Algorithmen und Datenstruktur

Martin Schmidli

2. Februar 2015

Inhaltsverzeichnis

Kapitel 1

Algorithm

Je nach Algorithmus dauert es länger eine Aufgabe zu lösen.

1.1 Sort Algorithmen

Ziel ist es eine Anzahl von n Ganzzahlenwerten ihrem Wert nach zu Ordnen (0,1,2...).

1.1.1 Bubblesort

Diese Tatsache wird im Bubblesort Algorithmus ausgenutzt.
Um eine Liste mit n Zahlen zu sortieren, muss im Worst Case Szenario i = n-1 mal durch die Liste durchiteriert werden. In jedem Durchgang werden die Positionen der Zahlen überprüft. Ist die das erste Element grösser als das zweite Element wird deren Position getauscht. Ist die erste Zahle kleiner behält sie ihren Platz.

Der Algorithmus wird Bubblesort genannt, weil die höchstwertigste Zahl sehr rasch an das Ende des Arrays verschoben wird. Die Zahl schwebt nach oben resp. an das Ende des Arrays wie in einer Blase/"Bubble". Nun müssen wir nicht jedes m mal durchiterieren, denn wir wissen, dass die Zahlen am Ende des Arrays bereits die richtige Position haben.

Bsp. Wir iterieren 2 mal durch das Array. Die 2 Letzten Zahlen im Array haben also die korrekte Position.
Aus diesem Grund überprüfen wir pro Durchgang immer eine Zahl weniger, i–.

Java Implementation:

```
/**
 * Non optimized bubble sort for an int array
* @param a, array of n integers which we would like to sort
*/
public static void bubbleSort(int[] a) {
        cnt=0;
        int m = a.length-1;
        for(int i=m; i>0; i--){
                for (int k=0; k < i; k++){
                        if(a[k]>a[k+1]) swap(a,k,k+1);
```

```
            }
            // now a[i] is on its final position!
        }
}
```

Eine mögliche Verbesserung wäre es, ein "isSorted" Flag einzubauen. Wir überprüfen pro Durchgang i, ob wir geswapt haben. Wurden keine Zahlen vertauscht, ist das Array korrekt sortiert und wir können die Schleife abbrechen.

1.1.2 Quicksort

Der Quicksort Algoritmus ist meist sehr viel schneller als der Bubble Sort Algoithmus. Nur wenn die Liste bereits sortiert und dieser Spezialfall nicht abgefagen wird, sind Sie gleich schnell. Das Grundproblem/ Aufgabe wird beim Quicksort in kleine Probleme zerlegt, wir sprechen von Partitionieren. In der Programmierung sprechen wir von Rekursion.

Wir suchen uns ein zufälliges Element aus dem Array aus. Dieses Element wird Pivot genannt.
Wir tauschen das letzte Element des Arrays mit dem Pivot. Damit fangen wir den Spezialfall einer bereits sortierten Liste ab.

Stellen wir uns zwei Zeiger "Left" und "Right" auf das Array vor. Left startet beim 1. Elements des Arrays, A[0]. Right startet beim Ende des Arrays [Anzahl Elemente-1]. Left fährt von Links nach Rechts. Der Index von Left wächst bis er auf ein Element zeigt, welches grösser ist als der Pivot. Right fährt von Rechts nach Links. Der Index von Right wird kleiner, bis es auf ein Element kleiner als der Pivot zeigt. Die beiden Elemente bei A[Left] und A[Right] werden ausgetauscht.

Dieser Ablauf wird fortgesetzt, bis die Zeiger aufeinander zeigen, Left == Right.
Alle kleineren Zahlen als der Pivot sind nun also auf der Linken Seite, alle grösseren auf der Rechten Seite des momentanen Indexes. Nun können wir den Pivot mit dem Element an dieser Stelle vertauschen.

Wir haben die korrekte Position des momentanen Pivot gefunden. Es ergebn sich nun zwei neue Bereiche. Links vom Pivot und Rechts vom Pivot.
Wir beginnen den Prozess von vorne.
Wir rufen die Funktion zweimal mit den neuen Bereichen (Begin,Left-1) und (Left+1, End) auf.
Der Prozess begint von vorne.

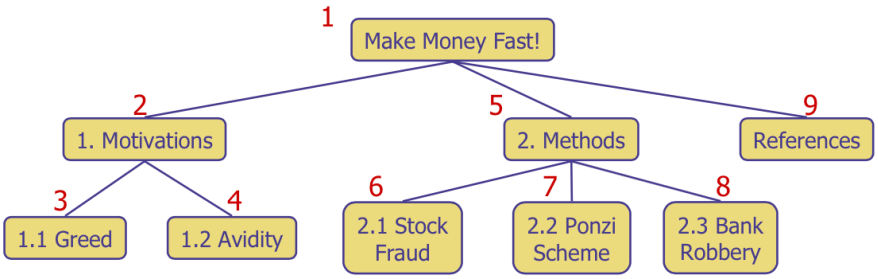
Java Implementation:

```
public static void quickSort(int[] a, int start, int end)
{
	int left = start;
	int right = end;

	// check if we have to do something
	if(end-start > 0)
	{
		// move random to last index
		Random rand = new Random();
		int random = rand.nextInt((end-start)+1)+start;
		swap(a,random,end);
		int pivot = a[end];

		// we do move until the pointer are at the same position
		while(left < right)
		{
			// move left pointer until we found a bigger number as pivot
			while(left < right && a[left] <= pivot)
			{
				left++;
```

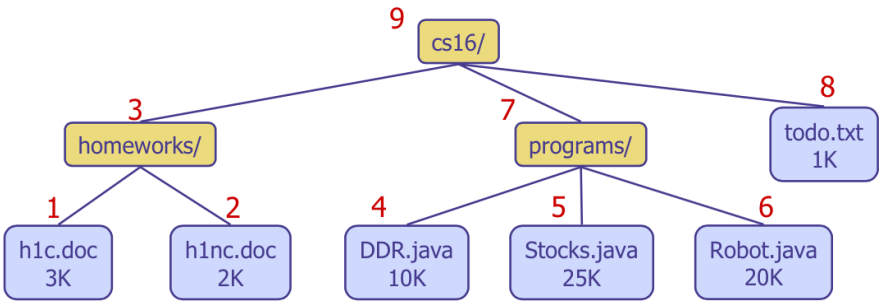
```
                }

                // move right pointer until we found a smaller number as pivot
                while(left < right && a[right] >= pivot)
                {
                        right--;
                }

                // exit if pointer are at the same position
                if(left == right) break;

                // we found two numbers, lets swap them
                swap(a,left, right);
            }
            //Move the pivot to his correct position
            swap(a, end,left);

            // Now we do have two new fields/ arrays. Lets call quickSort recursively
            quickSort(a, start, left-1);
            quickSort(a,left+1, end);
        }
}
```

1.2 Trees

Preorder Traversal



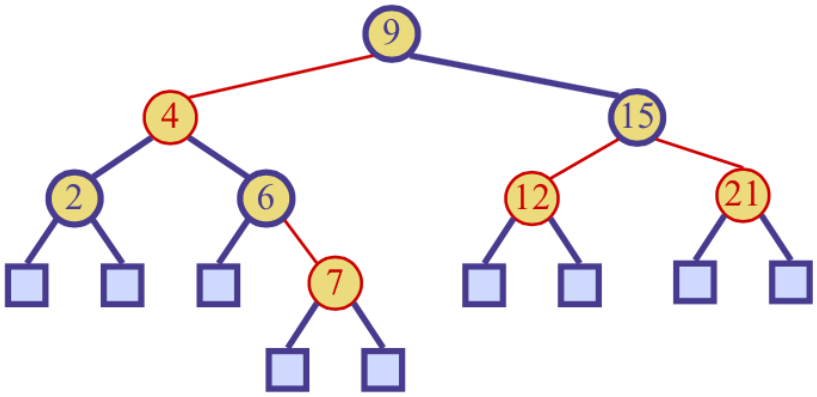
Postorder Traversal



1.2.1 Red Black Tree

Ein Red Black Tree hat 5 Eigenschaften

1. Jeder Node ist entweder schwarz oder rot
2. Der root Node ist immer schwarz
3. Jedes child und parent von einem roten Node ist schwarz.
   Ein roter Node kann kein rotes Child und keinen roten Parent haben.
4. Jeder Pfad von Root zu NULL node hat die gleiche Anzahl an schwarzen Nodes.
5. Jedes Leaf ist schwarz



Die meisten BST Operationen (einfügen, suchen, max ...) benötigen $O(h)$ Zeit. Wobei h die Höhe des Trees ist. Die Höhe des Red Black Tree ist immer Log n. Damit ist die Laufzeit dieser Operationen in einem Red Black Tree $O(Logn)$.

Der Ausgleich des Höhenunterschiedes, wie er beim AVL Tree vorkommt, muss hie rnicht angewendet werden. Ein Red Black Tree ist immer balanciert.

Der Suchalgorithmus in einem Red Bacl Tree ist derselbe wie in einem Binären Suchbaum.

Einfügen

Die Farbe eines neu eingefügten Elementes ist immer rot.
Damit ein solcher Baum nach dem einfügen noch immer in Balance ist, haben wir zwei Mittel.

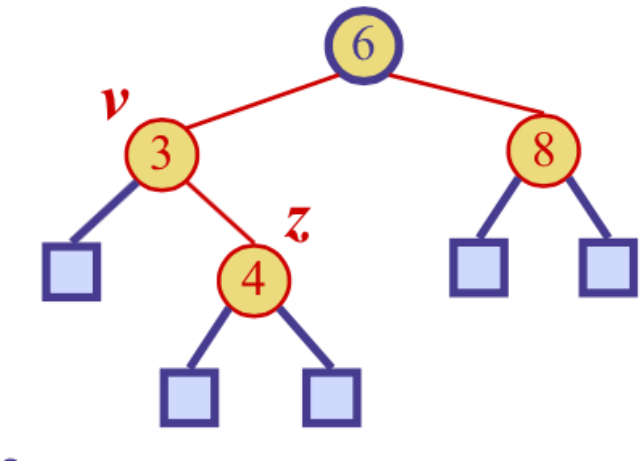
1. Recoloring
2. Rotation

Zuerst sollten wir immer Recoloring anzuwenden. Erst dann rotieren wir.

z ist das neu eingefügte Element.
v ist der parent.
w ist der uncle/Sibling des parent.

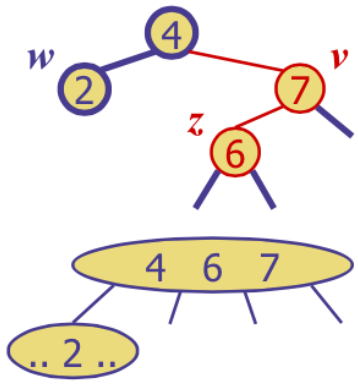
Wir unterscheiden die folgenden Szenarien

1. Der neu eingefügte Node z wird root. z wird schwarz.
2. Der Parent von z ist schwarz. Wir können den neuen Node einfach anhängen.
3. Der Parent von z ist rot. Wir haben ein Doppel Rot Problem generiert.

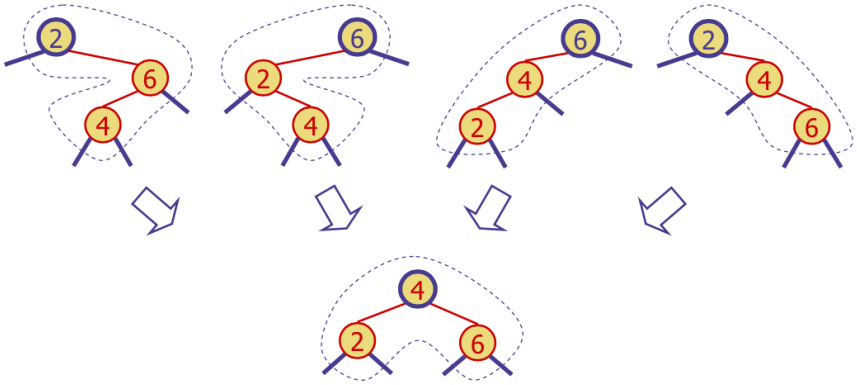


doppel Rot Problematik

(a) Der Uncle w vom neu eingefügten node z ist Schwarz.



Wir müssen restrukturieren/restructuring durchführen.

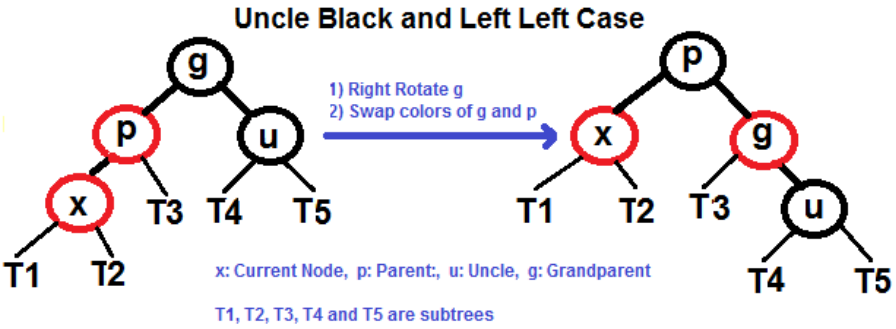


Restructering im Detail

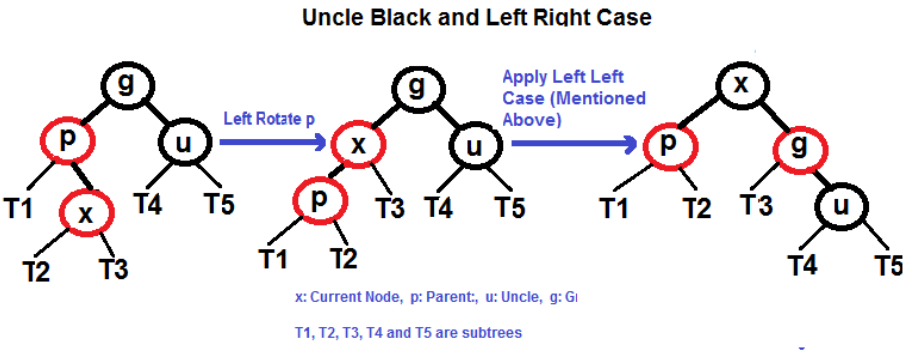
Nachfolgend die einzelnen schritte des restrcutering im Detail.
Wir gehen gleich vor wie beim ALV Tree.

x ist der neu eingefügte Node
p ist der parent.
g ist der grandparent.
u ist der uncle

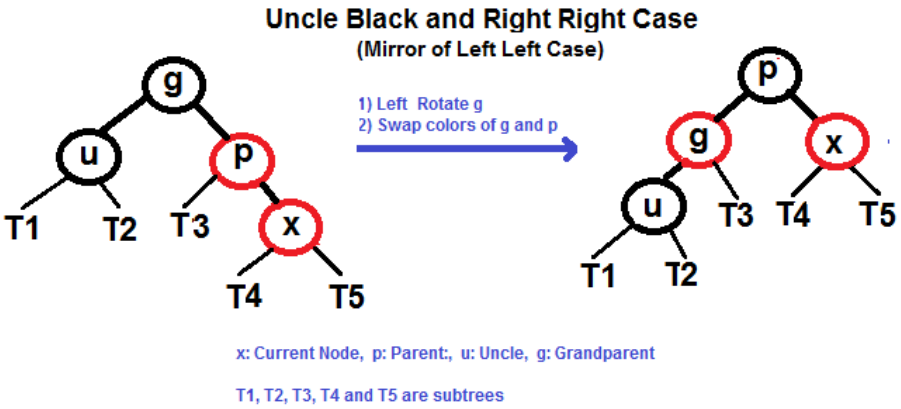
- Left Left Case
  p ist das linke Kind von g und x ist das linke Kind von p
  i. Right Rotation von g
  ii. Farbe tauschen von g und p



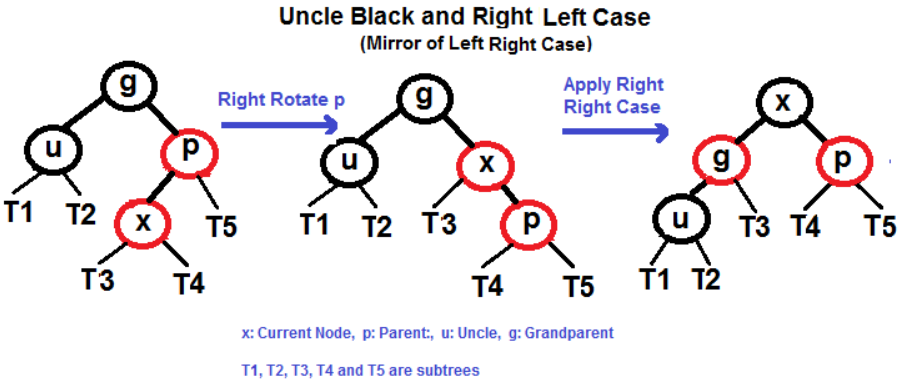
- Left Right Case
  p ist das linke Kind von g und x ist das rechte Kind von p



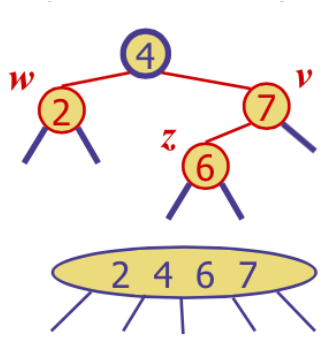
- Right Right Case
  i. Left Rotation von g
  ii. Farbe tauschen von g und p



- Right Left Case

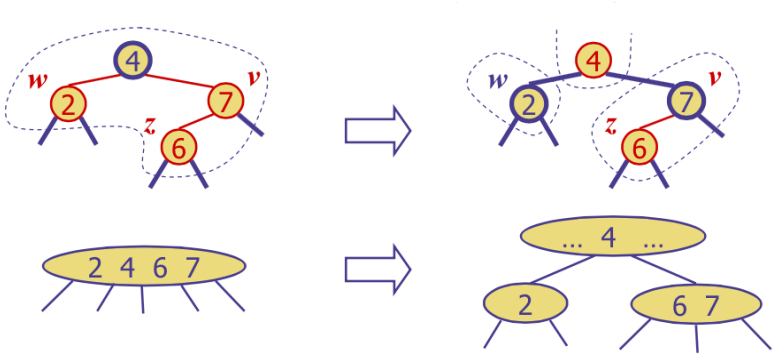


(b) Der Uncle w von z ist rot



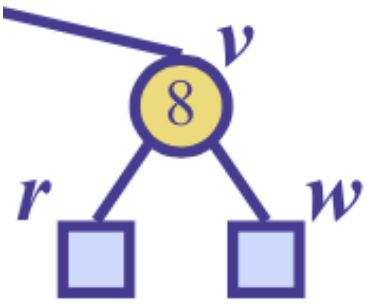
Wir müssen recolering durchführen.

i. Parent v und Uncle w werden Schwarz
ii. Grandparent wird Rot, ausser er ist der root knoten



Löschen

v ist das zu löschende interne Element.
w ist das zu löschende externe Element. Ist der Sibling von r. child von v.
r ist das child von v.

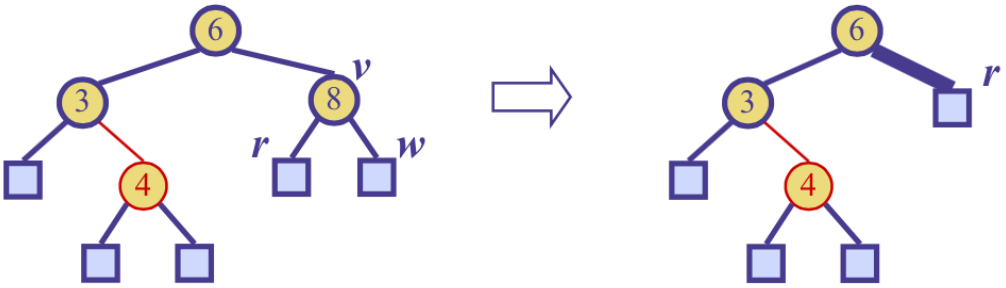


Wir unterscheiden diese Szenarien

1. Wenn wir den root Knoten löschen wollen.
   Wir ersetzen den Root Knoten mit dem ersten inorder Element des rechten Subtrees (analog 2,4 Tree).

2. wenn r oder v rot sind, färben wir r schwarz und löschen den node v.

3. r und v sind beide schwarz. wir färben r double black/ doppel schwarz. Damit ist sind die Bedingungen für einen RotSchwarz Baum nicht mehr erfüllt. Wir müssen den Baum reorganisieren.



doppel Schwarz Problematik

Das doppel Schwarz Problem lässt sich wie folgt lösen

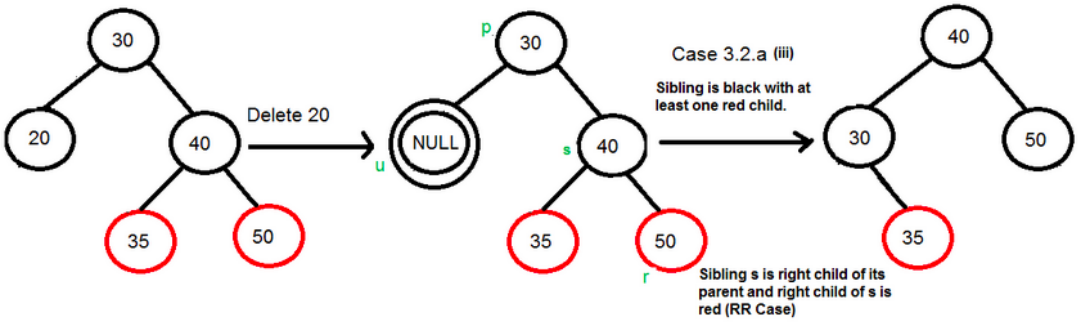
(a) y ist schwarz und hat ein rotes Kind

Wir führen ein restructering durch

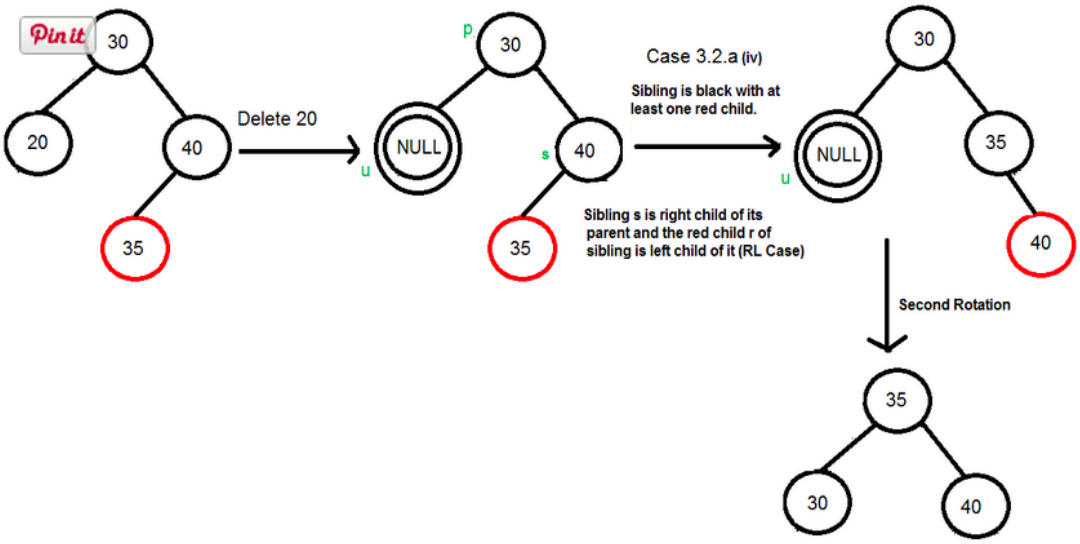
Es gibt 4 Szenarien

Restructuring im Detail

- Left Left
  Spiegelung von Right Right
- Left Right
  Spiegelung von Right Left
- Right Right

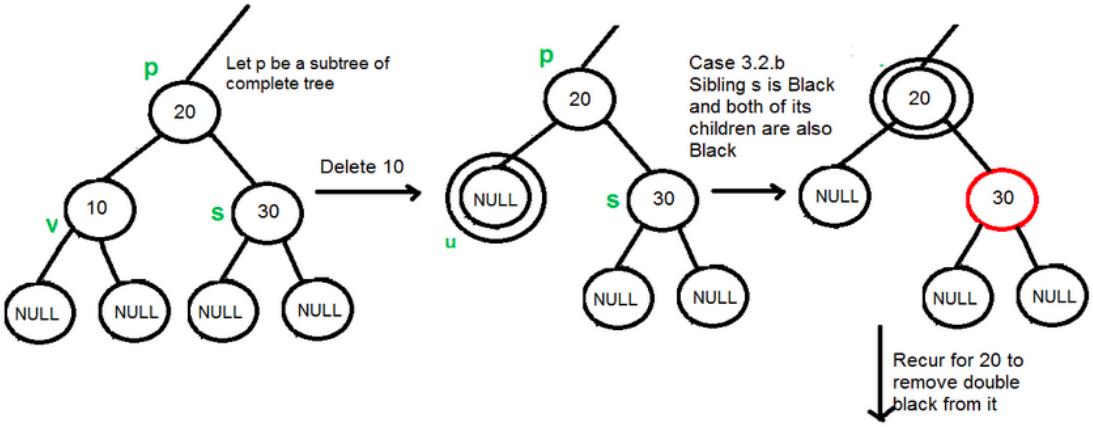


- Right Left

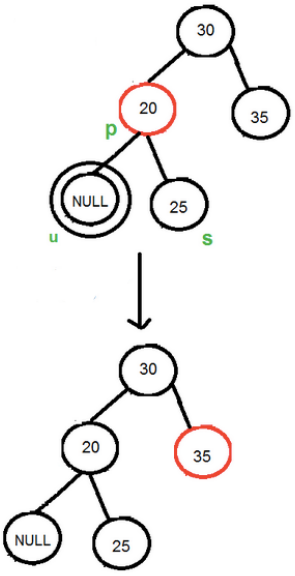

30
20
40
35
Delete 20
u
NULL
p
30
s
40
35
Case 3.2.a (iv)
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
30
u
NULL
35
40
Second Rotation
35
30
40

(b) y ist schwarz und beide Kinder sind schwarz

Wir führen ein recoloring durch



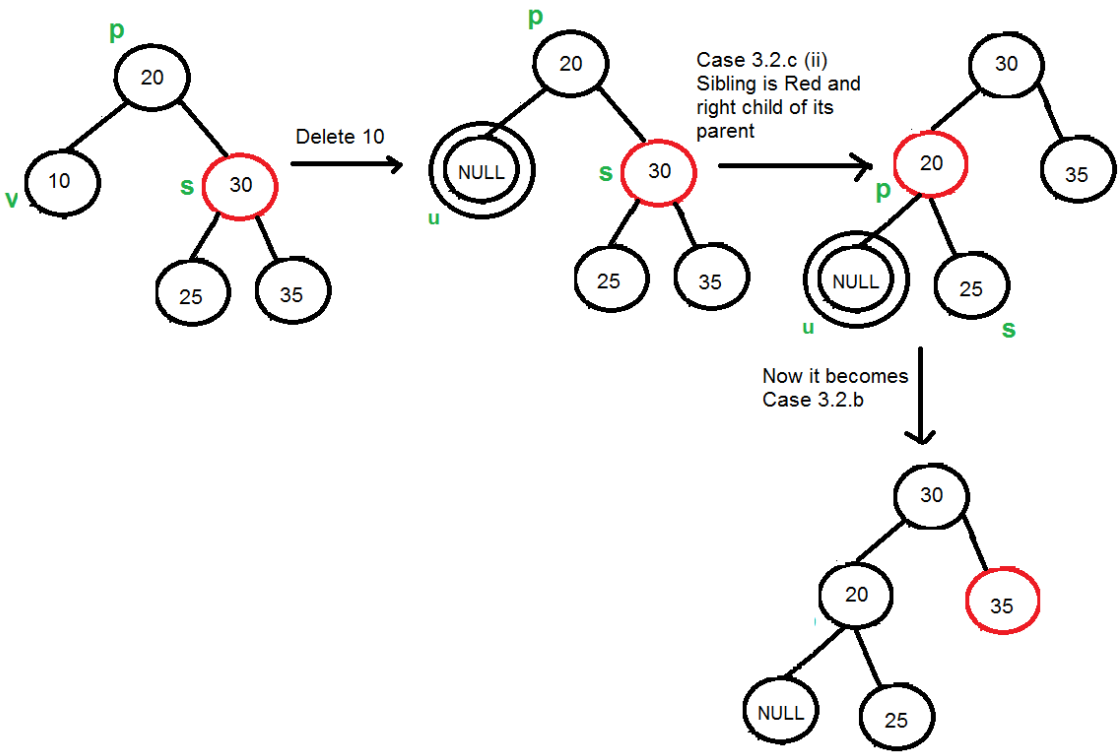
In diesem Beispiel ist der parent node schwarz.
Wenn der parent node rot ist, wird aus dem roten parent ein single black node. (red + double black = single black). Der sibling 35 wird rot.



(c) y ist rot

Wir führen Rotationen durch um den alten sibling nach oben zu bringen

- Left Case
  Spiegelung von Right Case
  Right rotation parent p
- Right Case
  Left rotation von parent p



(d) Wenn r root wird, mach es single black (Die Black height des baumes wird um 1 reduziert).

1.2.2 Splay Tree

Ein Splay Tree ist nicht balanciert.
Dies Grundfunktion eines solchen Baumes ist die Suche.
Die Suchfunktion wird solange ausgeführt, bis der gesuchte Node an der Root Position steht.

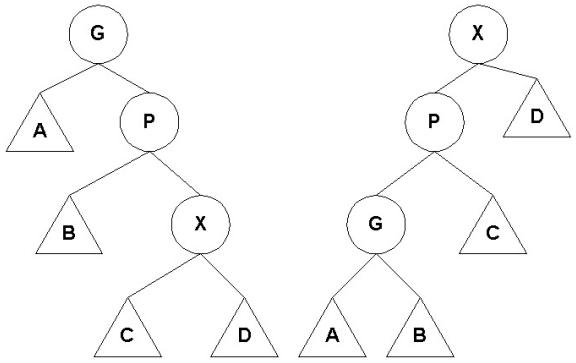
Für die Beispiele wird die nachfolgende Notation verwendet.

- X ist der Node/ Wert den wir suchen
- G ist der Parent Node von X
- P ist der Grossvater Node von X

Wir unterscheiden die folgenden Sitiuationen

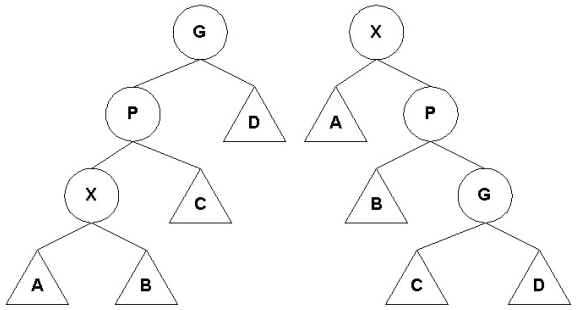
Zig-Zig

- X ist das rechte Kind von P und P ist das rechte Kind von G



ZIG_ZIG_RIGHT

- X ist das Linke Kind von P und P ist das linke Kind von G



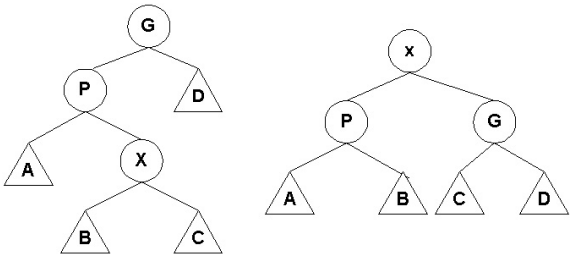
ZIG_ZIG_LEFT

Lösung:

1. G mit X ersetzen
2. P wird Kind von X
3. G wird Kind von P

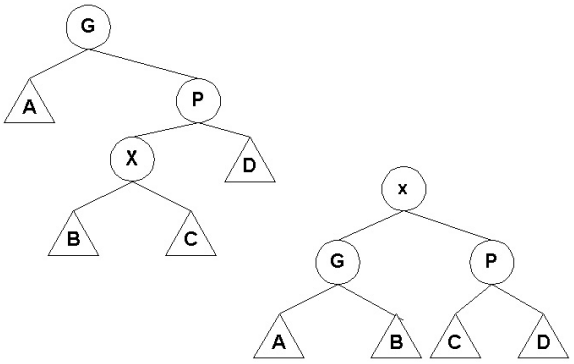
Zig-Zag

- Wenn P ein linkes und X ein rechtes Kind ist.



ZIG_RIGHT_ZAG_LEFT

- Wenn X ein linkes und P ein rechtes Kind ist.



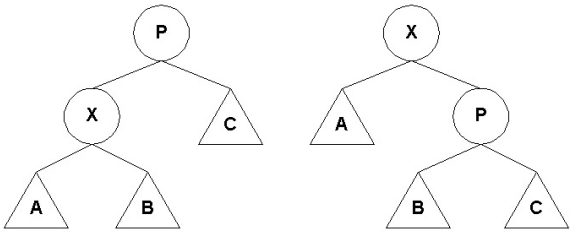
ZIG_LEFT_ZAG_RIGHT

Lösung:

1. G mit X ersetzen
2. G und Z werden Kinder von X.
   Z sollte dabei immer auf der Rechten Seite stehen, da es grösser als X ist.

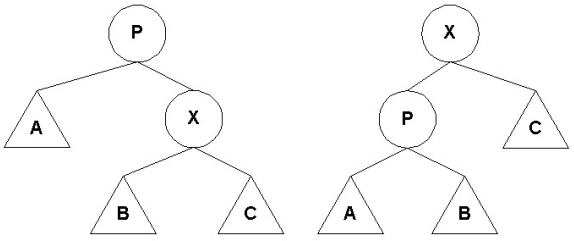
Zig

- X hat keinen Grossvater. X ist ein linkes Kind.



ZIG_LEFT

- X hat keinen Grossvater. X ist ein rechtes Kind.



ZIG_RIGHT

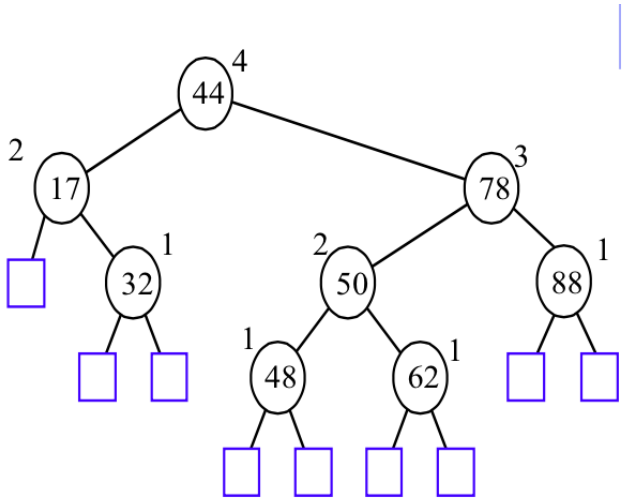
Lösung:

1. X wird mit P getauscht.
2. P wird ein Kind von X.
   Eines der früheren Kinder von X bleibt beim X.

1.2.3 AVL Tree

Die Subbäume des AVL Trees dürfen maximal einen Höhenunterschied von 1 haben.
AVL Bäume sind immer balanciert.

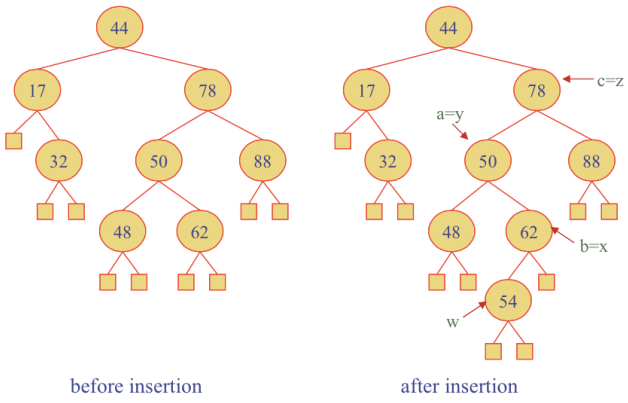
Ist ein Binary Search Tree.



Die Höhe eines AVL Tree ist $O(logn)$, wobei n die Anzahl gespeicherten Elemente ist.

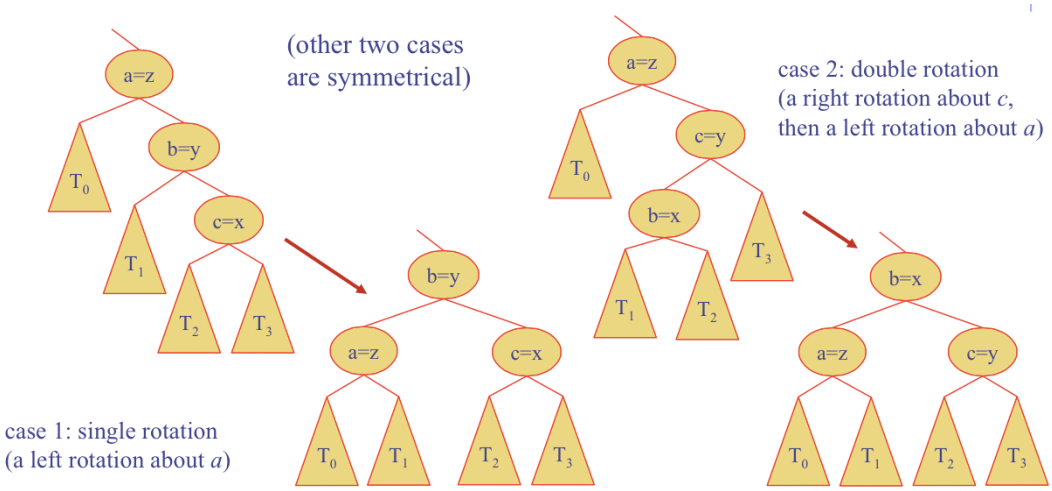
Einfügen

Wir wollen 54 einfügen.
Wir hängen das neue Element immer an einen externen Node.



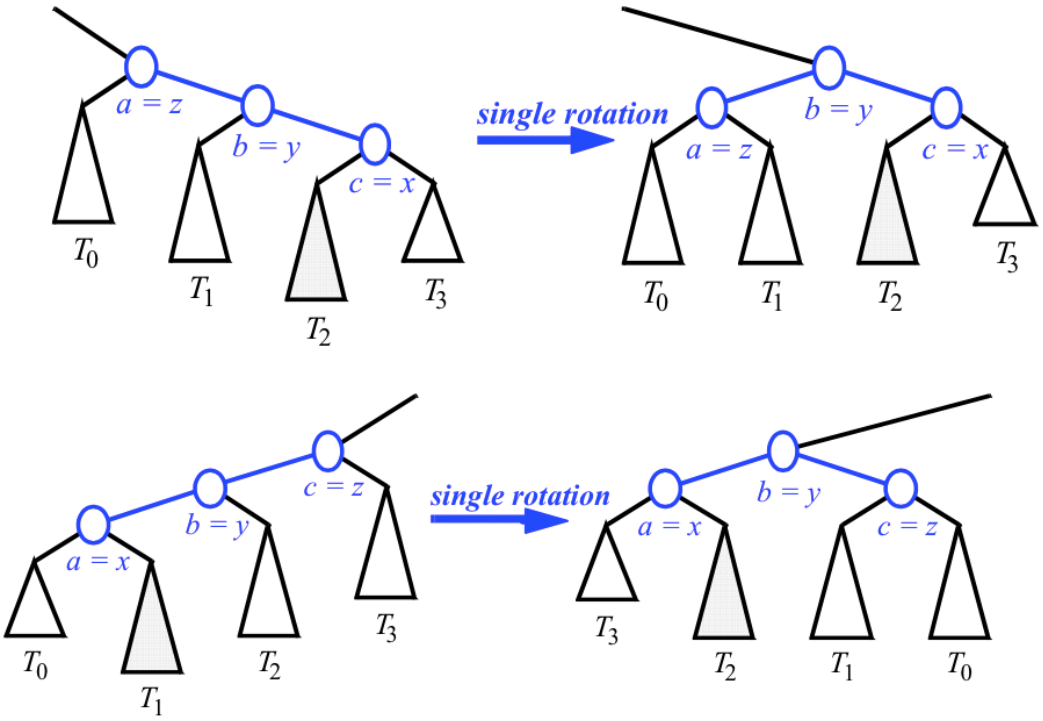
Nun ist aber für den Node 78 der Höhenunterschied seiner Äste auf 2 angestiegen.
Der Baum ist nicht mehr balanciert.
Wir müssen den Baum restrukturieren.

Restructering



Treerotation

Single Rotation



Left Rotation
Es Passieren 3 Operationen

1. Wir merken uns den rechten Sub Tree
2. Der linke Subtree des neuen Roots wird dem alten Root rechts angehängt
3. Der alte Root wird dem neuen Root links angehängt

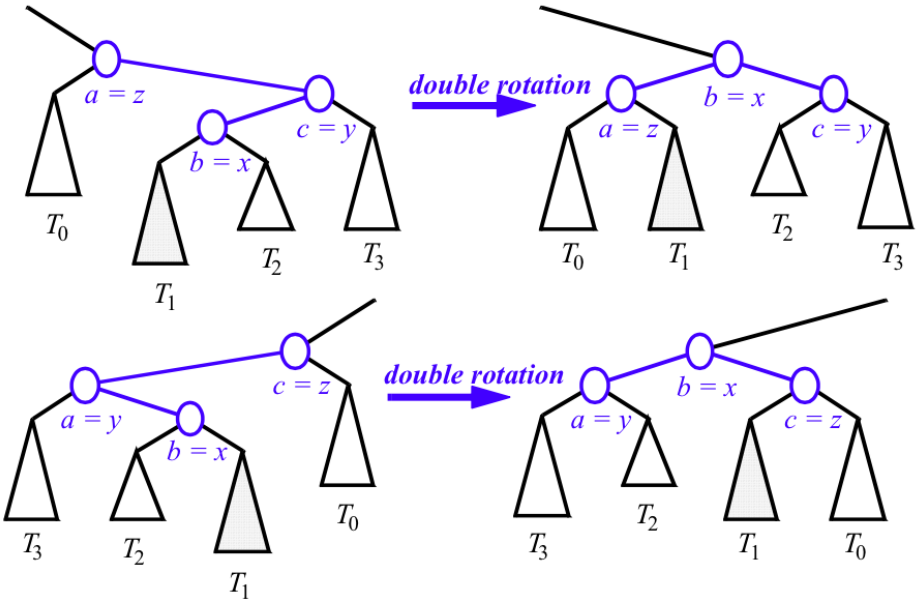
```
function LeftRotation(node)
{
        /* Predfinition: node.Left ! == null
        Post: node.Left is the new root of the subtree, node has become node.Left s right
            child and, BST properties are preserved */

        node newRoot = node.Right;
        node.right = newRoot.Left;
        newRoot.Left = node;
}
```

Right Rotation
Es Passieren 3 Operationen

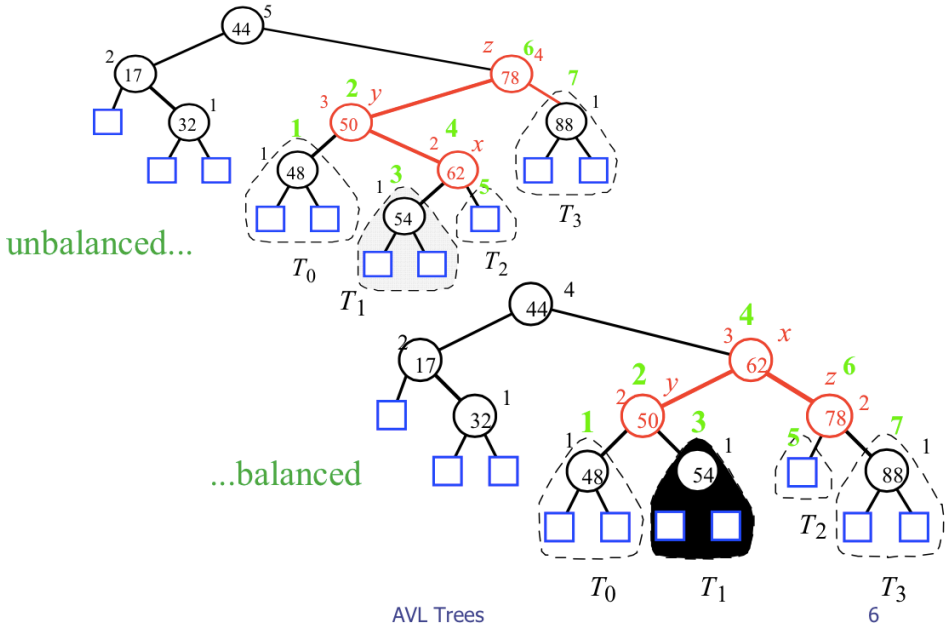
1. Wir merken uns den linke Sub Tree
2. Der rechte Subtree des neuen Roots wird dem alten Root links angehängt
3. Der alte Root wird dem neuen Root rechts angehängt

```
function RightRotation(node)
{
        /* Predfinition: node.Left ! == null
        Post: node.Left is the new root of the subtree, node has become node.Left s right
            child and, BST properties are preserved */

        node newRoot = node.Left;
        node.left = newRoot.Right;
        newRoot.Right = node;
}
```

Double Rotation

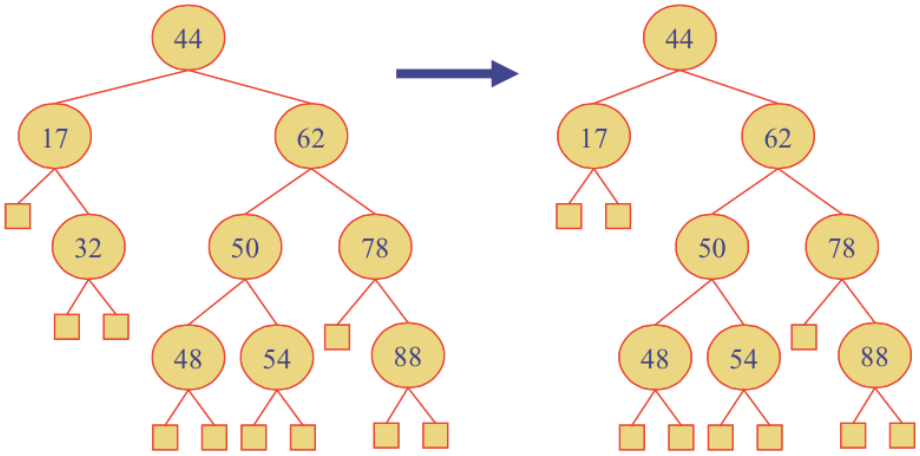


Rotation Example



Löschen

Der gelöschte Node wird ein leerer externer Node des parent.
Gleiches vorgehen wie beim Binary Seach Tree.



before deletion of 32 after deletion

1.2.4 Binary Search Tree

Jeder Binäre Suchbaum besitz ein Node Element/ Root mit dem Wert x. Der Linke Subtree von Root beinhaltet alle Elemente die kleiner sind als x, $<$ x. Der Rechte Subtree beinhaltet alle Elemente die grösser gleich als x sind, $\geq$ x.

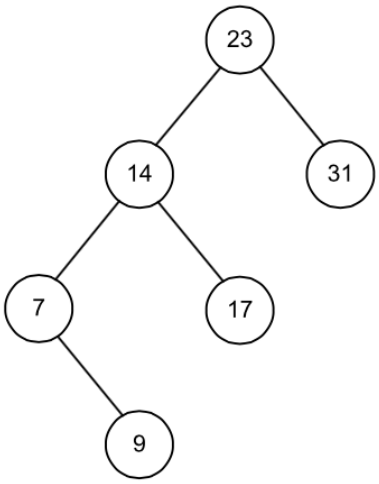
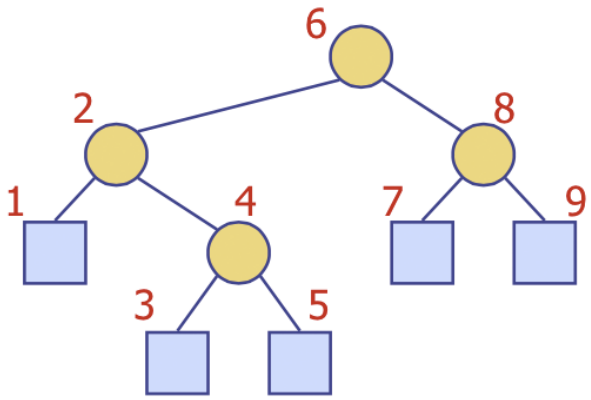


Abbildung 1.1: Einfacher unabalancierter binäre Suchbaum

Inorder Traversal



Einfügen

Der Aufruf funktioniert rekursiv.

Code:

```
// Check if root is available
function Insert(value)
{
	if (root == null) // No root exists
	{
		root = new node(value) ;
	}
	else
	{
		InsertNode(root, value);
	}
}

function InsertNode(current, value)
	// check size
	if(value < current.Value)
	{
	// Smaller, value will be added left
		if(current.Left == null)
		{
			current.Left = new node(value);
		}
		else
		{
			InsertNode(current.Left, value);
		}
	}
	else
	// Bigger as root, value will be added right
	{
		if(current.Right == null)
		{
			current.Right = new node(value);
		}
		else
		{
			InsertNode(current.Right, value)
		}
	}
```

Suchen

Es gibt 4 Szenarien, die wir unterschieden müssen.

1. the root == null in which case value is not in the BST
2. root.Value == value in which case value is in the BST
3. value < root.Value, we must inspect the left subtree of root for value
4. value > root.Value, we must inspect the right subtree of root for value

```
functions contains(root, value)
{
        if(root == null)
        {
                return false
        }
        if(root.Value == value)
        {
                return true;
        }
        else if(value < root.Value)
        {
                contains(root.left, value);
        }
        else
        {
                contains(root.right, value);
        }
}
```

Löschen

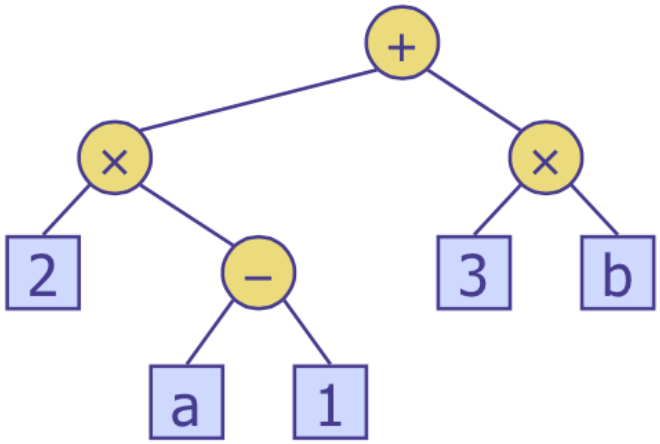
Es gibt 4 Szenarien, die wir unterschieden müssen.

1. the value to remove is a leaf node
   just remove the node
2. the value to remove has a right subtree, but no left subtree
   We link the parent with the right subtree
3. the value to remove has a left subtree, but no right subtree
   We link the parent with the left subtree
4. the value to remove has both a left and right subtree
   we promote the largest value in the left subtree

Arithmetic Expression Tree

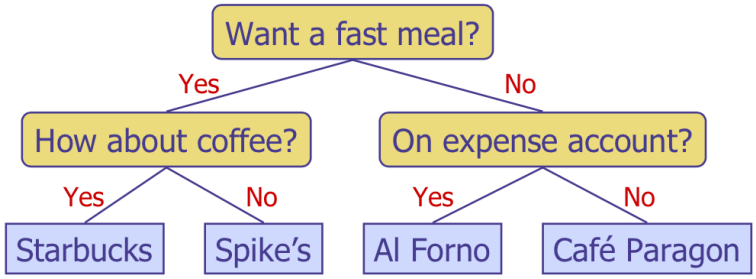
Interne Nodes sind Operatorem
Externe Nodes sind Zahlen/ Variabeln

$$(2 * (a - 1) + (3 * b))$$



Decision Tree

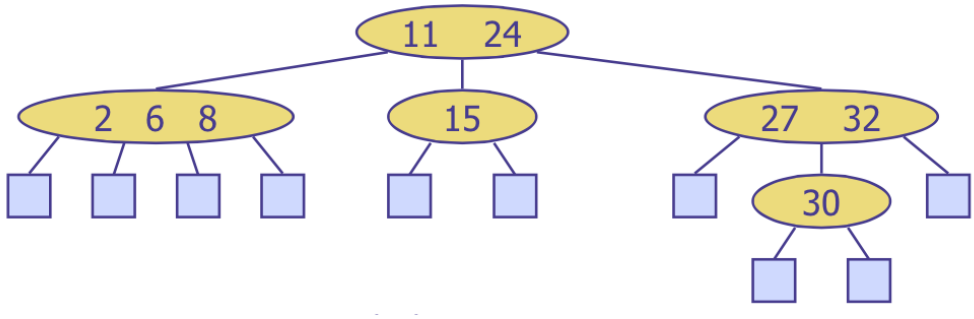
Enstcheidung wo man essen gehen will



1.2.5 Multi-Way Search Tree

Eigenschaften:

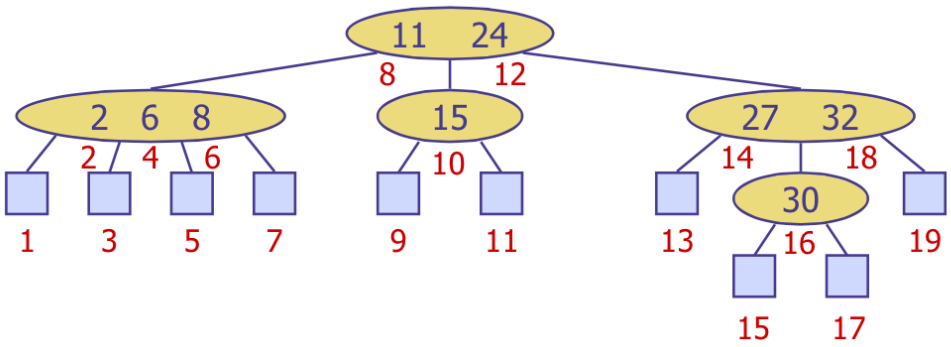
- Die Leafs dienen nur als Platzhalter.
- Jeder Interne Knoten besitzt mindestens 2 Kinder und speichert d-1 Key Items. Wobei d die Anzahl der Kinder ist.



Searching

Inorder Traversal

Eine Inorder Traversierung besucht die Keys in aufsteigender Reihenfolge.



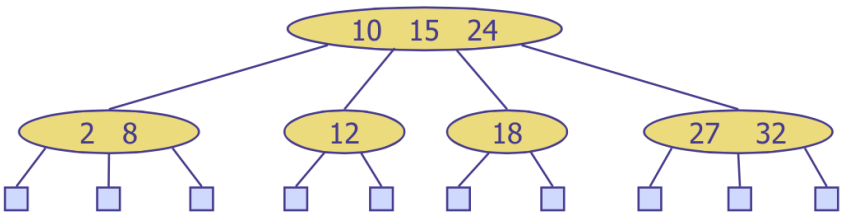
1.2.6 2,4 Tree

Auch 2-4 tree oder 2,3,4 Tree genannt.
Ist ein Multiway Search Tree.

Eigenschaften:

- Jeder interne Knote hat maxima 4 Kinder
- Jeder Externe Knoten hat die selbe Tiefe

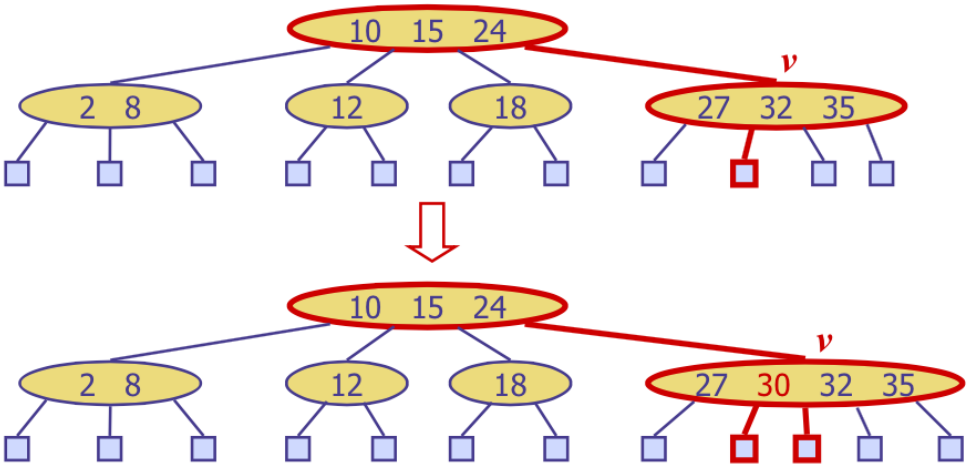
Ein 2,4 Tree mit n elementen hat die Höhe $O(logn)$
Die Suche in einem 2,4 Tree dauert $O(logn)$



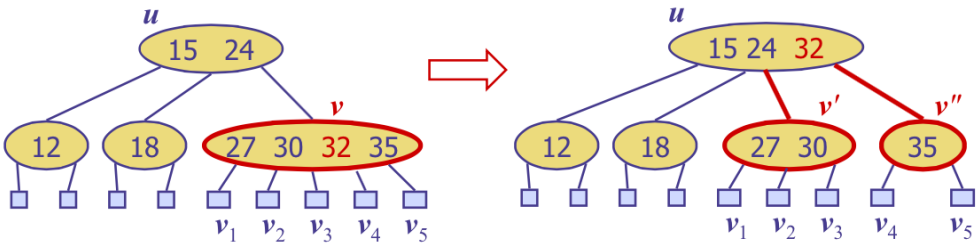
Um sich das ganze besser vorstellen zu können, hilft es sich den 2,4 Baum als Red Black Tree aufzuzeichnen.

Einfügen

Wir wollen den Key 30 einfügen.
Wir fügen 30 zum Knoten v hinzu.
Wir generieren einen Overflow, da ein interner Knoten max 4 Kinder haben darf.

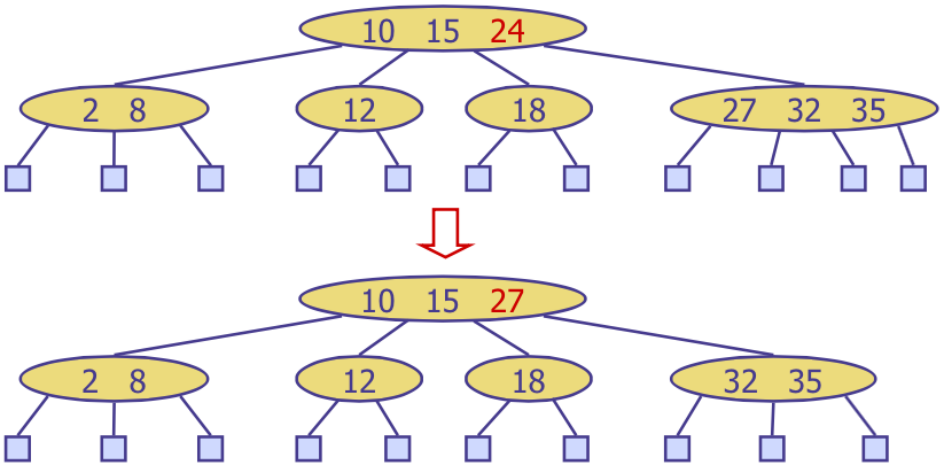


Wir führen einen Split durch. Node v wird augeteilt.
Key 32 wird zum Parent hinzugefügt.
v': 27 und 30 bilden einen 3-Node.
v": 35 bildet einen 2-Node



Löschen

Wir wollen Key 24. löschen
Wir ersetzen 24 mit dem 1. inorder Element (inorder Successor) des rechte Knotens.
In diesem Fall 27.



Underflow
Wenn wir Elemente aus einem 2,4 Tree löschen, kann es zu einem Underflow kommen. Node v wird in diesemfalle ein 1-Node mit nur einem Kind und keinem Key.

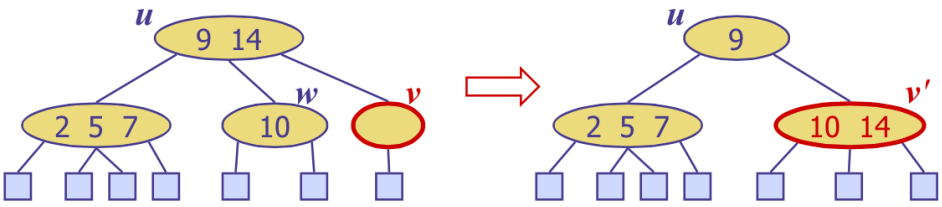
w: sibling
u: parent
v: underflown node

Es gibt 2 Möglichkeiten einen Underflow zu beheben

1. Fusion:

   Wir fusionieren w mit v. Es entsteht der Knoten v'.
   Wir transferien einen Key von u nach v'.

   Achtung. Eine Fusion kann zu einem Underflow im parent u führen

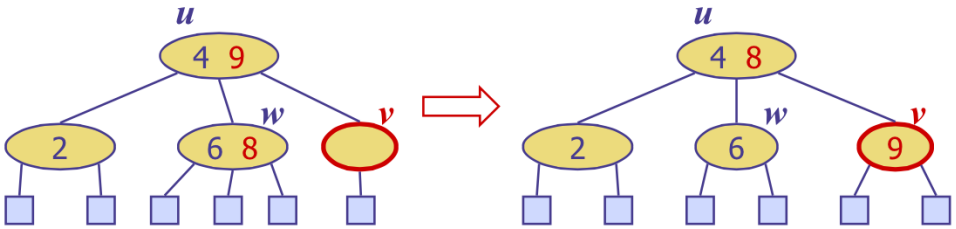


2. Transfer:

   Wenn der sibling w ein 3 oder 4-Node ist, können wir einen Transfer durchführen.

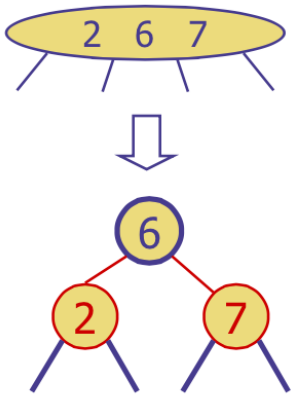
   1.) Wir verschieben ein Child von w nach v
   2.) Wir verschieben ein Key von u nach v
   3.) Wir verschieben ein Key von w nach u

   Nach einem Transfer sind alle Underflows behoben.

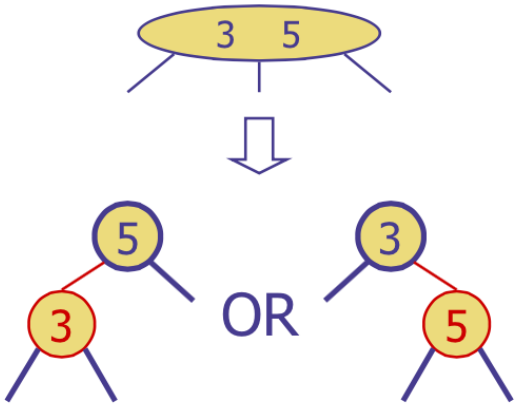


2,4 to Red Black Tree

Ein Red Black Tree kann recht simpel in einen 2,4 Baum umgewandelt werden.
Jeder schwarze Node gibt einen neuen Knoten im 2,4 Baum.
Die roten Kinder werden in diesen Knoten aufgenommen. Der schwarze Node steht dabei in der Mitte. Die Kinder 2 und 7 stehen links resp. rechts.



Manche Strukturen lassen sich nicht definitiv bestimmen. 5 wie auch 3 könnten hier der schwarze Node sein.



1.3 Graphen

1.3.1 BFS, Breadth-first search

Java Code:

```
// Not discovered node
private static final int FRESH = 0;
// Open node
private static final int OPEN = 1;
//Closed node
private static final int CLOSED = 2;
/**
 * Breadth-first search
 *
 * @param graph graph to be traversed
 * @param rootNr root node
 * @return array of predecessors
 */
public static GraphNodeIface[] breadthFirstSearch(GraphIface graph, int rootNr) {
    GraphNodeIface[] predecessors = new GraphNodeIface[graph.getVertexCount()]; //array
        of predecessors
    int[] state = new int[graph.getVertexCount()];
    for (int i = 0; i < state.length; i++) {
        state[i] = FRESH;
    }
    state[rootNr] = OPEN; //root node is open
    predecessors[rootNr] = null; //root has no predecessor

    Queue<GraphNodeIface> l = new LinkedList<GraphNodeIface>(); //queue, by replacing
        with stack the algorithm would behave as DFS
    l.add(graph.getNode(rootNr));

    while (!l.isEmpty()) {
        GraphNodeIface node = l.poll();
        List<GraphNodeIface> successors = node.getSuccessors();
        for (GraphNodeIface succ : successors) {
            if (state[succ.getId()] == FRESH) { //newly discovered node
                l.add(succ); //to be processed
                state[succ.getId()] = OPEN;
                predecessors[succ.getId()] = node;
            }
        }
        state[node.getId()] = CLOSED;
    }
    return predecessors;
}
```

1.3.2 DFS, Depth-first search

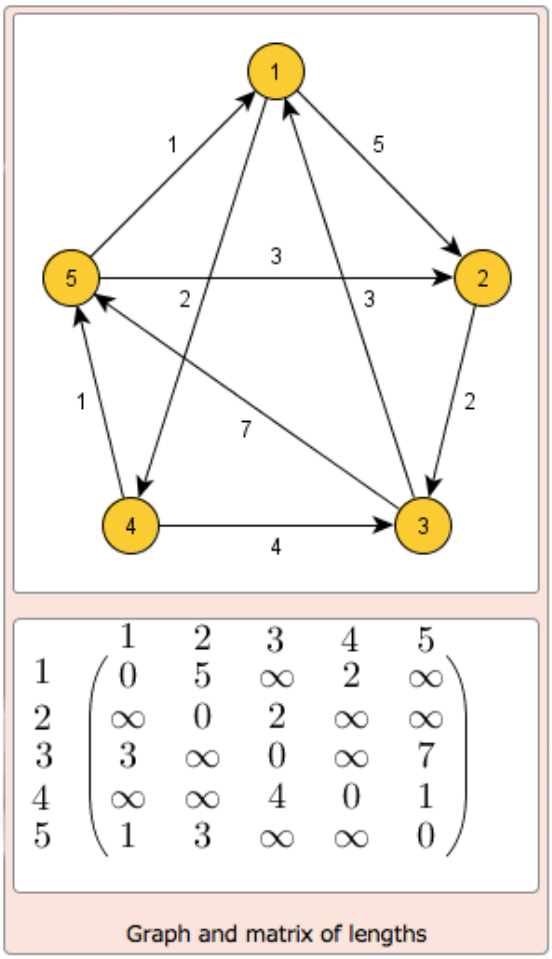
Java Code:

```
// Not discovered node
private static final int FRESH = 0;
// Open node
private static final int OPEN = 1;
//Closed node
private static final int CLOSED = 2;
/**
 * Recursive form of depth-first search
 *
 * @param graph
 */
public static void depthFirstSearch(GraphIface graph) {
    //node states
    int[] state = new int[graph.getVertexCount()];

    for (int i = 0; i < state.length; i++) {
        state[i] = FRESH;
    }
    //perform depth first search of all connected components
    for (int i = 0; i < graph.getVertexCount(); i++) {
        if (state[i] == FRESH) {
            doDFS(graph, i, state);
        }
    }
}

/**
 * Perform depth-first search
 *
 * @param graph graph
 * @param vertexNr node identifier
 * @param state array of node states
 */
private static void doDFS(GraphIface graph, int vertexNr, int[] state) {
    state[vertexNr] = OPEN;
    List<GraphNodeIface> succ = graph.getNode(vertexNr).getSuccessors();
    for (GraphNodeIface n : succ) {
        if (state[n.getId()] == FRESH) {
            doDFS(graph, n.getId(), state);
        }
    }
    state[vertexNr] = CLOSED;
}
```

1.3.3 Floyd-Warshall

Der Floyd-Warshall Algorithmus wird dazu verwendet, den kürzesten/ längsten Pfad für alle Node Paare, die keinen Cycle order negative Länge enthalten, zu finden.

Der Floyd Warshall Algorithmus nutzt eine Matrix D als Input.
In dieser Matrix werden die Distanzen/ Verbindungskosten zwischen den einzelnen Nodes vermerkt.
Besteht keine direkte Verbindung wird ∞ eingetragen.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \end{array}$$

Graph and matrix of lengths

Vertikale Achse: Verbindung von
Horizontale Achse: Verbindung zu

Beispiel 1.

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

1. D_0
   Die Input Matrix

2. *Wir suchen die kürzeste Distanz und tragen Sie in die Matrix ein.*

Java Code:

```java
/**
 * Floyd-Warshall algorithm. Finds all shortest paths among all pairs of nodes
 * @param d matrix of distances (Integer.MAX_VALUE represents positive infinity)
 * @return matrix of predecessors
 */
public static int[][] floydWarshall(int[][] d) {
    int[][] p = constructInitialMatixOfPredecessors(d);
    for (int k = 0; k < d.length; k++) {
        for (int i = 0; i < d.length; i++) {
            for (int j = 0; j < d.length; j++) {
                if (d[i][k] == Integer.MAX_VALUE || d[k][j] == Integer.MAX_VALUE) {
                    continue;
                }

                if (d[i][j] > d[i][k] + d[k][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                    p[i][j] = p[k][j];
                }

            }
        }
    }
    return p;
}

/**
 * Constructs matrix P0
 * @param d matrix of lengths
 * @return P0
 */
private static int[][] constructInitialMatixOfPredecessors(int[][] d) {
    int[][] p = new int[d.length][d.length];
    for (int i = 0; i < d.length; i++) {
        for (int j = 0; j < d.length; j++) {
            if (d[i][j] != 0 && d[i][j] != Integer.MAX_VALUE) {
                p[i][j] = i;
            } else {
                p[i][j] = -1;
            }
        }
    }
    return p;
}
```

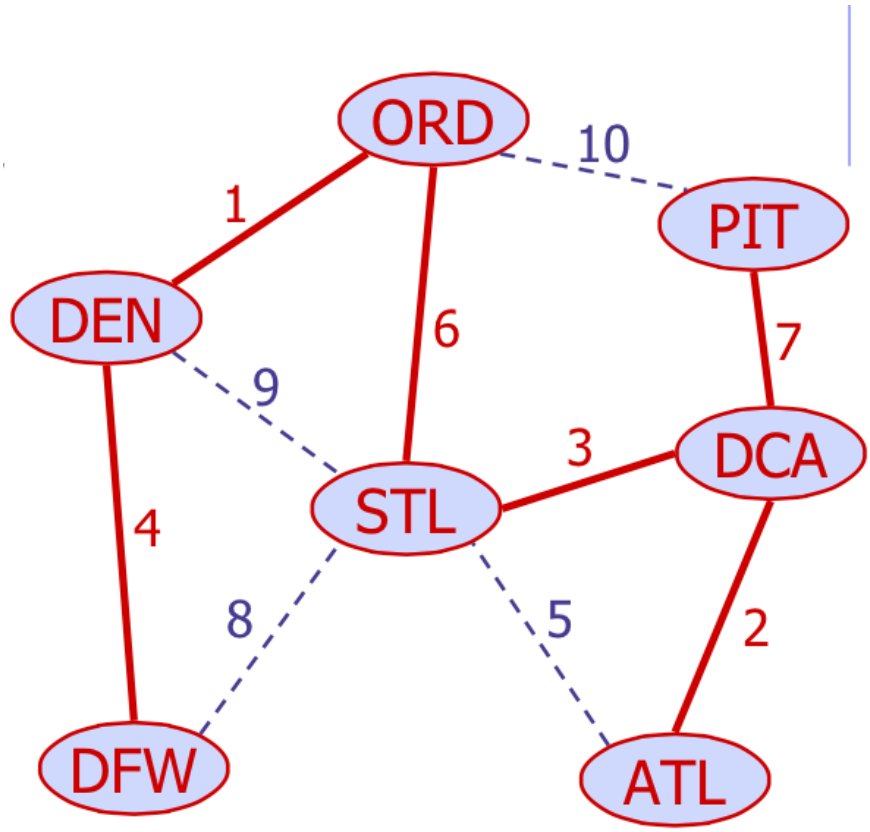
1.3.4 Minimum Spanning Trees (MST)

Jedes Element ist untereinander verbunden.
Es gibt keine Loops/ Kreisverbindungen.

Um einen MST zu bauen, darf bei jedem Element begonnen werden.
Wir suchen die Verbindung mit den niedrigesten Verbindungskosten.

Prim-Jarnik's

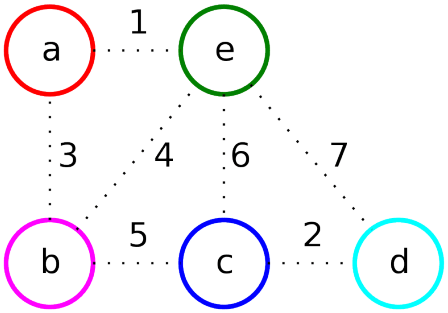
Beispiel 2. *Start bei **DFW**. Mit der nächste Verbindung gelangen wir entweder zu **DEN** oder **STL**. Wir wählen die Verbindung zu DEN, da die Verbindungkosten nur bei 4 liegen.*
Von DEN aus gesehen ist ORD die nächst günstigste Verbindung.
usw...



Kruskal

Die Ausgangslage

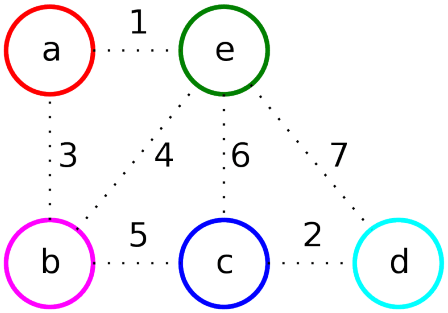
Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



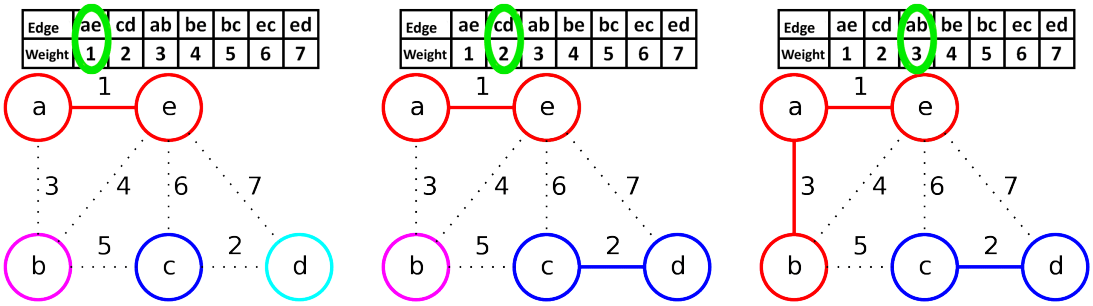
1. Sortiere die Edges nach ihrer Gewichtung, aufsteigend.

SORT THE EDGES

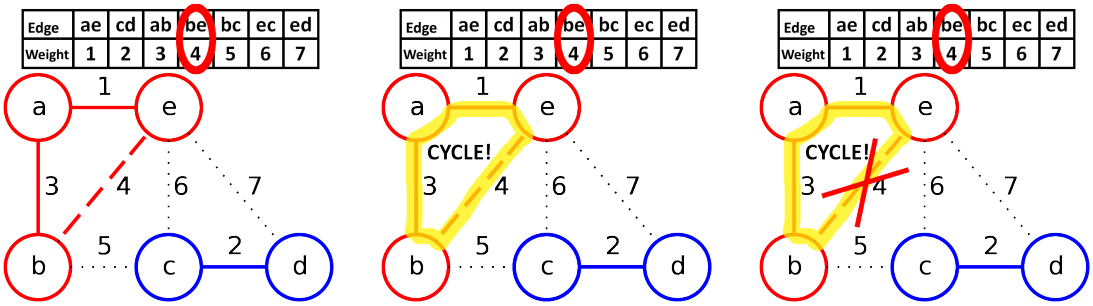
Edge	ae	cd	ab	be	bc	ec	ed
Weight	1	2	3	4	5	6	7



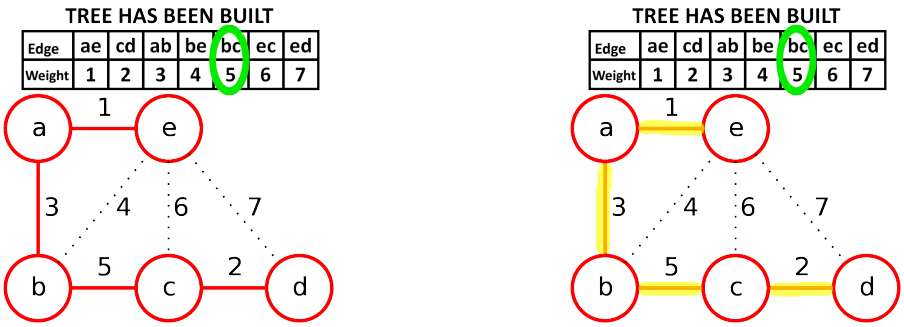
2. Nimm den kleinsten Edge.
   Überprüfe ob er zusammen mit dem bisher gebildeten Spanning Tree einen Cycle/ Kreis bildet. Wenn er keinen Kreis bildet, nimm diesen Edge in de Spanning Tree auf.



   Wenn er einen Kreis bildet verwirf diesen Edge.

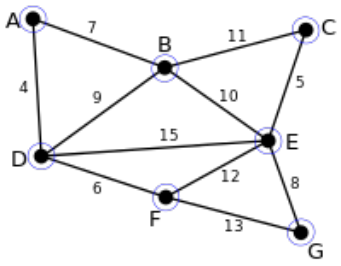


3. Wiederhole Schritt 2 bis der Spanning Tree aus **Anzahl Nodes -1** Edges besteht.



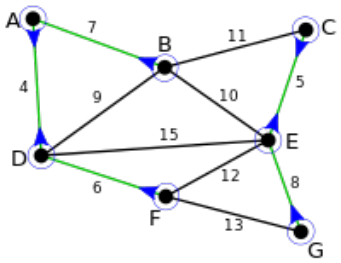
Baruvka

1. Jeder Node ist eine eigenständige Komponente/ Gruppe



Komponenten:
{A}
{B}
{C}
{D}
{E}
{F}
{G}

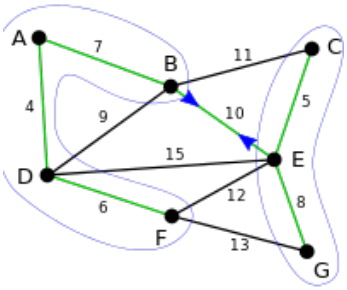
2. Wir wählen pro Komponente, die kleinste Verbindung.
Einige Verbindung werden doppelt ausgewählt (A,D) sowie (C,E).
2 Komponenten verbleiben.



Komponenten:
{A,B,D,F}
{C,E,G}

3. Pro Komponente suchen wir die kleinste Verbindung. Diese Verbinung darf nur zu einen Node führen, der nicht bereits in der Komponente enthalten ist. Aus diesem Grund wird die Verbindung (BE) ausgewählt. Die Verbindung (BD) wäre "kleiner", wird aber nicht berücksichtigt, da B und D bereits in der Komponent enthalten sind. Wir würden einen Circle generieren.

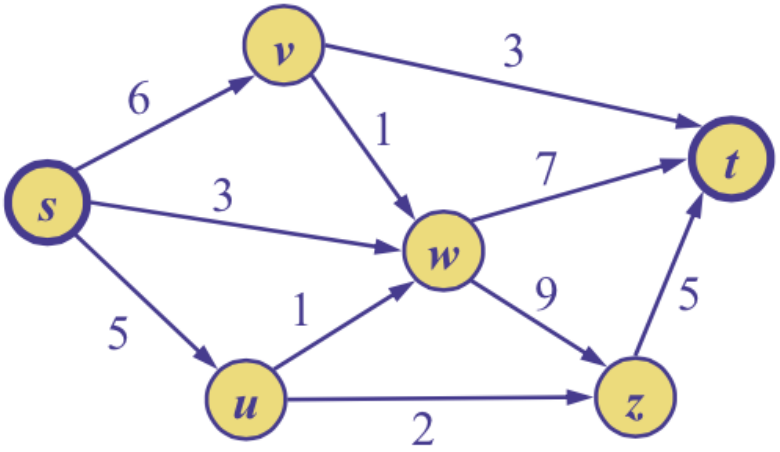
   Es verbleibt 1 Komponente. Der MST wurde erstellt.



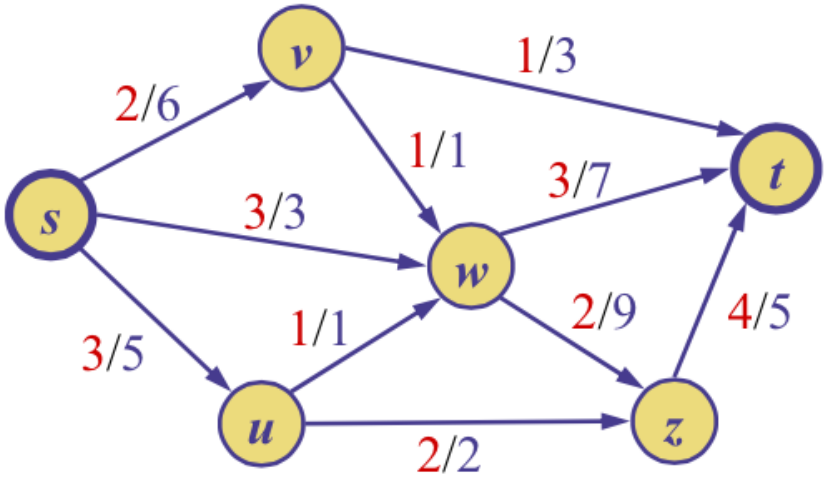
Komponenten:
{A,B,C,D,E,F,G}

1.3.5 Flow Network

Die Zahl auf den Kanten (Edgses) gibt die Kapazität an.
Punkt s und t werden Quelle und Abfluss genannt. Diese zwei Punkte besitzen keinen Zu- resp. Abfluss.

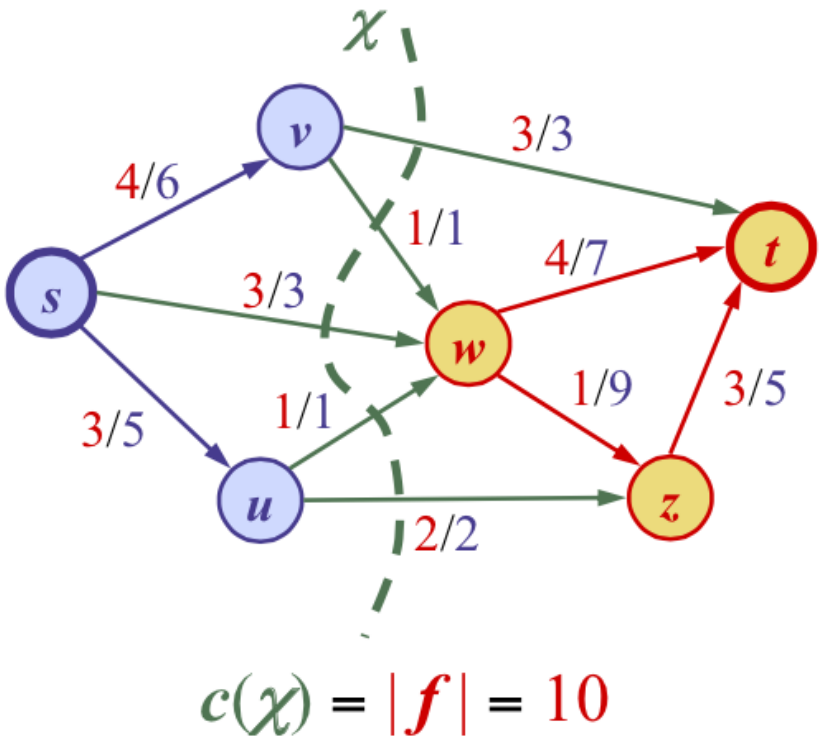


Die erste rote Zahl an einer Kante, gibt den tatsächlichen Fluss/ Flow $f$an. Der Fluss darf dabei die Kapazität c der Verbindung nicht übersteigen. Der Betrag des Flusses $|f|$ sagt aus, wie viel aus der Quelle fliesst. Ein Rückfluss ist ebenfalls möglich, wenn in der entsprechenden Kante genügend Kapazität verfügbar ist.



1.3.6 Minimal Cut

Bei einem Minimal Cut ist die Kapazität minimal. Der Fluss ist maximal. Wir wählen die Wege aus, die komplett gefüllt sind.



Edmonds-Karp

Mittels Edmond-Karp Algorithmus finden wir den Maximalen Fluss in einem Fluss-Graphen.

Wir erstellen eine Liste um Flows "zwischenzuspeichern". Wir nennen die Liste $FlowList$

Wir suchen uns den kürzesten Weg von s zu t.
Haben mehrere Wege die selbe Länge, wählen wir einfach einen aus. Den gefundenen Weg nennen wir w_1

Wir suchen die Verbindung/ Edges in w_1 mit der kleinsten Kapazität c.
Wir addieren die kleinste Kapazität c zu allen Edges als flow.
Ist der flow f einer Verbindung gleich der Kapazität c einer Verbindung, scheidet diese aus. Diese Verbindung/ Rohr kann nicht mehr verwendet werden. Das Rohr ist voll.

Wir können auch in gegenrichtung zu einem Pfeil transportieren. Der Flow auf diesem Weg wird -gerechnet.
Wir schreiben alle Punkt von w_1 in der Flowliste auf. Dahinter schreiben wir die Transportierte Menge, also c. Wir haben damit einen von x Augmenting Paths gefunden.
Wir suchen uns den nächsten "Augmented Path",also den nächst kürzesten Weg von s zu t.

Der Ablauf beginnt von vorne.
Kann kein Weg mehr gefunden werden ist der Algorithmus fertig.
Wir addieren nun alle Zahlen auf der Flowlist und erhalten damit den maximalen Flow des Graphen.

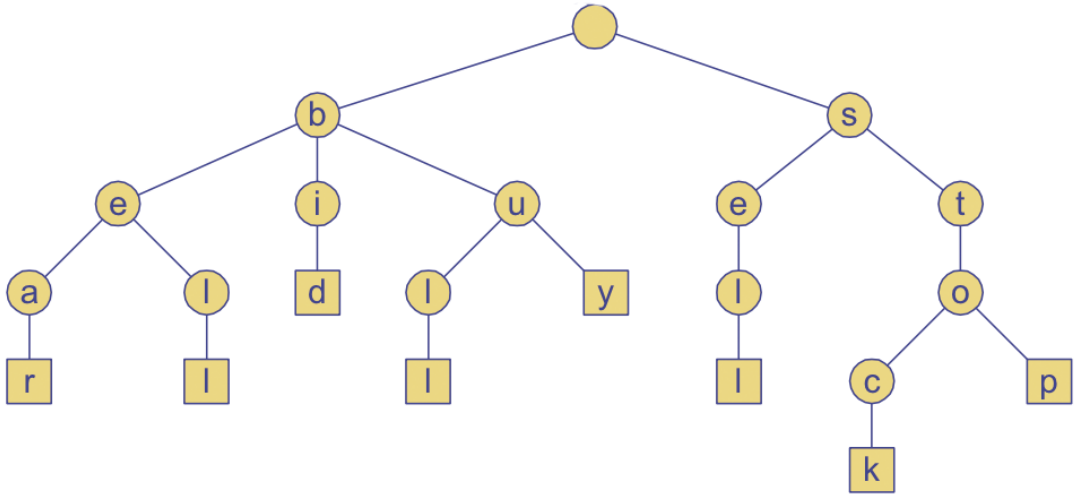
1.4 Tries

1.4.1 Standard Trie

Eingeschaften:

- Jeder Node, ausser dem root, ist mit einem Buchstaben beschriftet
- Die Kinder werden dem alphabet sortiert
- Der Pfad von einem Leaf zum root, stellt ein Wort dar

Beispiel 3. *S ={ bear, bell, bid, bull, buy, sell, stock, stop }*



Laufzeit

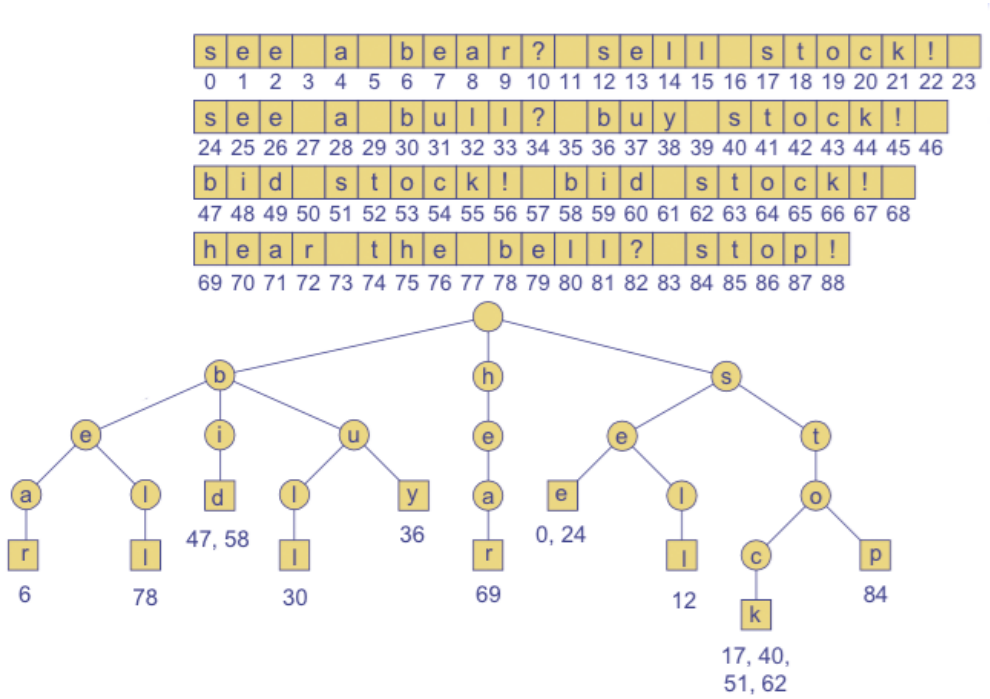
Ein Standard Trie braucht $O(n)$ Zeit und unterstützt suchen, einfügen und löschen mit einer Laufzeit von $O(dm)$ wobei

- n, die Anzahl aller Strings in S
- m, grösse des string Parameter der Operation (einfügen...)
- d, Grösse des Alphabetes

ist

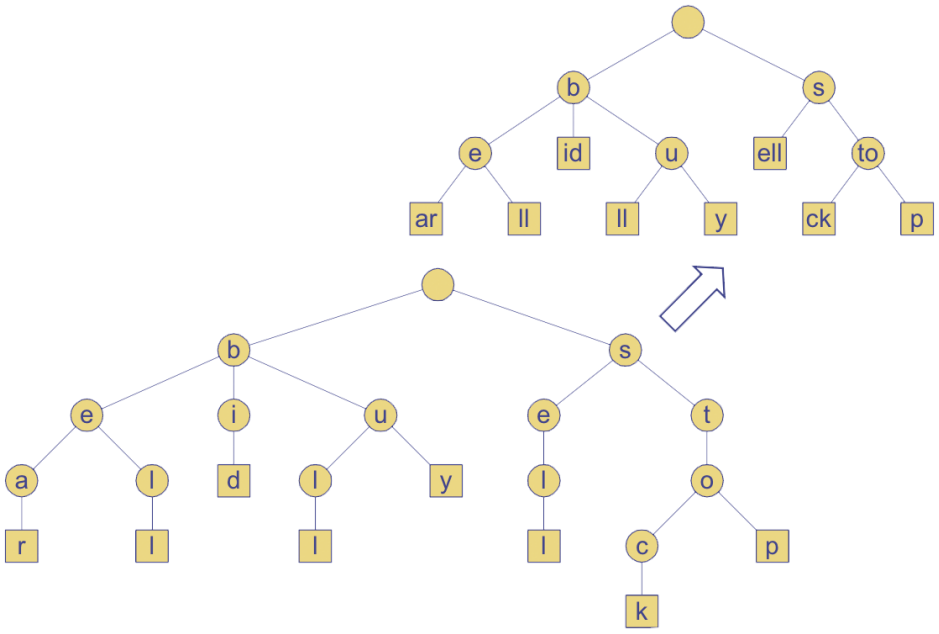
Word Matching

In jedem Leaf wird das vorkommen des Strings gespeichert.
Damit lässt sich inner kürzester Zeit feststellen, ob und wo der gesuchte String im Text vorkommt.



1.4.2 Compressed Trie

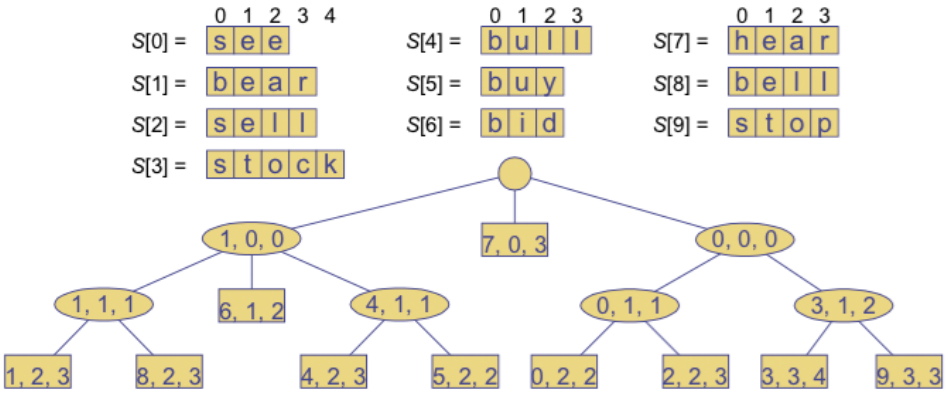
Pfade bei denen es keine Abzweigungen gibt, werden zusammengefasst.



Compact Representation

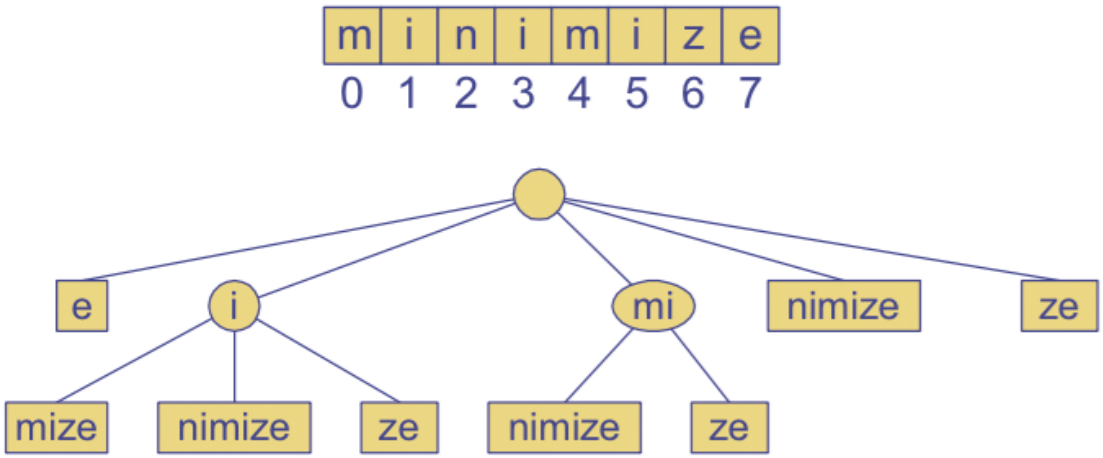
Hier werden nur die Indices der Strings gespeichert. Nicht der Substring.

1. Zahl Gibt an aus welchem Array der Substring kommt
2. Zahl Gibt den Start Index an
3. Zahl Gibt den End Index an



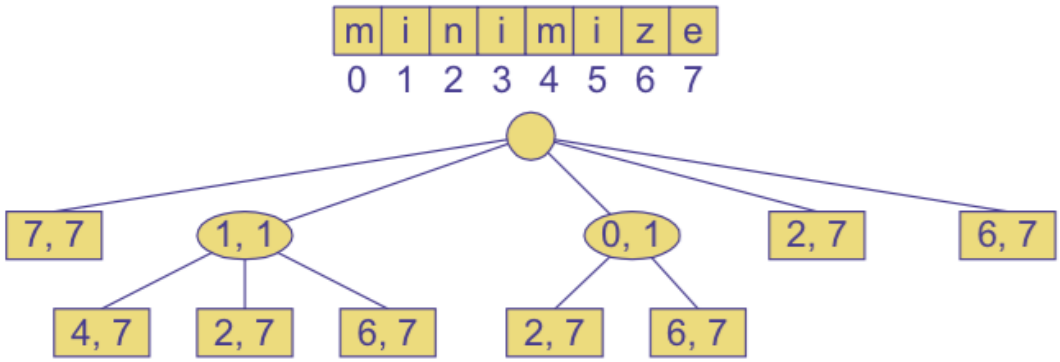
1.4.3 Suffix Trie

Der Suffix Trie des strings ”minimize”.



Compact Representation

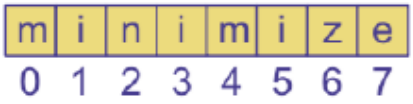
Darstellung durch indices



Erstellung

Wir gehen vom String ”minimize” aus.

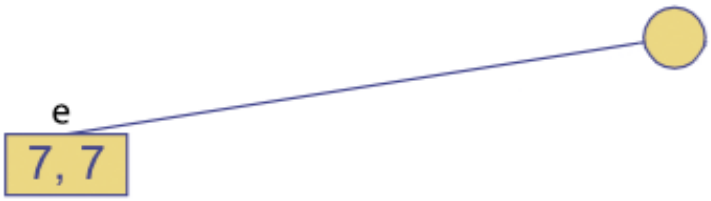
1. Wir kennzeichen jeden Buchstaben mit seinem Index



2. Wir erstellen einen Root Knoten

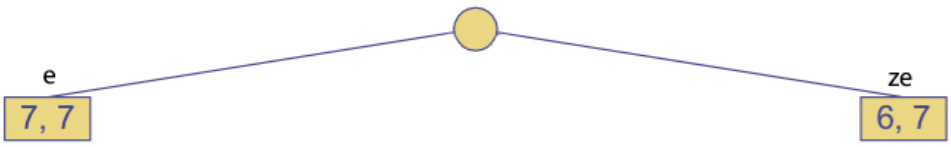
3. Wir beginnen bei letzten Buchstaben **e** index 7.
   Wir suchen auf der obersten Ebenen unseres Suffix Tries nach dem Buchstaben e. Es gibt noch keinen Knoten.

   Wir erstellen einen Knoten **e mit dem Index 7,7**



4. Auf zum nächsten Suffix **ze** Wir suchen auf der obersten Ebenen unseres Suffix Tries nach dem Buchstaben z. Es gibt noch keinen Knoten.

   Wir erstellen einen Knoten **ze mit dem Index 6,7**



...

5. Auf zum nächsten Suffix **imize** Wir suchen auf der obersten Ebenen unseres Suffix Tries nach dem Buchstaben i.
   Es gibt bereits einen Knoten der mit i beginnt. Der Knoten ize 5,7.

6. Wir überprüfen den nächsten Buchstaben **m** im suffix imize.
   Der Buchstabe m kommt im Knoten ize 5,7 nicht vor
   Wir unterteilen den Knoten.
   Der Knote **ize 5,7** wird zu **i 3,3**
   Wir hängen die beiden Suffixe **mize 4,7** und **ze 6,7** an
   ...

7. Auf zum nächsten Suffix **inimize**. Wir suchen auf der obersten Ebenen unseres Suffix Tries nach dem Buchstaben i.
   Es gibt bereits einen Knoten der mit i beginnt. Der Knoten i 3,3.

8. Wir überprüfen den nächsten Buchstaben **n** im suffix inimize.
   Der Buchstabe n kommt im Knoten imize nicht vor
   Wir unterteilen den Knoten.
   Der Knote **i 3,3** wird zu **i 1,1**
   Wir hängen den Suffix **nimize 2,7** an

9. Auf zum nächsten Suffix **minimize** Wir suchen auf der obersten Ebenen unseres Suffix Tries nach dem Buchstaben m.
   Es gibt bereits einen Knoten der mit m beginnt. Der Knoten mize 4,7.

10. Wir überprüfen den nächsten Buchstaben **i** im suffix minimize.
    Der Buchstabe i kommt im Knoten mize 4,7 vor
    Wir unterteilen den Knoten nicht.

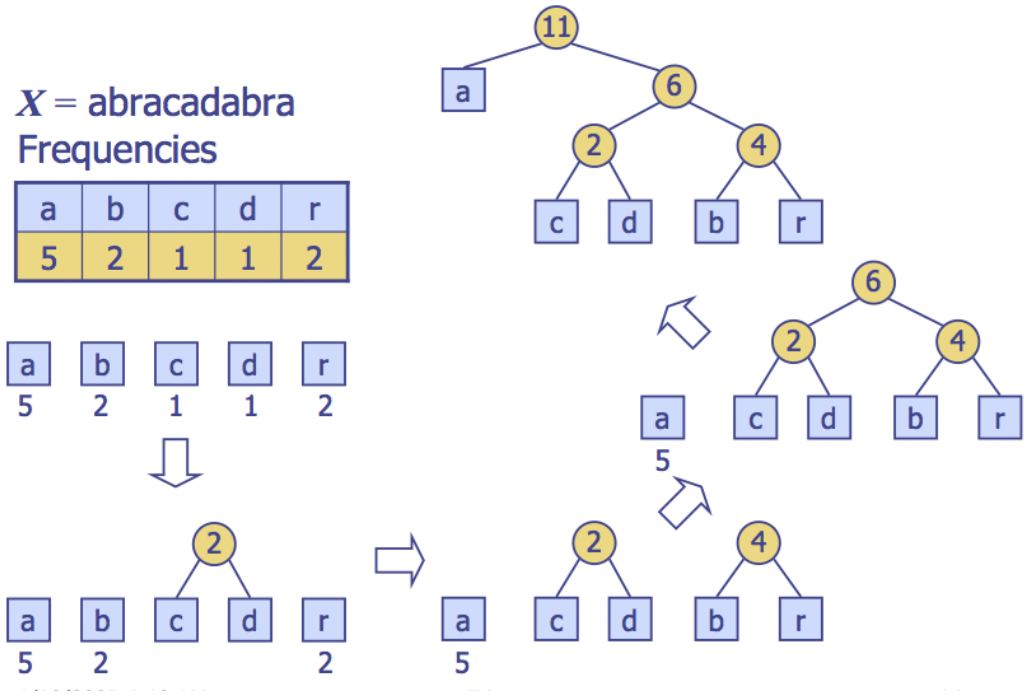
11. Wir überprüfen den nächsten Buchstaben **n** im suffix minimize.
    Der Buchstabe n kommt im Knoten mize 4,7 nicht vor
    Wir unterteilen den Knoten.
    Der Knoten **mize 4,7** wird zu **mi 0,1**
    Wir hängen die beiden Suffixe **ze 6,7** und **nimize 2,7** an

12. Wir haben alle Suffixe zum Trie hinzugefügt.

1.4.4 Encoding Tree (Huffmann Algorithm)

Wir zählen das vorkommen jedes Buchstabens und erstellen eine Frequencies Tabelle.

Wir Addieren immer den kleinsten mit dem zweikleinsten Wert.
Existieren zwei gleich keine Werte würde ich der ästetik halber die Zahl nehmen, welche sich weiter Rechts befindet. Auf das Resultat hat die Wahl aber keine Auswirkung. Zudem ist es einfacher zu zeichnen.



Binäre Grösse

Um die Grösse eines Huffmann Trees zu berechnen geht man wie folgt vor.

Buchstaben die sehr oft vorkommen stehen beim Encoding Tree weiter oben.
Diese Buchstaben werden mit einem kürzeren binären Code kodiert.
Buchtstaben die seltener vorkommen stehen weiter unten.
Diese Buchstaben werden mit einem längeren binären Code kodiert.
Durch dieses Vorgehen wird beim kodieren viel platz eingespart.

Die erste Ebene (von oben gesehen) hat den Wert 0Bit.
Die darunterliegende 1Bit usw.

Wir berechnen die Grösse, in dem wir die Häufigkeiten mit dem Bit Wert der Ebene auf der Sie sich befinden multiplizieren. Die einzelnen Ergebnisse werden addiert.

Im Obigen Beispiel ergibt sich die folgende Rechnung

$$Size = 1 * 5 + 3 * 1 + 3 * 1 + 3 * 2 + 3 * 2 = 5 + 3 + 3 + 6 + 6 = 23Bit$$

1.5 Pattern Matching

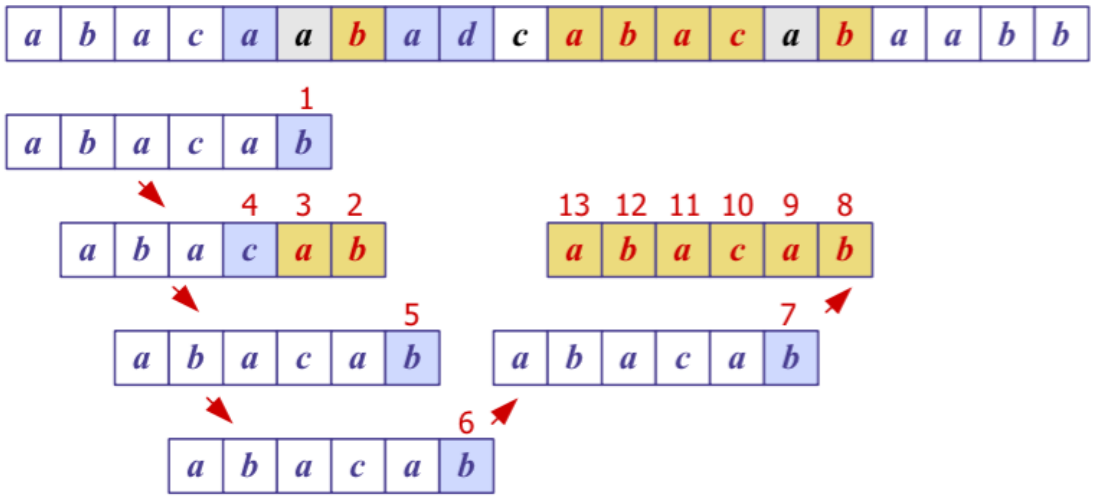
1.5.1 Brute Force

Überprüft jedes Zeichen miteinander.
Wenn ein Zeichen aus Pattern und Text übereinstimmen wird auch das nächste überprüft.

```
function bruteForceMatch(text, pattern)
{
	for(int i = 0; i < text.length()-pattern.length(); i++)
	{
		int j = 0;
		while(j < pattern.length() & text[i+j] == pattern[j])
		{
			if(j == pattern.length())
			{
				// we found a match
				return ''Match at''+i;
			}
			j++;
		}
		return null;
	}
}
```

1.5.2 Boyer Moore

Gegeben ist der untenstehende Text T sowie das Patter P ”abacab”.
Wir vergleichen das letzte Zeichen vom Pattern ”b” mit dem Zeichen vom Text, welches sich an der gleichen Stelle befindet.
Tritt an der Stelle T[i] = ”x” ein Mismatch auf, suchen wir das letzte Auftretenvon ”x”’ in Pattern P.
Wichtig: Wir suchen nur nach dem letzten auftreten!
Existiert dieses Zeichen in unserem Pattern nicht, dürfen wir um die Länge vom Pattern nach Rechts verschieben.
Existiert dieses Zeichen, verschieben wir das Pattern um soviel dass das letzte auftreten von ”x” an der Stelle T[i] steht.



1. Wir vegleichen den letzten Buchstaben von P mit dem Buchstaben aus T.
   b und a sind nicht gleich.
   Ein Mismatch, suchen wir die ”last Occurence”/ das letzte Auftreten von a in unserem Pattern.
   ”a” existiert an Stelle 5 unseres Patterns. Die Länge des Pattern beträgt 6.
   Wir dürfen das Pattern also nur um 1 nach Rechts verschieben.

2. Jetzt finden wir eine Übereinstimmung. b ist identisch mit b.
   Wir verschieben P nicht.

3. Der Pointer rutscht nur eines nach hinten. Überprüfen wir ob auch der zweite Buchstabe passt.
   auch a und a sind identisch.
   Der Pointer rutscht wider nach hinten.

4. a und c sind nicht identisch. Ein Mismatch, wieder suchen wir die ”last Occurence”/ das letzte

Auftreten von a in unserem Pattern.
a befindet sich gerade links, also dürfen wir wieder nur um eines verschieben.

5. Ein Mismatch, wieder suchen wir die ”last Occurence”/ das letzte Auftreten von a in unserem Pattern.
   a befindet sich gerade links, also dürfen wir wieder nur um eines verschieben.

6. Wieder ein Mismatch.
   In unserem Pattern existiert kein ”d”.
   Nun dürfen wir um 6 Stellen schieben.

7. Ein Mismatch, wieder suchen wir die ”last Occurence”/ das letzte Auftreten von a in unserem Pattern.
   a befindet sich gerade links, also dürfen wir wieder nur um eines verschieben.

8. Ein Match.
   Der Pointer rutscht nach Links.

9. Ein Match.
   Der Pointer rutscht nach Links.

10. Ein Match.
    Der Pointer rutscht nach Links.

11. Ein Match.
    Der Pointer rutscht nach Links.

12. Ein Match.
    Der Pointer rutscht nach Links.

13. Ein Match.
    Der Pointer kann nicht mehr nach Links rutschen.

1.5.3 Knuth-Morris-Pratt KMP

Der KMP Algorithmus ist in 2 Schritte unterteilt

Präfix Match Tabelle

Unser Pattern lautet

p	a	b	a	b	a	c	a

Wir erstellen pro Schritt **q** eine Liste von Prä und Suffixen.
Wir vergleichen die Su- und Präfixe.
Wenn es eine Übereinstiimung gibt, notieren wir die Anzahl der Zeichen.
Gibt es mehrere Übereinstimmungen, gewinnt der String mit den meisten Buchstaben.

1. q = 1
   String: a
   Wir erstellen eine Partial Match Tabelle.
   Π[0] ist immer 0.

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0						

2. q = 2
   String: ab
   Präfix: a
   Suffixe: b
   Keine Übereinstimmung
   Wir notieren 0 an der Stelle Π[2]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

3. q = 3
   String: aba Präfix: a, ab
   Suffixe: a, ba
   Eine Übereinstimmung. String a hat die Länge 1.
   Wir notieren 1 an der Stelle Π[3]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

4. q = 4
   String: abab
   Präfix: a, ab, aba
   Suffix: b, ab, bab
   Eine Übereinstimmung. String ab hat die Länge 2.
   Wir notieren 2 an der Stelle Π[4]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

5. q = 5
   String: ababa
   Präfix: a, ab, aba, abab
   Suffix: a, ba, aba, baba
   Zwei Übereinstimmung. String aba gewinnt, er hat die Länge 3.
   Wir notieren 3 an der Stelle Π[4]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

6. q = 6
   String: ababac
   Präfix: a, ab, aba, abab, ababa
   Suffix: c, ac, bac, abac, babac
   Keine Übereinstimmung.
   Wir notieren 0 an der Stelle Π[6]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	

7. q = 7
   String: ababac
   Präfix: a, ab, aba, abab, ababa,ababac
   Suffix: a, ca, aca, baca, abaca, babaca
   Eine Übereinstimmung. String ab hat die Länge 1.
   Wir notieren 1 an der Stelle Π[7]

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

Algorithmus

Das Pattern P[0] steht initial bei T[0] Text: bacbababaabcbab
Pattern: abababca

1. Wir erstellen die Partial Match Table

```
char:  | a | b | a | b | a | b | c | a |
index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
value: | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 |
```

2. i = 0
   Wir verschieben P um 1, bis wir einen Partial Match finden.

3. i = 1
   Wir haben einem Match gefunden

```
bacbababaabcbab
 |
 abababca
```

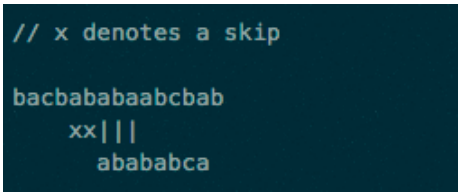
   Wir prüfen wie viele Zeichen matchen.
   Es matched nur 1 Zeichen, a.
   Wir suchen in der PMT nach dem Index
   $partialMatchLength - 1 \rightarrow table[1-1] \rightarrow table[0] \rightarrow table[0] = 0$
   das heisst wir können keine Zeichen überspringen.
   Wir verschieben um 1 bis wir wieder einem Match finden.
   ...

4. i = 4
   Wir haben einem Match mit 5 Zeichen gefunden

```
bacbababaabcbab
    |||||
    abababca
```

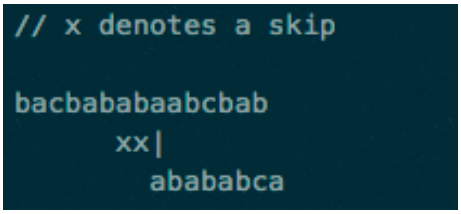
   Wir suchen in der PMT nach dem Index
   $partialMatchLength - 1 \rightarrow table[5-1] \rightarrow table[4] \rightarrow table[4] = 3$
   Das heisst wir können
   $partialMatchLength - table[partialMatchLength - 1] \rightarrow 5 - table[4] \rightarrow 5 - 3 \rightarrow 2$
   Zeichen überspringen.

5. i = 7 Wir haben 2 Zeichen übersprungen.
   Wir haben einen Match von 3 Zeichen gefunden.



   Wir suchen in der PMT nach dem Index
   $partialMatchLength - 1 \rightarrow table[3-1] \rightarrow table[2] \rightarrow table[2] = 1$
   Das heisst wir können
   $partialMatchLength - table[partialMatchLength - 1] \rightarrow 3 - table[2] \rightarrow 3 - 1 \rightarrow 2$
   Zeichen überspringen.

6. i = 9
   Wir haben 2 Zeichen übersprungen.
   Das Pattern kann nicht mehr matchen.



   Algorithmus beendet

Der Vorteil von KMP ist, dass wir anhand der PMT Wissen, wo der nächste Match stattfinden wird. Somit können wir einzelne Vergleiche auslassen und sparen damit Rechenzeit.

1.6 Landau-Symbole

Landau-Symbole werden verwendet um das Asymptotische Verhalten einer Funktion zu beschreiben. Sie sagen dir wie schnell eine Funktion wächst oder kleiner wird.

g und f sind zwei Funktionen $g, f : \mathbb{N} \to \mathbb{R}$

Definition 1. $O(f)$
"gross O von f" oder "Big O von f"
$g \in O(f)$ falls gilt: die asymptotische Wachstumsrate von g ist höchstens ($\leq$) so gross wie jene von f.

exakt:

$$\exists n_0 \in \mathbb{N}, \exists c > 0 \text{ so dass } g(n) \leq c \cdot f(n), \forall n > n_0$$

Beispiel 4.

$$\begin{aligned} n \in O(n^2) \quad & \textit{denn } n_0 = 1, c = 1 \\ n^2 + 1 \in O(n^2) \quad & \textit{denn } n_0 = 1, c = 2 \\ 10n^2 \in O(n^2) \quad & \textit{denn } n_0 = 1, c = 11 \end{aligned}$$

Definition 2. $o(f)$
"klein o von f" oder "little o von f"
$g \in o(f)$ heisst: Die asymptotische Wachstumsrate von g ist echt kleiner ($<$) als jene von f.

exakt:

$$\forall c > 0, \exists n_0 \in \mathbb{N} \text{ so dass gilt: } c \cdot g(n) < f(n), \forall n > n_0$$

BILD TODO

Beispiel 5.

$$\begin{aligned} n \in o(n^2) \textit{ denn } \forall c > 0 \textit{ gilt } \exists n_0 > c \\ n^2 > cn \forall n > n_0 \end{aligned}$$

g wird also immer von der Funktion f überholt.

Definition 3. $\Theta(f)$
"Theta von f"
$g \in \Theta(f)$ heisst: g hat diesselbe asymptotische Wachstumrate wie f

exakt:

$$\begin{aligned} \exists c_1 > 0, \exists c_2 > 0 \text{ und } n_0 \in \mathbb{N} \\ \text{so dass gilt} \\ c_1 f(n) > g(n) > c_2 f(n), \forall n > n_0 \end{aligned}$$

Beispiel 6.

$$\begin{aligned} 5n^2 + 1 \in \Theta(n^2) \\ c_1 n^2 > 5n^2 + 1 > c_2 n^2 \forall n > 1 \textit{wenn } c_1 = 0, c_2 = 1 \end{aligned}$$

Abkürzungen W(f) asymptotisch prozentuales Wachstum von f
W = Wachstum

$g \in o(f)$	W(g) $<$ W(f)
$g \in O(f)$	W(g) $\leq$ W(f)
$g \in \Theta(f)$	W(g) $=$ W(f)
$g \in \Omega(f)$	W(g) $\geq$ W(f)
$g \in \omega(f)$	W(g) $>$ W(f)

Beispiel 7. *Zeige:* $n + 100 \in o(n^2)$

$$\begin{aligned}
&\textit{zu zeigen ist: } \forall c > 0 \exists n_0 \in \mathbb{N} \textit{ so dass gilt} \\
&c(n+100) < n^2 \forall n > n_0 \\
&c(n+100) < n^2 \rightarrow \textit{Gleichheit bei } n_0 \\
&c(n_0+100) = n_0^2 \\
&n_0^2 - cn_0 - 100c = 0 \\
n_0 = \frac{-b \pm \sqrt{b^2-4ac}}{2a} &\wedge an_0 + bn_0 + c = 0 \rightarrow \\
&n_0 = \frac{+c \pm \sqrt{c^2+4c100}}{2}
\end{aligned}$$

1.7 Einige Theoreme

- $g \in o(f)$
  $f + g \in \Theta(f)$

  Beispiel 8.

  $$10n^7 + 4n^5 + n^2 + 100 \in \Theta(n^7)$$

  Gleiche Wachstumsrate. Eine Addition der Funktion hat keine Auswirkung auf die Wachstumsrate.

- $\forall a > 0$
  $af \in \Theta(f)$
  Der Faktor hat keine Auswirkung

- $0 < a < b$
  $\rightarrow n^a \in o(n^b)$
  n^a steigt immer weniger als n^b

- $\log(n) \in o(n)$
  Der log ist immer flacher / n wird $\log(n)$ immer "überholen". BILD TODO LOG

  $$c \cdot \log(n) < n$$
  Umkehrfunktion von
  $$e^{c \cdot \log(n)} < e^n$$
  $$n^c < e^n$$

  Logarithmen sind prozentual zueinander sie lassen sich mit einer Konstanten ineinander umrechnen.

In the following f, g, h are positive monotonically growing functions from $\mathbb{N}$ to $\mathbb{R}$. Furthermore we define $(f+g)(n) = f(n) + g(n)$ and $(f \cdot g)(n) = f(n) \cdot g(n)$.

1. $\Theta(f) \cap o(f) = \{\}$
2. $g \in O(f) \Rightarrow (f+g) \in \Theta(f)$
3. $a > 0 \Rightarrow a \cdot f \in \Theta(f)$
4. $0 < a < b \Rightarrow n^a \in o(n^b)$
5. $g \in \Theta(f) \Rightarrow h \cdot g \in \Theta(h \cdot f)$
6. $a, b > 0 \Rightarrow \log_a(n) \in \Theta(\log_b(n))$
7. $a > 0, n \in \mathbb{N} \Rightarrow \log(n) \in o(n^a)$ even for $a < 1$ (!)
8. $a > 0, n \in \mathbb{N} \Rightarrow n^a \in o(\exp(n))$
9. $0 < a < b \Rightarrow a^n \in o(b^n)$

1.8 Funktionstabelle nach asymp. Wachstum

1	1
2	$\log^k(n) \wedge k \geq 1$
3	$\sqrt{n}$
4	n
5	$n \log(n)$
6	$n^{1+e} \wedge 1 > e > 0$
7	n^2
8	$n^2 \log(n)$
	...
9	2^n
10	$n!$

9 & 10 haben eine explosive Wachstumsrate und sind daher inefizient.

1.9 Andere Notationen

$$f \in O(g)$$
$$f \text{ is } O(g)$$
$$f = O(g)$$
$$f + o(f) < \Theta(f)$$

1.10 Zeitkomplexität

1.10.1 Erkennen

Aus Programmcode kann die Zeitkomplexität des programmierten Algorithmus abgelesen werden.

Anzahl der Schleifen

Eine Schleife (for, while...) lässt auf einen Algorithmus mit der Zeitkomplexität $O(n^x)$ schliessen. Wobei x für die Anzahl der Schleifen steht.

Beispiel 9. *Java Code:*

```
public static int[][] floydWarshall(int[][] d) {
        int[][] p = constructInitialMatixOfPredecessors(d);
        for (int k = 0; k < d.length; k++) {
                for (int i = 0; i < d.length; i++) {
                        for (int j = 0; j < d.length; j++) {
                                if (d[i][k] == Integer.MAX_VALUE || d[k][j] ==
                                    Integer.MAX_VALUE) {
                                continue;
                                }

                                if (d[i][j] > d[i][k] + d[k][j]) {
                                d[i][j] = d[i][k] + d[k][j];
                                p[i][j] = p[k][j];
                                }
                        }
                }
        }
        return p;
}
```

Es gibt 3 Schleifen.
Die Elemente werden 3 mal miteinander verrechnet, anders ausgedrückt $n * n * n$. Was einer Zeitkomplexität von $O(n^3)$ entspricht.

Division Durch 2

Eine Division durch 2 lässt auf einen Algorithmus mit der Zeitkomplexität $O(logn)$ schliessen

1.10.2 Laufzeit

Kann man mit Kenntnis der Zeitkomplexität auch die Laufzeit voraussagen?
Zeitkomplexität sagt nur etwas über das relative Anwachsen der Laufzeit aus.

Wenn man annimmt, dass sich die laufzeit im asymptotischen Bereich befindet und man weiss, dass die mittlere Laufzeit $T(n)$ in der Klasse $\Theta(f(n))$ befindet, dann gilt:

$$\frac{T(n_1)}{T(n_2)} = \frac{f(n_1)}{f(n_2)}$$

Beispiel 10. $n \rightarrow$ *Anzahl Zahlen*

1. *Bubblesort*
   Eine Homogene Funktion

$$
\begin{aligned}
T(n) \in \Theta(n^2) & \\
sei\ T(1000) &= 1ms \\
T(10^6) &= T(1000) \cdot \frac{(10^6)^2}{(10^3)^2} \\
&= 1ms \cdot 10^6 \\
&= 1000s
\end{aligned}
$$

2. *Quicksort*
   keine Homogene Funktion

$$
\begin{aligned}
f(n) \in \Theta(n \log(n)) & \\
sei\ T(1000) &= 1ms \\
T(10^6) &= T(1000) \cdot \frac{10^6 \cdot \log(10^6)}{10^3 \cdot \log(10^3)} \\
&= 1ms \cdot 10^3 * 2 \\
&= 2s
\end{aligned}
$$

Kapitel 2

Basic Data Structures

2.1 Stacks

push(object)	inserts an element on top of the stack
pop()	removes the top element of the stack
top()	returns the last inserted element
size()	number of elements
isEmpty	check if stack is empty

2.1.1 Verdoppelung eines Arrays

Ist das Array voll, kann entweder eine Exception geworfen oder das Array vergrössert werden. Um das Array zu vergrössern, gibt es 2 Ansätze:

- Inkrement Methode
  Vergrösserung um eine Konstante c
- Verdoppelung
  Verdoppelung der Arraygrösse

Wir vergleichen die Mathoden. Wir analysieren dazu die Zeit $T(n)$ die verwendet wird um n push Operationen durchzuführen.
Wir beginnen mit einem leeren Array mit der Grösse 1.
"amortized time" = $T(n)$ n

Inkrement Methode

Das Array wird $k = n$ c mal ersetzt.
Wir berechnen die totale Zeit

Verdoppelung

2.1.2 polnische Notation

Zu lösenden Aufgabe:

$$((3 + 5) \cdot 8 + 21) \cdot 3$$

Ausdruck in umbekehrter polnischer Notation zuerst Operanden dann Operator

Reihenfolge der Eingabe

$$3, 5, +, 8, \cdot, 21, +, 3, \cdot$$

Stack:

$$\begin{array}{r} 3, 5 \\ + \\ 8, 8 \\ \cdot \\ 64, 21 \\ + \\ 85, 3 \\ \cdot \\ 255 \end{array}$$

2.2 Queue, zirkuläres Array

enqueue(object)	insert an element at the end of the queue
dequeue()	removes and returnes the element at the front of the queue
front()	return the element at the front without removing it
size()	returns the number of elements in the queue

2.3 Linked List

Java implementation:

Kapitel 3

Kurzreferenz

3.1 Begriffe

asymptotisch	wenn n sehr gross ist
diskreter Ablauf	schrittweiser Ablauf
Partitionieren	Problem in kleinere Problme zerlegen
Rate	prozentualer Zuwachs

3.2 Laufzeiten

Ford- Fulkerson's	$O(\|f \cdot \|(n+m))$
Red Black Tree	A red-black tree storing n items has height O(log n)