph.vertices
```

```
5  dist[start] = 0
6  predecessor[start] = nil
7  while queue is not empty
8    u = queue.remove_min()
9    if(predecessor[u] is not nil)
10     add (predecessor[u], u) to MSB
11   for every vertix v adj. to u
12     if v in queue and cost(u, v) < dist[v]
13       predecessor[v] = u
14       decrease_key(v, cost(u, v))
```

Die Queue muss die Operation `decrease_key` unterstützen, die einen zugeordneten Wert verringert und ggf. die Min-Heap-Bedingung wiederherstellen muss. Prims Algorithmus hat eine Komplexität von $O(m \log n)$.

Ein Beispielgraph bearbeitet mit Prims Algorithmus

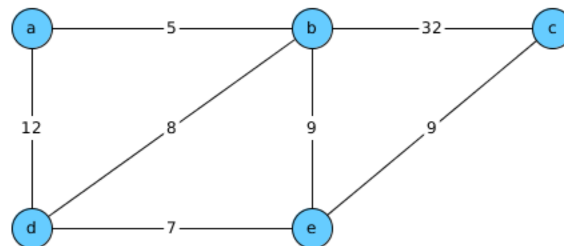

Abbildung 45: Beispielgraph Bernd

Wir beginnen damit, dass wir uns einen zufälligen Startknoten wählen. Wir nehmen hierzu Knoten a. Wir betrachten die minimalste ausgehende Kante – diese führt zu Knoten b. Wir fügen die Kante $(a, b, 5)$ in unseren MSB ein. Nun suchen wir die minimalste Kante, die von a oder b zu einem Knoten führt, der noch nicht in der MSB-Menge enthalten ist. Wir wählen daher die Kante von b nach d. Wir fügen die Kante $(b, d, 8)$ zur MSB-Menge hinzu. Die nächste minimale Kante die einen neuen Knoten beisteuert ist die Kante von d nach e. Wir fügen $(d, e, 7)$ zu unserem MSB hinzu. Der letzte Knoten, der noch fehlt, ist c. Wir fügen also die Kante $(e, c, 9)$ hinzu. Somit sind alle Knoten des Graphen in der MSB-Menge, die folgendermaßen aussieht:

$$MSB = \{\ (a, b, 5),\ (b, d, 8),\ (d, e, 7),\ (e, c, 9)\ \}$$

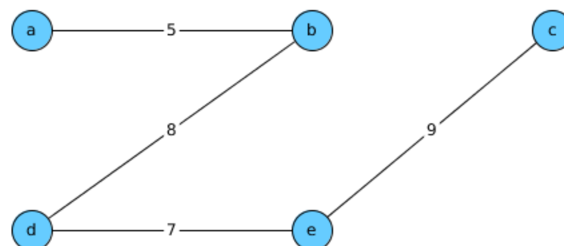

Abbildung 46: Bernd ganz gelöst

A Big-O Tabellen

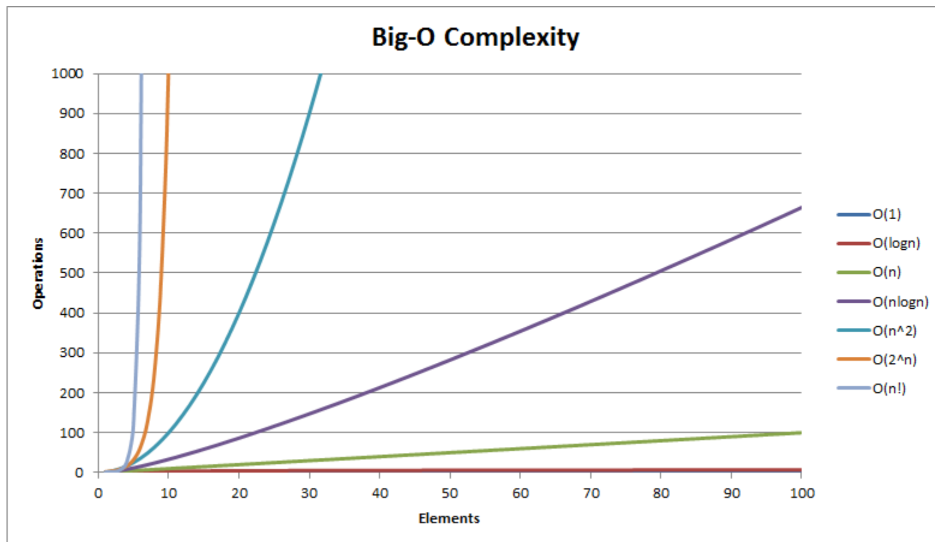

Abbildung 47: Wachstumsraten graphisch dargestellt

A.1 Datenstrukturen

Best case

	Random Access	Suche	Einfügen	Entfernen
Array	O(1)	O(n)	O(n)	O(n)
Dictionary	–	O(1)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(n)
Stack	O(1)	–	O(1)	O(1)
Binärsuchbaum	O(log n)	O(log n)	O(log n)	O(log n)
B-Baum	O(log n)	O(log n)	O(log n)	O(log n)

Worst case

Dictionary entartet zu einem “Array” (je nach Implementation auch zu einer Liste o.ä.), Binärsuchbaum entartet zu einer linearen Liste.

	Random Access	Suche	Einfügen	Entfernen
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Dictionary	–	$O(n)$	$O(n)$	$O(n)$
Singly-linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Stack	$O(n)$	–	$O(1)$	$O(1)$
Doubly-linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Binärsuchbaum	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
B-Baum	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

A.2 Sortieralgorithmen

	Laufzeit			Speicher
	Best	Average	Worst	Worst
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
Radixsort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

A.3 Graphen

	Adjazenzliste	Adjazenzmatrix
Depth First Search	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$
Breadth First Search	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$
Dijkstra	$O(\|E\| \log \|V\|)$ (Heap!)	$O(\|V\|^2)$
Bellman-Ford	$O(\|V\| * \|E\|)$	$O(\|V\| * \|E\|)$
Floyd-Warshall	$O(\|V\|^3)$	$O(\|V\|^3)$
Kruskal	$O(\|E\| \log \|E\|)$	$O(\|E\| \log \|E\|)$
Prim	$O(\|E\| \log \|V\|)$ (Heap!)	$O(\|V\|^2)$