Algorithmen und Datenstrukturen
Überblick Test 1

AvocadoNyanCat

2022S

Dies ist bloß dazu gedacht, einen groben Überblick über die Inhalte zu geben und dient nicht als Ersatz für die Folien/VOs. Keine Garantie auf Korrektheit oder Vollständigkeit. Auch Rechtschreib- und/oder Grammatikfehler sind möglich und sogar wahrscheinlich, da hier so einige schlaflose Nächte enthalten sind.

Zudem ist es stark angelehnt und als Ergänzung zur Zusammenfassung von Patrick gedacht, welche ich sehr empfehlen kann.

Contents

1 Analyse von Algorithmen **2**
1.1 Asymptotisches Wachstum . . . 2
1.2 Asymptotische Dominanz . . . 2
1.3 Laufzeiten einiger gebräuchlicher Funktionen . . . 2

2 Graphen **3**
2.1 Grundlegende Definitionen . . . 3
2.2 Repräsentation von Graphen . . . 3
2.3 Kantenzüge und Pfade . . . 4
2.4 Durchmusterung von Graphen . . . 5
2.5 Teilgraphen und Zusammenhangskomponenten . . . 6
2.6 DAGs, Topologische Sortierung und Dijkstra . . . 7
2.7 Heaps, Heapify Up und Heapify Down . . . 8

3 Greedy-Algorithmen **10**
3.1 Definition und Optimalität . . . 10
3.2 Intervall Scheduling . . . 10
3.3 Minimaler Spannbaum . . . 11

4 Divide-and-Conquer-Algorithmen **12**
4.1 Grundlegende Definition . . . 12
4.2 Mergesort . . . 12
4.3 Quicksort . . . 13
4.4 Inversionen zählen . . . 14
4.5 Dichtestes Punktpaar . . . 14

5 Suchbäume **14**
5.1 Binäre Suchbäume 14
5.2 AVL-Bäume 16
5.3 B-Bäume und B*-Bäume 18

6 Hashing **20**
6.1 Definitionen und Grundlagen 20
6.2 Hashfunktionen 20
6.3 Kollisionsbehandlung 20

7 Praktische Datenstrukturen **21**
7.1 Datenstrukturen im Java Collection Framework 21
7.2 TimSort 22

1 Analyse von Algorithmen

1.1 Asymptotisches Wachstum

Ist eine Funktion $f(n)$ von einer anderen Funktion $g(n)$ von **oben beschränkt** ist oder anders gesagt $f(n) = O(g(n))$ zutrifft, bedeutet dies, dass ebenso die Ungleichung $g(n) \leq c \cdot f(n)$ gilt, wobei $c > 0$ eine beliebige Konstante ist. Das $O(n)$ steht dabei als "Obere Schranke von n". Daraus folgt auch, dass $f(n)$ asymptotisch schneller wächst als $g(n)$

Analog dazu ist auch für die **untere Beschränktheit** definiert. Bei dieser gilt $g(n) = \Omega(f(n))$, wenn $f(n) \geq c \cdot g(n)$ mit $c > 0$. Auch hierbei ist das $\Omega(n)$ als "Untere Schranke von n" zu verstehen. Der Zusammenhang zwischen den beiden Schranken besteht darin, dass falls $g(n) \leq c \cdot f(n)$ gilt, automatisch auch $g(n) \geq c \cdot f(n)$ gilt.

Trifft wiederum $g(n) = O(f(n))$ und $g(n) = \Omega(f(n))$ zur gleichen Zeit zu, spricht man von **asymptotisch gleichem Wachstum** bzw. einer **scharfen Schranke**, $g(n) = \Theta(f(n))$.

1.2 Asymptotische Dominanz

Wird $f(n)$ von $g(n)$ dominiert, wenn $f(n) = O(g(n))$ aber nicht $g(n) = O(f(n))$ gilt oder wenn $lim_{n\to\infty} f(n)/g(n) = 0$ Man schreibt dann $f(n) \ll g(n)$. Hierbei gelten außerdem folgende Verhältnisse (mit $c > 1$, $k > 3$ und $0 < \epsilon < 1$):

$1 \ll log(n) \ll \sqrt{n} \ll n \ll n \cdot log(n) \ll n^{1+\epsilon} \ll n^2 \ll n^3 \ll n^k \ll c^n \ll n! \ll n^n$

1.3 Laufzeiten einiger gebräuchlicher Funktionen

(i) **Konstantes Wachstum:** $O(1)$ - Unabhängig von der Eingabegröße n ist der Aufwand für eine Operation konstant.

(ii) **Logarithmisches Wachstum:** $O(log(n))$ - Die Eingabegröße wird in jedem Schritt halbiert, gedrittelt, etc wird. (Wie beispielsweise bei der binären Suche)

(iii) **Lineares Wachstum:** $O(n)$ - Laufzeit ist proportional zur Eingabegröße. (Hier wäre die Maximumsuche ein passendes Beispiel oder die Verschmelzung (merge) zweier Listen)

(iv) **Logarithmisch-Lineares/Leicht Überlineares Wachstum:** $O(n \cdot log(n))$ - Üblicherweise die Laufzeit von "Divide-and-Conquer"-Algorithmen. (Wie etwa bei Mergesort)

(v) **Quadratisches Wachstum:** $O(n^2)$ - Meistens zwei Schleifen, die verschachtelt sind

(vi) **Kubisches Wachstum:** $O(n^3)$ - Meistens drei Schleifen, die verschachtelt sind

(vii) **Polynomielle Laufzeit:** $O(n^k)$ - Meistens mehr als drei Schleifen, die verschachtelt sind

(viii) **Exponentielles Wachstum:** $O(c^n)$ - Eine endliche Menge mit n Elementen hat genau 2n Teilmengen und jede dieser Teilmengen muss untersucht werden. (Teilsummenproblem)

2 Graphen

2.1 Grundlegende Definitionen

Graphen sind ein hilfreiches Werkzeug, um Zusammenhänge oder auch Strukturen zu modellieren. Sind zwei Knoten u und v einer Kante $e = (u, v)$ **adjazent**, bedeutet dies, dass u ein **Nachbar** von v ist und umgekehrt. Liegen mehrere Kanten zwischen einem Knoten, werden diese **Mehrfachkanten** genannt. **Schleifen** sind Kanten, die einen Knoten mit sich selbst verbinden. Enthält ein ungerichteter Graph weder Mehrfachkanten, noch Schleifen, wird er ein **schlichter Graph** genannt. Er könnte beispielsweise wie folgt aussehen (linker Graph):

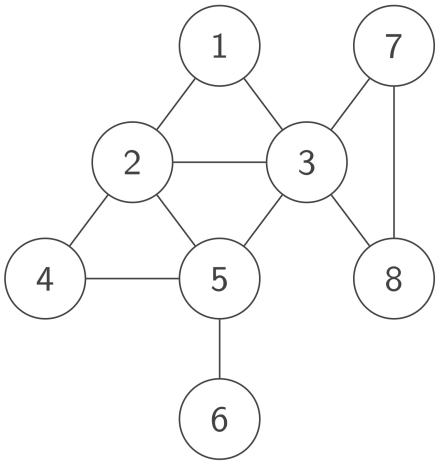


Figure 1: ungerichteter Graph

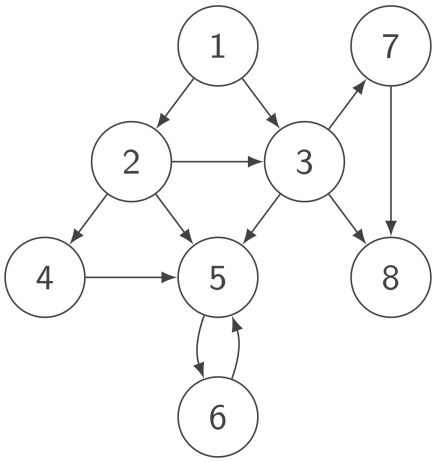


Figure 2: gerichteter Graph

Zudem heißen u oder auch v **inzident** zur Kante e, wenn sie mit dieser benachbart sind. Handelt es sich um einen ungerichteten Graphen, entspricht der **Knotengrad** von v ($deg(v)$) der Anzahl der inzidenten Kanten von diesem. Zudem gilt das Handschlaglemma:
Handschlaglemma: $\sum_{v \in V} deg(v) = 2 \cdot |E|$

Gerichtete Graphen besitzen hingegen sowohl einen **Eingangsknotengrad** $deg(v)^-$, als auch einen **Ausgangsknotengrad** $deg(v)^+$. Dabei gilt $deg(v) = deg(v)^- + deg(v)^+$

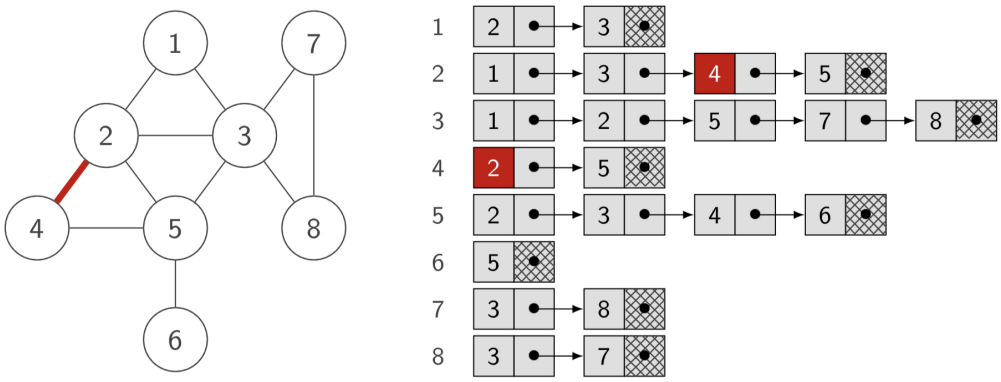
2.2 Repräsentation von Graphen

Es gibt verschiedene Arten, adjazenz von Graphen darzustellen. Abhängig von der Art des Graphen, ist die eine oder andere Datenform sinnvoller. Zu diesem Zweck, ist es wichtig zu wissen, dass es sich um einen **dichten (dense) Graphen** handelt, wenn der Graph bis zu $m = \Theta(n^2)$ Kanten hat und um einen **lichten (sparse) Graphen**, wenn er bis zu $m = O(n)$ Kanten besitzt.

Will man einen lichten Graphen darstellen, ist eine **Adjazenzliste** von größerem Vorteil. Sie speichert im jeweils v-ten Arrayeintrag mit welchen Knoten v adjazent ist und ggf. dessen Gewicht mittels einer

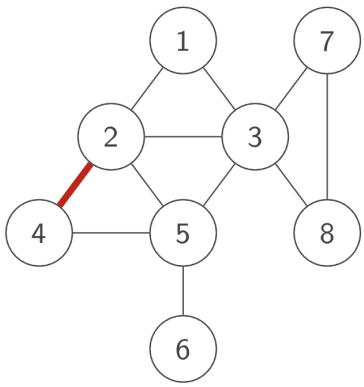
LinkedList.
Zudem hat sie einen **Platzbedarf** von $\Theta(m+n)$. Möchte man **überprüfen, ob eine Kante darin enthalten ist**, wird eine Laufzeit von $O(deg(u))$ benötigt. Eine **Aufzählung aller Kanten** hat hingegen eine Laufzeit von $\Theta(m+n)$.

Die Adjazenzliste sieht dann wie folgt aus:



Zusätzlich gibt es **Adjazenzmatrizen**, welche man bei dichten Graphen verwenden kann, aber auch bei lichten. Handelt es sich hierbei um einen gewichteten Graphen, nehmen die Einträge reele Werte an. Ansonsten enthält sie Bool'sche Werte.
Ihr **Platzbedarf** liegt in $\Theta(n^2)$. Die **Abprüfung, ob eine Kante darin enthalten ist** benötigt eine konstante Laufzeit. Auch die **Aufzählung aller Kanten**, muss man mit einer Laufzeit von $\Theta(n^2)$ rechnen.

Die Adjazenzmatrix sieht dann wie folgt aus:

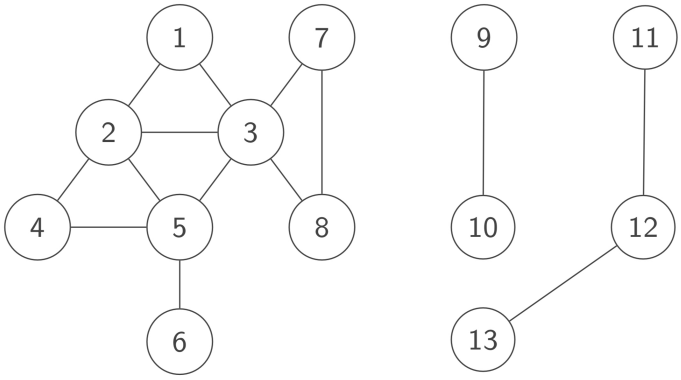


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

2.3 Kantenzüge und Pfade

Nun zurück zu einigen Definitionen, die für ungerichtete Graphen gelten. Ein **Kantenzug** ist eine folge von Knoten, die alle unter einander mit Kanten verbunden sind. Enthält diese keinen Knoten doppelt, so spricht man von einem **Weg/Pfad**. Darüber hinaus heißt ein Knoten u von einem anderen Knoten v aus **erreichbar**, wenn ein Pfad zwischen u und v existiert. Kantenzüge die den selben Anfangs- und Endknoten besitzen, nennt man **Kreise**, Mengen an Kanten, die genau einen Endpunkt in der Knotenteilmenge haben, **Kantenschnittmengen**. Die selben Definitionen besitzen bei gerichteten Graphen die zusätzliche Eigenschaft, dass sie über gerichtete Kanten erfolgen.

Zudem heißt ein ungerichteter Graph **zusammenhängend**, wenn jedes Paar von Knoten von einander erreichbar ist und die **Distanz** zwischen zwei Knoten ist charakterisiert durch die Länge des kürzesten Pfads zwischen diesen. (1-8 sind z.B. zusammenhängend, 9 und 10, 11-13 auch. 1 bis 13 jedoch nicht)



Ist ein ungerichteter (Teil-)Graph zusammenhängend und kreisfrei, wird dieser ein **Baum** genannt. Gehen alle Kanten von einem sogenannten **Wurzelknoten** weg, so nennt man den Graphen einen **Wurzelbaum**.

2.4 Durchmusterung von Graphen

Zwei mögliche Algorithmen zur Graphendurchmusterung wären die Breitensuche und die Tiefensuche, welche beide eine Laufzeit von $O(m+n)$ haben.

Die **Breitensuche (Breadth First Search - BFS)** findet ihre Anwendung beim s-t-Zusammenhangsproblem ("Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?") und sucht den kürzesten Pfad zwischen ihnen, falls es einen solchen gibt. Dies macht er, indem er nach und nach Knoten "findet" bis er zum gesuchten Endknoten gelangt. Dazu wird ein Array *Discovered*, eine Queue Q, Graph $G = (V, E)$ und ein Startknoten s benötigt. Der Algorithmus sieht dann folgendermaßen aus:

```
BFS(G,s):
Discovered[s] ← true
Discovered[v] ← false für alle anderen Knoten v ∈ V
Q ← {s}
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    Führe Operation auf u aus (z.B. Ausgabe)
    foreach Kante (u,v) inzident zu u
        if  !Discovered[v]
            Discovered[v] ← true
            Füge v zu Q hinzu
```

Diese Methode liefert schlussendlich einen **BFS-Baum**, dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet. Diese werden in Ebenen aufgeteilt, die davon abhängen, welche Distanz die jeweiligen Knoten zu s besitzen. Die Ermittlung der Ebene eines einzelnen Knotens ist hierbei auch möglich, indem man die Ebenen iterativ durchläuft. Dazu sind ein Array *Level*, eine Queue Q, ein Graph $G = (V, E)$ und ein Startknoten s nötig und folgender Algorithmus:
Die **Tiefensuche (Depth First Search - DFS)** geht hingegen rekursiv vor und ruft sich bei jedem weiteren Nachbarsknoten selber auf, bevor die weiteren Nachbarsknoten besucht werden. Anders als bei der Breitensuche, wird somit versucht, sich möglichst weit von Startknoten zu entfernen. Gibt es in der Nachbarschaft keine möglichen Knoten, dann geschieht ein Backtracking bis zur letzten möglichen

```
BFS(G,s):
Level[s] ← 0
Level[v] ← -1 für alle anderen Knoten v ∈ V
Q ← s
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    foreach Kante (u,v) inzident zu u
        if  Level[v] == -1
            Level[v] ← Level[u] + 1
            Füge v zu Q hinzu
```

Verzweigung. Hier sind bloß Startknoten s, ein globales Array *Discovered* und ein Graph $G = (V, E)$ notwendig, welche dieser Algorithmus dann in Anspruch nimmt:

```
DFS(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
Führe Operation auf u aus (z.B. Ausgabe)
foreach Kante (u,v) inzident zu u
    if  !Discovered[v]
        DFS1(G,v)
```

2.5 Teilgraphen und Zusammenhangskomponenten

Ein Graph T wird **Teilgraph** von G genannt, wenn die Kantenmenge von T eine Teilmenge der Kantenmenge von G ist und die Knotenmenge von T eine Teilmenge der Knotenmenge von G.
Eine **Zusammenhangskonponente** ist der maximale zusammenhängende Teilgraph eines ungerichteten Graphen. Möchte man herausfinden, welche Knoten zu einer solchen gehören, reicht ein Aufruf des DFS oder BFS. Ist das Ziel hingegen, die Anzahl der Zusammenhangskonponenten herauszufinden, wäre beispielsweise der **DFSNUM-Algorithmus** hilfreich, dessen Laufzeit in $O(m + n)$ liegt. Er benötigt einen Startknoten s, ein globales Array *Discovered* und einen Graph $G = (V, E)$ und sieht im wesentlichen so aus:

```
DFSNUM(G):
Discovered[v] ← false für alle Knoten v ∈ V
i ← 0
foreach Knoten v ∈ V
    if Discovered[v] = false
        i ← i + 1
        DFS1(G,v)
return i
```

Bei gerichteten Graphen besteht zusätzlich die Unterscheidung zwischen schwachen und starken Zusammenhangskonponenten. Auch die Erreichbarkeit von Knoten ist leicht anders definiert. Zwei Knoten u und v sind **gegenseitig erreichbar**, wenn es sowohl einen Pfad von v zu u als auch von u zu

v gibt. **Starke Zusammenhangskomponenten** bauen auf dieser Definition auf, es muss nämlich jedes Paar von Knoten von einander aus erreichbar sein. **Schwache Zusammenhangskomponenten** hängen hingegen unter Missachtung der Richtung mit einander zusammen bzw. würden im zugehörigen ungerichteten Graphen zusammenhängen. Auch hier existiert ein Algorithmus, der dazu dient, herauszufinden, ob ein Graph G stark zusammenhängend ist. Die Laufzeit von diesem liegt in $O(m+n)$.

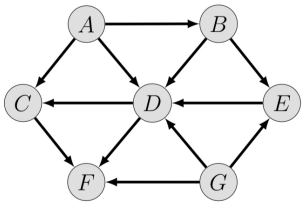
2.6 DAGs, Topologische Sortierung und Dijkstra

Was ist ein DAG? Das steht für **Directed Acyclic Graph** oder auch übersetzt **Gerichteter azyklischer Graph**. Besitzt ein Graph eine topologische Sortierung, handelt es sich dabei garantiert um einen solchen. Umgekehrt hat auch jeder DAG eine topologische Sortierung. Folgender Algorithmus genau dies zu nutze, indem er wiederholt Kanten löscht und sie neu anordnet, bis sich eine topologische Sortierung ergibt, oder feststeht, dass es sich bei der Eingabe um keinen DAG handelt:

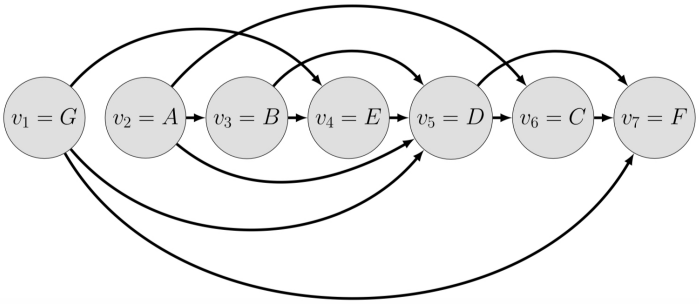
```
while G hat mindestens einen Knoten
    if G hat eine Quelle
        Wähle eine Quelle v aus
        Gib v aus
        Lösche v und alle inzidenten Kanten aus G
    else return G ist kein DAG
return G ist ein DAG
```

Zusätzlich ist es relevant zu wissen, dass ein Knoten ohne eingehenden Kanten auch **Quelle** genannt wird und jeder DAG eine solche enthält. Die sogenannte **Topologische Sortierung** kümmert sich dabei darum, dass die Kanten und Knoten im Graphen von ihrer Quelle aus so sortiert werden, dass eine lineare Ordnung entsteht. Was das bedeutet, ist hier gut zu erkennen:

Ein DAG:



Eine topologische Sortierung:



Es gibt jedoch auch einen weiteren Löschalgorithmus, welcher dazu dient, eine topologische Sortierung in $O(m+n)$ Laufzeit herauszufinden. Dieser simuliert nämlich das Löschen von Knoten mittels einem Hilfsarray *count* und beginnt mit der anfangs leeren Liste L:

```
foreach v ∈ V
    count[v] ← 0
foreach v ∈ V
    foreach Kante (v,w) ∈ E
        count[w] ← count[w]+1
foreach v ∈ V
    if count[v] = 0
        Gib v zur Liste L am Anfang hinzu
while L ist nicht leer
    Sei v erstes Element in L, lösche v aus L
    Gib v aus
    foreach Kante (v,w) ∈ E
        count[w] ← count[w]−1
        if count[w] = 0
            Gib w zur Liste L am Anfang hinzu
```

Nun bleibt noch der **Dijkstra Algorithmus**, welcher den Zweck hat, den kürzesten gerichteten Pfad positiver Länge zu finden. Dieser arbeitet so, dass er sich die Distanz von Startknoten zu allen benachbarten Knoten ansieht und den Knoten mit der kürzesten Länge auswählt. Möchte man den Dijkstra-Algorithmus mit einer **LinkedList** implementieren (Laufzeit $O(log(n))$), benötigt man zusätzlich zur Liste L zwei Arrays $Discovered$ und d, einen Graphen $G = (V, E)$, einen Startknoten s und folgendes "Kochrezept":

```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
L ← V
while L ist nicht leer
    wähle u ∈ L mit kleinstem Wert d[u]
    lösche u aus L
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            d[v] ← min(d[v], d[u]+ℓ_e)
```

Die zweite Möglichkeit der Implementierung würde mittels einer **Priotity Queue** erfolgen und ist auch deutlich effizienter (Laufzeit $O(log(n))$). Dies würde dann statt der Liste L eine Vorrangwarteschlange Q benötigen, in der die Knoten v nach dem Wert d[v] geordnet sind. Es läuft dann folgendermaßen ab:

2.7 Heaps, Heapify Up und Heapify Down

Beim letzten Teil der Graphentheorie geht es um **(Min-)Heaps**, welche geordnete, binäre Wurzelbäume sind. Dies hat den Vorteil, dass man die Knoten des Baumes ebenfalls in einem Array speichern kann.

```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
Q ← V
while Q ist nicht leer
    wähle u ∈ Q mit kleinstem Wert d[u]
    lösche u aus Q
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            if d[v] > d[u]+ℓ_e
                lösche v aus Q
                d[v] ← d[u]+ℓ_e
                füge v zu Q hinzu
```

Ist von **Heapify-Up** die Rede, wird ein Algorithmus gemeint, der die Einträge nach oben vertauscht, sobald die Ordnung durch das Einfügen eines neuen Knotens verletzt wird. Hier ist ein Algorithmus, welchem dies mit einer Laufzeit von $O(log(n))$ gelingt:

```
Heapify-up(H,i):
if i > 1
    j ← ⌊i/2⌋
    if H[i] < H[j]
        Vertausche die Array-Einträge H[i] und H[j]
        Heapify-up(H,j)
```

Analog dazu gibt es auch die Möglichkeit, es nach unten zu sortieren. Dies nennt sich dann **Heapify-Down** und wird zur Sortierung nach dem Löschen eines Knotens verwendet. Der Algorithmus dazu funktioniert wie folgt:

```
Heapify-down(H,i):
n ← length(H)-1
if 2·i > n
    return
elseif 2·i < n
    left ← 2·i, right ← 2·i+1
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ← 2·i
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H,j)
```

Zusätzlich gibt es noch weitere Heap-Operationen wie Insert ($O(log(n))$), FindMin ($O(1)$), Delete ($O(log(n))$ und ExtractMin ($O(log(n))$. Das Erstellen eines Heaps aus einem Array A der Länge n, was noch keine Heap-Eigenschaften erfüllt, liegt außerdem in $O(n)$ und benötigt folgenden Algorithmus:

```
Init(A,n):
for i = ⌊n/2⌋ bis 1
    Heapify-down(A,i)
```

3 Greedy-Algorithmen

3.1 Definition und Optimalität

Das markante Merkmal an **Greedy-Algorithmen**, ist dass sie immer die beste lokale Lösung heranziehen und so schrittweise eine Gesamtlösung aufbauen. Bereits getätigte Entscheidungen werden jedoch nicht wieder zurückgenommen. Somit sind sie zwar einfach zum implementieren, aber führen nicht immer zum optimalen Ergebnis (Gegenbeispiel: Der Geldwechsel Algorithmus liefert bei $S = 40$ die Lösung 1 x 25, 1 x 10, 1 x 5 statt 2 x 20).

3.2 Intervall Scheduling

Bei diesem Beispiel besteht das Problem darin, die jeweils **größte Menge an kompatiblen (sich nicht überlappenden) Teiljobs** zu finden. Job j startet dabei zum Zeitpunkt s_j und endet zum Zeitpunkt f_j.
Der **klassische Greedy-Algorithmus** besteht nun darin, die Jobs in einer natürlichen Ordnung zu betrachten und einen Job zu wählen, wenn er kompatibel mit den bisher gewählten Jobs ist. Zusätzlich gibt es auch folgende Variationen/Strategien davon:

i **Früheste Startzeit**: Berücksichtige Jobs in aufsteigender Reihenfolge von s_j.

ii **Früheste Beendigungszeit**: Berücksichtige Jobs in aufsteigender Reihenfolge von f_j.

iii **Kleinstes Intervall**: Berücksichtige Jobs in aufsteigender Reihenfolge von $s_j - f_j$.

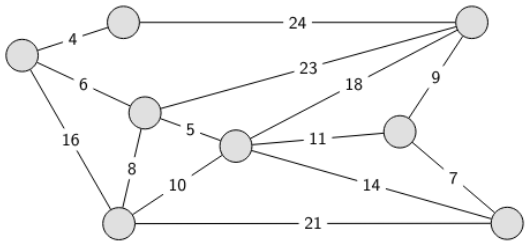
iv **Wenigste Konflikte**: Zähle für jeden Job j die Anzahl c_j der nicht kompatiblen Jobs. Berücksichtige Jobs in aufsteigender Reihenfolge von c_j.

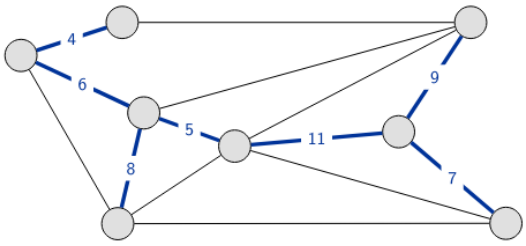
Diese Gegenbeispiele zeigen jeweils, welche der Greedy-Strategien fallweise keine optimale Lösung liefern. Die Strategie **"früheste Beendigungszeit"** liefert hingegen immer eine solche in einer Laufzeit von $O(n \cdot log(n))$ und sieht folgendermaßen aus:

```
Sortiere Jobs nach Beendigungszeit, sodass f_1 ≤ f_2 ≤ ··· ≤ f_n
A ← ∅
for j ← 1 bis n
    if Job j ist kompatibel zu A
        A ← A ∪ {j}
return A
```

3.3 Minimaler Spannbaum

Ist ein **zusammenhängender, schlichter Graph** mit reellwertigen Kantengewichten gegeben, ist sein **minimaler Spannbaum (Minimum Spanning Tree, MST)** ein Teilgraph, der alle Knoten enthält, aber nur jene Kanten, sodass er ein aufspannender Baum mit minimaler Summe der Kantengewichte ist. Hier ist ein passendes Beispiel dazu:





Da sich das Finden von einem solchen bei **exponentiell vielen Möglichkeiten** durch "Herumprobieren" als ineffizient erweisen kann, wurden zwei Algorithmen erstellt, die dafür sorgen, dass es in polynomieller Zeit lösbar ist. Einer davon wäre der **Algorithmus von Prim**, welcher besonders für **dichte Graphen** besser geeignet ist. Er fängt mit einem beliebigen Startknoten an und fügt anschließend in jedem Schritt eine billigste Kante zum MST hinzu, die keinen Kreis erzeugt. Seine Laufzeit liegt je nach Implementierung in $O(n^2)$ (**mittels Array**) oder $O(m \cdot log(n))$ (**mittels Min-Heap**). Hier ist noch der Pseudocode dazu:

```
Prim(G, c):
foreach (v ∈ V)
    A[v] ← ∞
Initialisiere eine leere Priority Queue Q
foreach (v ∈ V)
    Füge v in Q ein
S ← ∅
while Q ist nicht leer
    u ← entnehme minimales Element aus Q
    S ← S ∪ {u}
    foreach Kante e = (u, v) inzident zu u
        if v ∉ S und c_e < A[v]
            Verringere die Priorität A[v] auf c_e
```

Zudem gibt es noch den **Algorithmus von Kruskal**, welcher mit einem leeren MST startet und daraufhin die Kanten in aufsteigender Reihenfolge ihrer Kosten betrachtet. Abschließend fügt er nur jene Kante zum MST, die dadurch kein Kreis erzeugen. Wegen dem Sortieren ist hierbei die Laufzeit immer $O(m \cdot log(n))$. Somit ist er besser für **dünne/lichte Graphen** geeignet. Diese Vorgehensweise wird dabei verwendet:

```
Kruskal(G, c):
Sortiere Kantengewichte so, dass c_1 ≤ c_2 ≤ ··· ≤ c_m
T ← ∅
foreach (u ∈ V) erzeuge eine einelementige Menge mit u
for i ← 1 bis m
    (u, v) = e_i
    if  u und v sind in verschiedenen Mengen
        T ← T ∪ {e_i}
        Vereinige die Mengen mit u und v
return T
```

sind u und v in unterschiedlichen Zusammenhangskomponenten?
Vereinige zwei Komponenten

Zuletzt bleiben noch einige Lemmata, die sich bei diesem Thema als hilfreich erweisen könnten:

i **Kantenschnittlemma**: Sei S eine beliebige Teilmenge von Knoten und sei e die minimal gewichtete Kante mit genau einem Endknoten in S. Dann enthält der MST die Kante e.

ii **Kreislemma**: Sei C ein beliebiger Kreis und sei f die maximal gewichtete Kante in C. Dann enthält der MST f nicht.

iii **Paritätslemma**: Ein beliebiger Kreis und eine beliebige Kantenschnittmenge haben eine gerade Anzahl von Kanten gemeinsam.

4 Divide-and-Conquer-Algorithmen

4.1 Grundlegende Definition

Anders als die Greedy-Algorithmen, teilen **Divide-and-Conquer-Algorithmen** ein Problem in Teilprobleme auf und lösen jedes Teilproblem unabhängig und meist rekursiv. Schlussendlich kombinieren sie die Lösung für die Teilprobleme zu einer Lösung für das ursprüngliche Problem.

4.2 Mergesort

Dem Namen getreu, funktioniert dieser Algorithmus auch so, dass er das Input-Array in zwei Hälften teilt und jede Hälfte rekursiv sortiert. Anschließend werde die zwei Hälften zu einem sortierten Ganzen verschmolzen. Für den rekursiven Teil gilt dabei die **Rekursionsgleichung**:

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C(\lceil n/2 \rceil)}_{\text{linke Hälfte}} + \underbrace{C(\lfloor n/2 \rfloor)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Wobei $C(n)$ die Anzahl der Schlüsselvergleiche (comparisons) in Mergesort bei einer Eingabegröße n ist. Damit lässt sich beweisen, dass die Anzahl der Schlüsselvergleiche und somit auch die Laufzeit des rekursiven Teils in $O(n \cdot log_2(n))$ liegt.

Das Verschmelzen nennt sich **Merging** und wird so implementiert, dass man ein temporäres Array benutzt und beide Listen vom Anfang an durchläuft. Dann führt man die Elemente beider Listen

im Reißverschlussverfahren zusammen und übernimmt dabei jeweils das kleinste Element der beiden Listen. Er liegt dann in $O(n)$ und wird somit von der Laufzeit des Teilen dominiert. Mit einer Laufzeit von $\Theta(n \cdot log(n))$ gilt er als **zeitoptimaler Sortieralgorithmus** (= asymptotisch schnellstmögliche Laufzeit) und sieht dermaßen aus:

```
Merge(A,l,m,r):
i ← l, j ← m+1, k ← l
while i ≤ m und j ≤ r
    if A[i] ≤ A[j]
        B[k] ← A[i], i ← i+1
    else
        B[k] ← A[j], j ← j+1
    k ← k+1
if i > m
    for h ← j bis r
        B[k] ← A[h], k ← k+1
else
    for h ← i bis m
        B[k] ← A[h], k ← k+1
for h ← l bis r
    A[h] ← B[h]
```

4.3 Quicksort

Auch wenn dieser nicht so scheint, ist **Quicksort** auch ein Divide-and-Conquer-Algorithmus. Er teilt das Problem, indem er ein "Pivotelement" x aus einer Folge A auswählt und diese dann ohne x so in zwei Teilfolgen A_1 und A_2 teilt, dass A_1 nur Elemente $\leq x$ und A_2 nur Elemente $\geq x$ enthält. Schlussendlich ruft er diese Teilfolgen rekursiv auf.
Im **Best Case** haben die beiden Teilfolgen immer (ungefähr) die gleiche Länge. Die Höhe des Aufrufbaumes ist folglich $\Theta(log(n))$ und auf jeder Aufrufebene werden $\Theta(n)$ Vergleiche durchgeführt. Somit liegt die Anzahl der Vergleiche und die Laufzeit in $\Theta(n \cdot log(n))$. Im **Worst Case** wird jede (Teil-)Folge wird immer beim letzten (oder immer beim ersten) Element geteilt und resultiert in einer Laufzeit von $\Theta(n^2)$. Dieser Fall kann vermieden werden, indem entweder das **Pivotelement zufällig** wählt oder dafür immer das erste, letzte und mittlere Element ansieht und den **Median als Pivotelement** nimmt.
Der **Average Case** sieht jedoch so aus, dass die Anzahl der Vergleiche und somit auch die Laufzeit in $\Theta(nlogn)$ liegen.

Tabelle: Laufzeit und Vergleiche.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabelle: Zusätzlicher Speicher.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

4.4 Inversionen zählen

Die Anzahl der Inversionen zwischen zwei Rangfolgen sind ein **Ähnlichkeitsmaß**. Hat man zwei Folgen A $(1, 2, ..., n)$ und B $(a_1, a_2, ..., a_n)$, so heißen ihre Songs i und j sind **invertiert**, wenn $i < j$, aber $a_i > a_j$. Dabei würde ein "Divide-and-Conquer"-Algorithmus so vorgehen, er die Liste in zwei Teile teilt und auf jedem von diesem die Inversionen zählt, wobei a_i und a_j sich in unterschiedlichen Hälften befinden. Zum Schluss werden die zwei Hälften in eine sortierte Folge verschmolzen. Dies erfolgt in $O(n \cdot log(n))$.

4.5 Dichtestes Punktpaar

Bei diesem Problem geht es darum von n verschiedenen Punkten in einer Ebene ein Paar mit der **kleinsten euklidischen Distanz** zwischen den beiden Punkten zu finden. Hierbei wird zunächst der Raum so geteilt, dass sich ungefähr $\frac{1}{2}n$ Punkte auf jeder Seite befinden. Anschließend wird ein dichtestes Punktepaar auf jeder Seite und mit je einem Punkt auf beiden Seiten rekursiv gewählt. Zuletzt wird die beste Lösung dieser drei ausgegeben. Dies geschieht in $O(n \cdot log(n))$ wie folgt:

```
Dichtestes Paar(p_1, ..., p_n):
    if n = 1 return ∞

    Berechne Trennlinie L, sodass Punkte auf zwei Hälften
    aufgeteilt werden.

    δ_1  =  Dichtestes Paar(linke Hälfte)
    δ_2  =  Dichtestes Paar(rechte Hälfte)
    δ    =  min(δ_1, δ_2)

    Lösche alle Punkte die weiter als δ von der
    Trennlinie L entfernt sind

    Sortiere die restlichen Punkte nach y-Koordinate.

    Scanne die Punkte in y-Reihenfolge und vergleiche
    die Distanz zwischen jedem Punkt und den nächsten
    11 Nachbarn. Wenn eine dieser Distanzen kleiner als δ
    ist, ändere δ

    return δ.
```

5 Suchbäume

5.1 Binäre Suchbäume

Ein **Binärer Suchbaum** ist ein Wurzelbaum, von dem jeder Knoten x höchstens zwei Kinder y und z besitzt, ein "linkes" $(y.key \leq x.key)$ und ein "rechtes" $(x.key \leq z.key)$ Kind. Die **Tiefe** eines Knotens ist die Anzahl der Knoten, die es von der Wurzel traversieren muss. Die Menge aller Knoten der gleichen Tiefe wird auch **Ebene/Niveau** genannt. Die **Höhe** des Baumes bezeichnet die Länge eines längsten Pfades von Wurzelknoten aus. Unter **Blattknoten** sind Knoten zu verstehen, die selber keine Kinder besitzen. Die restlichen Knoten werden als **innere Knoten** bezeichnet.

Weiteraus gibt es verschiedene Möglichkeiten, den Graphen zu Traversieren. Hier sind einige davon:

i **Inorder**: Behandelt rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum. (Laufzeit - $\Theta(n)$)

ii **Preorder**: Behandelt rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum.

iii **Postorder**: Behandelt rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel.

Zusätzlich sind jene Operationen auf Binären Suchbäumen üblich (Lautzeit - $O(log(n))$ falls balanciert, $O(n)$ falls Liste):

- **Suchen**:

```
Search(p,s):
while p ≠ null und p.key ≠ s
    if s < p.key
        p ← p.left
    else
        p ← p.right
return p
```

- **Minimum**: Solange beim linken Kind weitergehen, bis es keinen linken Nachfolger mehr gibt.
- **Maximum**: Solange beim rechten Kind weitergehen, bis es keinen rechten Nachfolger mehr gibt.
- **Predeccessor**:

```
Predecessor(p):
if p.left ≠ null
    return Maximum(p.left)
else
    q ← p.parent
    while q ≠ null und p = q.left
        p ← q
        q ← q.parent
    return q
```

- **Successor**:

```
Successor(p):
if p.right ≠ null
    return Minimum(p.right)
else
    q ← p.parent
    while q ≠ null und p = q.right
        p ← q
        q ← q.parent
    return q
```

- **Einfügen**:

```
Insert(root, q):
r ← null, p ← root
while p ≠ null
    r ← p
    if q.key < p.key
        p ← p.left
    else
        p ← p.right
q.parent ← r, q.left ← null, q.right ← null
if r = null
    root ← q
else
    if q.key < r.key
        r.left ← q
    else
        r.right ← q
```

- **Entfernen**:

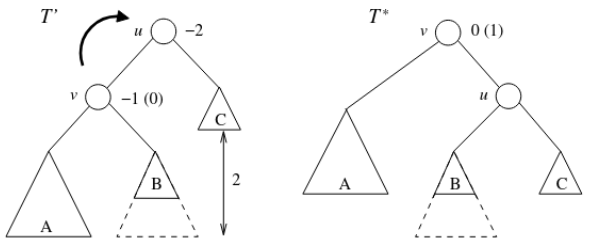
```
Remove(root, q):
if q.left = null oder q.right = null
    r ← q
else
    r ← Successor(q), q.key ← r.key, q.info ← r.info
if r.left ≠ null
    p ← r.left
else
    p ← r.right
if p ≠ null
    p.parent ← r.parent
if r.parent = null
    root ← p
else
    if r = r.parent.left
        r.parent.left ← p
    else
        r.parent.right ← p
```

5.2 AVL-Bäume

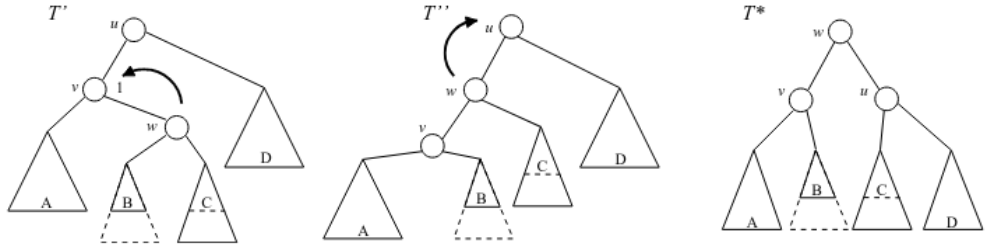
Um einen **Balancierten Baum** zu kreieren, wird der Suchbaum durch Randbedingungen und Umordnung so ausbalanciert, dass seine Höhe logarithmisch bleibt. **Höhenbalancierte Bäume** werden davon charakterisiert, dass die Unterbäume eines Knotens höchstens eine um eine Konstante niedriger/höher sind. **Gewichtsbalancierte Bäume** zeichnen sich hingegen davon aus, dass sich die Knotenanzahl der Unterbäume höchstens um einen konstanten Faktor unterscheidet. Zuletzt bleiben die **(a,b)-Bäume** mit $2 \leq a \leq b$, von welchen jeder Knoten (außer der Wurzel) zwischen a und b Kinder hat und alle Blätter den gleichen Abstand zur Wurzel haben.

Eine Unterart der Höhenbalancierten Bäume sind die sogenannten Adelson-Velski/Landis-Bäume oder auch kurz **AVL-Bäume**. Dadurch, dass eine Höhendifferenz von 1 verlangt wird, wird die Degeneration der Suchbäume verhindern. Formal bedeutet dies, dass ein Knoten **balanciert** ist, wenn $bal(v) = h_2 - h_1$ (mit h_1, h_2 = Höhe des linken/rechten Unterbaums) und $bal(v) \in -1, 0, 1$ gilt. Gerät der Baum nach dem Einfügen eines Knotens aus der Balance, besteht die Möglichkeit der **Rebalancierung**. Diese geschieht durch eine der folgenden vier Rotationen:

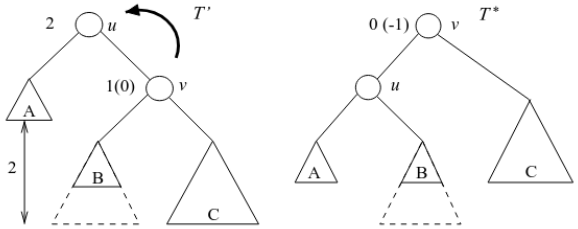
i **einfache Rechtsrotation**:



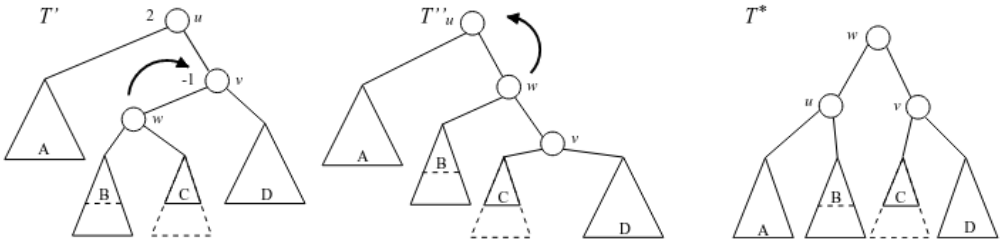
ii **doppelte Links-Rechtsrotation**:



iii **einfache Linksrotation**:



iv **doppelte Rechts-Linksrotation**:

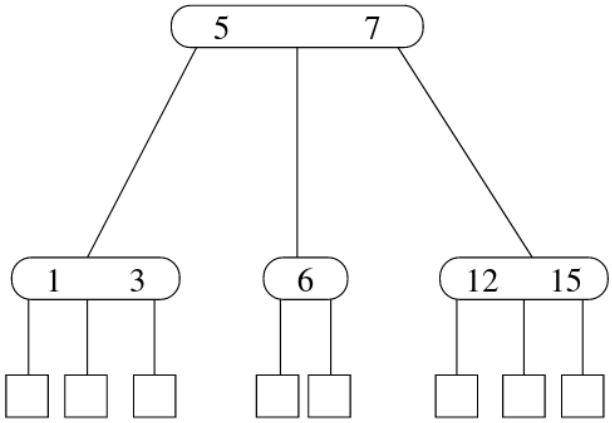


Bezüglich der Laufzeit dieser Operationen kann gesagt werden, dass Rotationen in $O(1)$ liegen und das Einfügen und Entfernen von Knoten in $O(log(n))$. Auch die Größe dieser Baumart liegt immer in $O(log(n))$.

5.3 B-Bäume und B*-Bäume

Genau wie AVL-Bäume eine Unterart der Höhenbalancierten Bäume sind, sind **B-/B*-Bäume** Unterarten der (a,b)-Bäume. Hierbei fasst man innere Knoten zusammen und geht von **leeren Blattknoten gleicher Tiefe** aus. Dabei hat jeder innere Knoten mindestens $\lceil \frac{m}{2} \rceil$ und maximal m Kinderknoten. Außerdem hat jeder Knoten mit l Schlüsseln $l+1$ Kinder und die Werte der Schlüssel sind wie in folgender Abbildung sortiert:

Beispiel: B-Baum der Ordnung $m = 3$ (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.



Die **Suche** ist bei B-Bäumen ein notwendiger Bestandteil des Einfügens und Löschen von Knoten. Dazu benötigt man ein B-Baum mit Wurzel p und ein Schlüssel x als Eingabe. Zusätzlich wäre dieser Algorithmus von Vorteil:

```
Search(p,x):
i ← 1
while i ≤ p.l und x > p.key[i]
    i ← i + 1
if i ≤ p.l und x = p.key[i]
    return (p, i)
if p.l = 0
    return null
else
    return Search(p.child[i − 1], x)
```

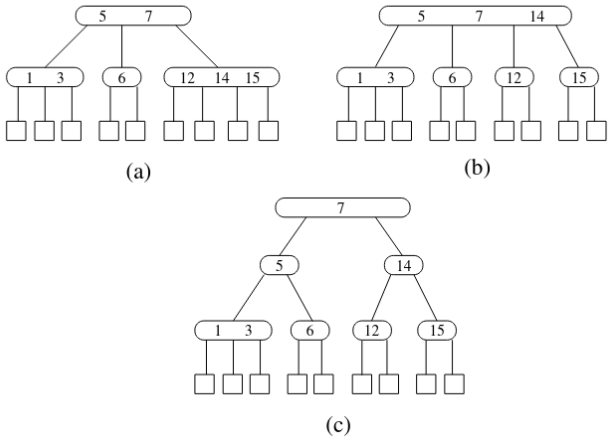
Zum Einfügen und Entfernen kommen wir später. Zunächst sind die **Eigenschaften von B-Bäumen** von Relevanz:

i Die Anzahl der Blätter in einem B-Baum T ist immer um 1 größer als die Anzahl der Schlüssel.

ii Für die Höhe h eines B-Baumes mit N Schlüsseln und $(N+1)$ Blättern muss gelten: B-Bäume haben immer eine Höhe $h = \Theta(log(N))$, somit sind die Operationen Suchen, Einfügen und Entfernen in $\Theta(log(N))$ Zeit durchführbar

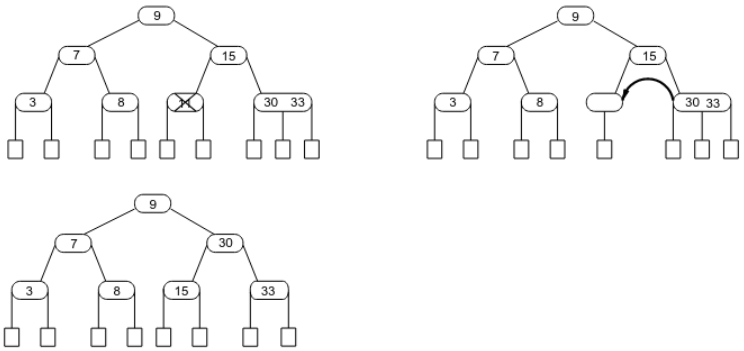
Bezüglich dem **Einfügen von Knoten** bleiben nach dem Finden des Knotens folgende Schritte:

Beispiel: Einfügen von Schlüssel 14.

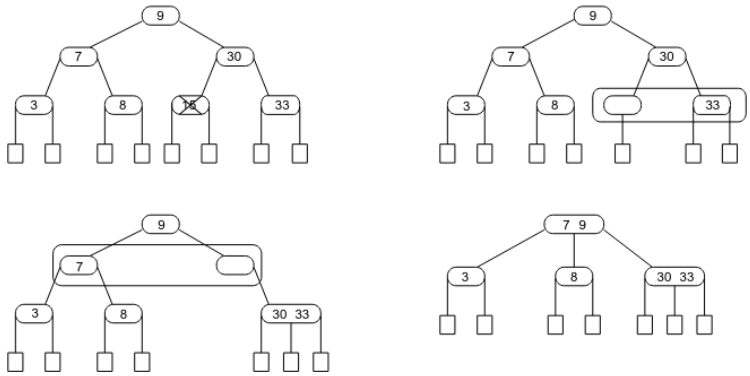


Als letzte Operation verbleibt noch das **Entfernen von Knoten**, welches folgendermaßen funktioniert:

Beispiel: 11 aus B-Baum (Ordnung 3) entfernen.



Beispiel: 15 aus B-Baum (Ordnung 3) entfernen.



Zuletzt bleiben noch die **B*-Bäume**, eine spezielle Variante der B-Bäume, die sich darin unterscheiden, dass sie die kompletten Datensätze in den Blattknoten haben. Die Schlüssel sind dabei immer das größte Element der Liste auf die sie verweisen. Die Operationen **Suchen, Einfügen und Entfernen** besitzen die selbe Laufzeit wie auf B-Bäumen.

6 Hashing

6.1 Definitionen und Grundlagen

Die Grundlegende Idee von **Hashing** besteht darin, dass man Elemente im Speicher durch eine arithmetische Berechnung ermittelt, statt es in einer Menge von Datensätzen durch Schlüsselvergleiche zu suchen. Die **Hashtabelle** T wird dabei durch ein Array der Größe m realisiert. Speichert diese n Schlüssel, so lautet ihr **Belegungsfaktor** $\alpha = \frac{m}{n}$. Die **Hashfunktion** $h(k)$ muss hingegen so gewählt werden, dass sie jedem Schlüssel einen eindeutigen – aber nicht umgekehrt eindeutigen – Hashwert zuordnenen kann. Erhalten zwei Schlüssel den gleichen Hashwert, so nennt sich das **Kollision**. Ohne Kollisionen liegt die Laufzeit der Operationen Suchen, Einfügen und Entfernen in $\Theta(1)$, solange $h(k)$ in konstanter Zeit berechenbar ist. Sonst kann sie sogar in $O(n)$ liegen.

6.2 Hashfunktionen

Die **Divisions-Rest-Methode** erfolgt durch $h(k) = k \mod m$. Dadurch kann die Hashfunktion sehr schnell berechnet werden und m sollte als Primzahl gewählt werden. Zudem gibt es die **Multiplikationsmethode**, welche durch die Funktion $h(k) = \lfloor m(k \cdot A - \lfloor k \cdot A \rfloor \rfloor)$ charakterisiert wird. Die Wahl von m ist zwar in diesem Fall irrelevant, aber A muss eine irrationale Zahl sein.

6.3 Kollisionsbehandlung

Eine Möglichkeit wäre die **Kollisionsbehandlung durch Verkettung der Überläufer**. Diese erfolgt so, dass in jedem Arrayeintrag eine LinkedList gespeichert wird. Zu dieser werden immer vorne Elemente angehängt, wenn etwas den gleichen Hashwert bekommt. Auch die Suche und das Entfernen von Elementen erfolgen ganz normal wie es in einer LinkedList üblich ist.

Zudem gibt es mehrere **Offene Hashverfahren**, welche sich davon charakterisiert, dass alle Datensätze direkt in einem einfachen Array gespeichert werden und pro Platz ein Flag $f_i \in frei, besetzt, wiederfrei$ gesetzt wird. Ist ein Platz belegt, wird **sondiert**, d.h. es werden weitere Plätze in einer bestimmten Reihenfolge betrachtet. Wird ein Schlüssel auf diese Art verschoben, muss dessen Flag auf "wieder frei" gesetzt werden, da dieser sonst nicht gefunden werden würde. Die Suche durchmustert nämlich alle Plätze bis der gesuchte Schlüssel entweder gefunden wird oder ein Platz als frei markiert ist (und dieser folglich nicht enthalten ist).

Eine Art der Sondierung nennt sich **Lineare Sondierung** und hat folgende Sondierungsfunktion: $h(k,i) = (h'(k)+i) \mod m$ für $i = 0, 1, 2, ..., m-1$. Dies führt jedoch dazu, dass die Einfügwahrscheinlichkeit der verschiedenen Plätze unterschiedlich groß ist. Zudem haben lange belegte Teile der Hashtabelle eine stärkere Tendenz zu wachsen als kurze, was umso mehr verstärkt wird, wenn kurze Stücke zusammenwachsen, sobald ihr Lücken gefüllt werden. Dieses Phänomen der **primären Häufung/primary clustering** verschlechtert die Effizienz drastisch, sobald sich α dem Wert 1 annähert.

Das sogennante **Uniform Hashing** ist ein weitere Sondierungsform, die als Idealform des Sondierens gilt, da jeder Schlüssel mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $0, 1, ..., m-1$ als Sondierungsreihenfolge zugeordnet bekommt.

Zudem gibt es noch das **Quadratische Hashing**, welches die Sondierungsfunktion $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ und eine normale Hashfunktion $h'(k)$ verwendet. c_1 und c_2 sind dabei beliebige Konstanten. Dadurch werden zwar primäre Häufungen vermieden, aber es kann zu **sekundären Häufungen** kommen.

Anders als die bisherigen Sondierungsfunktionen verwendet **Double Hashing**, wie der Name schon sagt zwei Hashfunktionen $h_1(k)$ und $h_2(k)$ und die Sondierungsfunktion $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Dadurch ist es effizienter als quadratisches Hashing und fast so gut wie Uniform Hashing.

Zusätzlich gibt es die **Verbesserung von Brendt**, welche zwei weitere Funktionen $j_1 = (j + h2(k)) \bmod m$ und $j_2 = (j + h2(k')) \bmod m$ (mit $k' = T[j].key$) einführt. Ist nun j_1 besetzt aber j_2 frei, verschiebe k' auf j_2 um für k auf j Platz zu machen. Dadurch liegt die Laufzeit selbst beim Extremfall $\alpha = 1$ in $\Theta(1)$. Der Pseudocode sieht dann aus wie folgt:

```
Insert-Brent(T, k):
j ← h1(k)
while T[j].status = used
    k' ←T[j].key
    j1 ←(j + h2(k)) mod m
    j2 ←(j + h2(k')) mod m
    if T[j1].status ≠ used oder T[j2].status = used
        j ← j1
    else
        T[j] ← k
        k ← k'
        j ← j2
T[j] ← k
T[j].status ← used
```

7 Praktische Datenstrukturen

7.1 Datenstrukturen im Java Collection Framework

- **Collection**: Dies ist eine **Interface**, die keine direkte Implementierung erlaubt und Objekte verwaltet. Algorithmen für die Verarbeitung von Container-Klassen werden als statische Methoden hier gesammelt.
  - **Sort**: Sortiert die Elemente einer generischen Liste nach aufsteigender Größe
  - **Binary Search**: Sucht ein Element in der sortierten Liste (Voraussetzung) und liefert einen Index zurück, wenn das Element gefunden wurde (ansonsten eine negative Zahl)
  - **Max**: Liefert das größte Element einer Collection
  - **Shuffle**: Mischt die Elemente einer generischen Liste zufällig
- **Set**: Eine Collection, die keine Duplikate erlaubt
  - **TreeSet**: Die Implementierung erfolgt als Rot-Schwarz-Baum. Die Laufzeit von Einfügen, Suchen und Löschen eines Elements liegt bei einem Baum mit n Elementen in $O(log(n))$. Außerdem müssen alle Elemente davon das Interface Iterable implementieren. null-Elemente sind nicht erlaubt.
    - **Rot-Schwarz-Baum**: Dies ist ein Spezialfall eines B-Baums der Ordnung 4, bei dem das Einfügen und Löschen effizienter ist, als in B-Bäumen solange die Datenmenge nicht zu groß ist. Ist der Baum selbstbalanciert, bedeut das, dass jeder Knoten entweder rot oder schwarz ist. Die Wurzel und die Blattknoten sind jedoch immer schwarz.

Zusätzlich beinhaltet jeder Pfad, der vom Knoten weggeht immer gleich viele schwarze Knoten.

  - **HashSet**: Einfügen, Suchen und Löschen sind zwar in konstanter Zeit möglich, aber Rehashing kann zu Problemen führen. Das Hashing erfolgt mittels Verkettung der Überläufer. Falls die Liste zu lange wird, wird sie in eine TreeMap umgewandelt. Bei Überschreitung des Belegungsfaktors ($\alpha > 0.75$) verdoppelt sich die Größe. null-Elemente sind erlaubt.

- **List**: Ein geordnetes Set
  - **ArrayList**: Indexzugriff von Elementen erfolgt in $O(1)$. Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer. Insgesamt somit in $O(n)$.
  - **LinkedList**: Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer, somit erneut in $O(n)$. Einfügen und Löschen ohne Indexzugriff erfolgt in konstanter Zeit.

- **Queue/Dequeue**: Verwalten von Warteschlangen, Einfügen an beiden Enden möglich, wenn Dequeue
  - **LinkedList**: s.o.
  - **Priority Queue**: Dies wird als Min-Heap implementiert, somit erfolgen Einfügen eines Elements und Löschen des ersten Elements in $O(log(n))$. Das Löschen eines beliebigen Elements braucht hingegen eine Laufzeit von $O(n)$. Das Lesen des ersten Elements erfolgt in $O(1)$.

- **Maps**: Verwaltung von Schlüsseln mit zugehörigen Werten. Außerdem eine Verallgemeinerung von Arrays mit einem beliebigen Indextyp (nicht nur int)
  - **HashMap**: primitive Implementierung von HashSet
  - **TreeMap**: primitive Implementierung von TreeSet

	Interface				
Konzept	Set	List	Queue	Deque	Map
Arrays	–	ArrayList	–	ArrayDeque	–
Bäume	TreeSet	–	–	–	TreeMap
Hashtabellen	HashSet	–	–	–	HashMap
Heap	–	–	PriorityQueue	–	–
Verkettete Listen	–	LinkedList	LinkedList	LinkedList	–

7.2 TimSort

Diese Mischung aus Mergesort und Insertionsort wird dafür verwendet, Collections zu sortieren, indem er schon sortierte Stücke (Runs) des Inputs S findet und die dann paarweise verschmilzt (Merging). Was die **Berechnung der Runs** angeht, wird eine Mindestlänge von Min_Run vorausgesetzt, welche so gewählt wird, dass $\frac{Groeßedes Arrays}{Min_Run} \leq$ eine Zweierpotenz ist.

Nun werden startend bei dem ersten Element, das noch nicht in einem Run ist, so lange **Elemente zum Run hinzugefügt**, bis die Sequenz entweder aufsteigend oder strikt absteigend ist. Ergibt sich nun eine strikt absteigende Sequenz, wird sie invertiert. Ist dies $\geq$ Min_Run, wird dieser Run verwendet, ansonsten werden so lange Elemente hinzugefügt bis Min_Run erreicht wird.

Die **schlussendliche Verschmelzung** erfolgt so, dass $A[0]$ und $B[0]$ verglichen werden und der kleinere Wert ins Merge-Array verschoben wird. Gewinnt das gleiche Array mehr als Min_Galloping Mal, wechselt der Algorithmus in den Galloping Modus. Dies funktioniert so: Hat A Min_Galloping mal gewonnen, sucht es das erste Element A[i] in A aus, das größer als B[0] ist. Dann kopiert es $A[0]$ bis $A[i-1]$ in das Merge-Array und sucht nach $A[i]$ mittels galoppierender binärer Suche, (d.h. es vergleicht $B[0]$ mit $A[1]$, $A[3]$, $A[7]$, ... , $A[2k-1]$ bis $B[0]$ kleiner ist, dann binäre Suche)

Stack $\mathcal{R}$ von Runs mit Aufbau $\mathcal{R}[1], \ldots, \mathcal{R}[\text{height}(\mathcal{R})]$, wobei $\mathcal{R}[1]$ top-of-stack ist.
Die Länge von Run $\mathcal{R}[i]$ wird mit r_i bezeichnet.

```
merge_collapse(R)
while height(R) > 1
    if height(R) > 2 und r3 ≤ r2 + r1
        if  r3 < r1
            verschmelze R[2] und R[3] am Stack
        else
            verschmelze R[1] und R[2] am Stack
    elseif height(R) > 1 und r2 ≤ r1
        verschmelze R[1] und R[2] am Stack
    else break
```

Nach Ausführung von merge_collapse($\mathcal{R}$) gelten folgende Invarianten:

(1) $r_3 > r_2 + r_1$

(2) $r_2 > r_1$

Der Vorteil an TimSort ist, dass dieser sehr schnell auf Arrays ist, die viel Struktur haben. Average- und Worst-Case $O(n \cdot log(n))$ wie Mergesort. Der Nachteil ist jedoch, dass es ziemlich viel zusätzlichen Speicher benötigt. Der Pseudocode sieht dann so aus:

```
TimSort(S)
runs ← partitioniere S in Runs
R ← leerer Stack
while runs ≠ ∅
    entferne Run r von runs and pushe r auf den Stack R
    merge_collapse(R) // ausbalanciertes Verschmelzen
while height(R) ≠ 1
    verschmelze die beiden oberen Runs am Stack
```