

# C++ Algorithmen und Datenstrukturen

Link EN: <http://www.cplusplus.com/>

Link DE: <http://de.cppreference.com/w/>

## INHALTSVERZEICHNIS

1.	Präprozessor .....	3
2.	Kommando Linien Argumente argc & **argv oder *argv[ ] .....	4
3.	Laufzeitanalyse .....	5
4.	Zeitmessung während Laufzeit .....	6
5.	Iterativ Lösungsansatz .....	6
6.	Rekursiver Lösungsansatz .....	6
7.	Rule of three .....	8
7.1.	Klassendeklaration .....	8
7.2.	Kopierkonstruktor .....	8
7.3.	Zuweisungsoperator .....	9
8.	Vererbung mit Polymorphie .....	10
9.	Sortieren .....	13
9.1.	Bubble Sort .....	13
9.2.	Selection Sort .....	15
9.3.	Insertion Sort .....	17
9.4.	Merge Sort .....	19
9.5.	Quick Sort .....	21
9.6.	Randomisierter Quick Sort .....	25
10.	Suchen .....	26
10.1.	Binary Search .....	26
10.2.	Lineares suchen .....	26
11.	Linked List .....	28
12.	Containers .....	30
12.1.	Queue .....	30
12.2.	Stack .....	30
12.3.	Binary Tree .....	31
12.4.	Binary Search Tree .....	32
12.5.	Binary Tree Traversal .....	32
12.6.	Graph .....	33
12.7.	Graph ohne Gewichtung .....	34
12.8.	Graph mit Gewichtung .....	34
12.9.	Breath First Search für Graphen .....	35
12.10.	Depth First Search für Graphen .....	36
12.11.	Dijkstra's Algorithmus .....	37
13.	Templates .....	39
14.	Containers in C++ STL Library .....	41
14.1.	Beispiel Container Deque, Vector, List .....	41
14.2.	Beispiel Set / Map .....	42
15.	Algorithmen in C++ STL Library .....	43
16.	String Matching Algorithmen .....	44
16.1.	Naives String Matching .....	44
16.2.	Boyer Moore String Matching .....	45
16.3.	Knuth Morris Pratt (KMP) String Matching .....	45

# 1. Präprozessor

Mit der folgenden Anwendung wird ausgeschlossen, dass ein Header File mehrfach während der Laufzeit inkludiert wird. Somit kann sichergestellt werden, dass es immer nur eine „gültige“ Variante des Header File gibt, welche inkludiert ist.

Weil ein Header File mehrfach in verschiedenen cpp Files aufgerufen wird, merkt der Compiler beim precompilen dass die gleiche Datei mehrfach inkludiert ist und gibt einen Fehler „redefining oft a variable“. Wenn der Precompiler sieht, dass das File\_H bereits inkludiert ist, springt er automatisch an die Zeile mit #endif und beugt so einem mehrfachen inkludieren vor.

Beispiel:

```
#ifndef FILE_H
#define FILE_H

/* ... Declarations etc here ... */

#endif
```

## #ifdef

Mit der #ifdef-Direktive kann geprüft werden, ob ein Symbol definiert wurde. Falls nicht, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifdef-Direktive muss durch eine #endif-Direktive abgeschlossen werden

## #ifndef

Die #ifndef-Direktive ist das Gegenstück zur #ifdef-Direktive. Sie prüft, ob ein Symbol nicht definiert ist. Sollte es doch sein, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifndef-Direktive muss ebenfalls durch eine #endif-Direktive abgeschlossen werden

## #endif

Die #endif-Direktive schließt die vorhergehende #ifdef-, #ifndef-, #if- bzw #elif-Direktive ab

## 2. Kommando Linien Argumente argc & \*\*argv oder \*argv[ ]

argc = Argument counter und beinhaltet die Anzahl von Argumenten, die dem Programm beim Start übergeben wurden. Dabei handelt es sich um einen Integerwert.

Argv = Argument values /- vector in welchen die einzelnen Argumente stehen. Diese werden als Strings in einer Stringtabelle gespeichert. Im Argv[0] steht immer der Pfad und Dateiname, im Argv[0+n] die einzelnen Parameter die vor dem ausführen der Quelldatei übergeben werden.

Definieren der Paramter vor dem Programmstart:

Dem Programm werden Argumente beim Aufruf übergeben. Unter MS-Windows können hierfür die Eingabeaufforderung cmd.exe verwenden. Unter Linux genügt eine einfache Konsole bzw. Shell. Sofern eine Entwicklungsumgebungen (IDEs) verwenden wird, müssen die Kommandozeilenargumente anders übergeben werden. IDE bieten hierfür meistens beim Menü Ausführen noch ein Untermenü Parameter je nach IDE abhängig an, um die Argumente noch vor dem Programmstart festzulegen und zu übergeben.

Testprogramm „Kommandozeilenargumente.cpp“

```
#include <iostream>
using namespace std;

int main (int argc, char **argv){
    cout << argc << endl;

    for (int i = 0; i < argc; i++){
        cout << "argv[" << i << "] = " << argv[i] << endl;
    }

    cout<<"\n\n";
    return 0;
}
```

Ausgabe Testprogramm:

```
1
argv[0] = .....\\...\\ Kommandozeilenargumente.exe
```

Parameter mit IDE Dev-C++ 5.nn übergeben

Im Menü „Ausführen/Parameter“ können die Parameter an das Programm übergeben werden. Die Zahl argc der Parameter sind hierbei jeweils durch einen Leerschlag getrennt.

Beispiel: Hello World im Menü „Ausführen/Parameter“ eingeben und dann erneut „Compilieren und Ausführen“ der Quelldate „Kommandozeilenargumente.cpp“

Ausgabe Testprogramm:

```
3
argv[0] = .....\\...\\ Kommandozeilenargumente.exe
argv[1] = Hello
argv[2] = World
```

### 3. Laufzeitanalyse

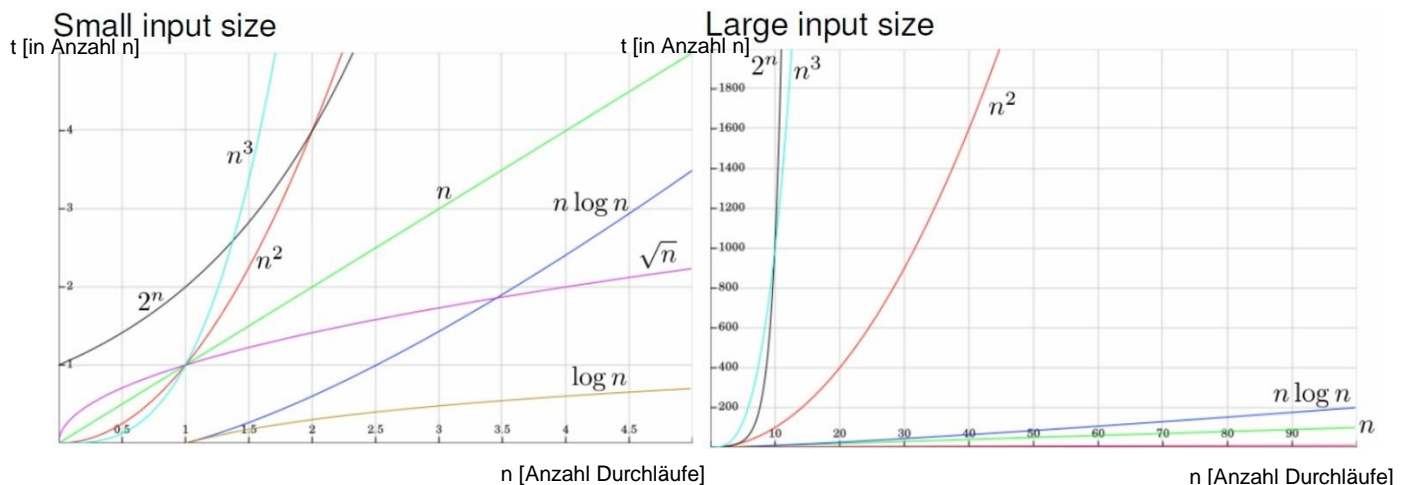
Mit dem grossen Buchstaben O wird die Laufzeit angegeben die ein Algorithmus braucht für die Lösung in Abhängigkeit von der Länge der Eingabe. Damit ist das Zeitverhalten des Algorithmus für eine potenziell unendlich große Eingabemenge gemeint.

Es interessiert also nicht der Zeitaufwand eines konkreten Programms auf einem bestimmten Computer, sondern viel mehr, *wie* der Zeitbedarf wächst, wenn mehr Daten zu verarbeiten sind, also z. B. ob sich der Aufwand für die doppelte Datenmenge verdoppelt oder quadriert (Skalierbarkeit). Die Laufzeit wird daher in Abhängigkeit von der Länge  $n$  der Eingabe angegeben und für immer größer werdende  $n$  unter Verwendung der Groß-O-Notation abgeschätzt.

$O(1)$	Konstante Laufzeit	<b>(In der realen Welt nicht möglich!)</b>
$O(n)$	Linear Laufzeit	
$O(\ln)$	Logarithmische Laufzeit	(Logarithmus naturalis)
$O(n * \ln)$	Logarithmische-Lineare Laufzeit	(Logarithmus naturalis)
$O(n^2)$	Quadratische Laufzeit	(2 Schleifen verschachtelt)
$O(n^3)$	Kubische Laufzeit	(3 Schleifen verschachtelt)
$O(c^n)$ ; $c > 1$	Exponentielle Laufzeit	

Beispiele Laufzeit anhand Laufzeit  $n = 1$  ms

$n$	$O(1)$	$O(n)$	$O(n^2)$	$O(n * \log n)$
10	1 ms	1 ms	$10^2 = 100$ ms	$10 * \ln \approx 23$ ms
$10^2$	1 ms	$10^2$ ms = 100 ms	$(10^2)^2 = 10'000$ ms	$10^2 * \ln \approx 460$ ms
$10^3$	1 ms	$10^3$ ms = 1'000 ms	$(10^3)^2 = 10^6$ ms	$10^3 * \ln \approx 6'907$ ms
$10^6$	1 ms	$10^6$ ms = 1'000'000 ms	$(10^6)^2 = 10^{12}$ ms	$10^6 * \ln \approx 13.8 * 10^6$ ms



## 4. Zeitmessung während Laufzeit

Die Laufzeit eines Programmes oder einer bestimmten Sequenz während dem ausführen vom Code kann wie folgt ermittelt werden:

<b>Stoppuhr</b>	<div><div><code>#include &lt;ctime&gt;</code></div><div><i>WICHTIG: immer Bibliothek integrieren</i> <i>Durch die Bibliothek ist ein neuer Datentyp clock_t verfügbar, für den genau gleich wie z.Bsp int eine Variable deklariert werden kann.</i></div><div><code>clock_t start, stop;</code><div>Deklarieren Variabel</div></div><div><code>start = clock();</code><div>Stoppuhr starten</div></div><div><code>..(zu messender Code)..</code></div><div><code>stop = clock();</code><div>Stoppuhr stoppen</div></div><div><code>cout &lt;&lt; ((double) stop - start) / CLOCKS_PER_SEC &lt;&lt; "s";</code> Zeit rechnen &amp; auswerfen</div></div>
-----------------	---

## 5. Iterativ Lösungsansatz

Beschreibt allgemein einen Prozess mehrfachen Wiederholens gleicher oder ähnlicher Handlungen zur Annäherung an eine Lösung oder ein bestimmtes Ziel. In der Informatik wird die Wiederholung selbst als Iteration bezeichnet, heisst das i in der Syntax „for ( int i = 0; i < x; i++ )“ steht für Iteration (Wiederholung)

## 6. Rekursiver Lösungsansatz

Bezeichnet man die Eigenschaft von Regeln, dass sie auf das Produkt, das sie erzeugen, von neuem angewandt werden können, wodurch potenziell unendliche Schleifen entstehen. In Mathematik, Logik und Informatik kann dies so ausgedrückt werden, dass eine Funktion in ihrer Definition selbst nochmals aufgerufen wird (*rekursive Definition*).

Es wird beim Rekursiven Ansatz versucht die Ausnahmen mittels if und return Syntax abzufangen. Der Selbstaufwurf der Formel geschieht damit, dass der Name der Funktion oder Methode in der Formel selbst aufgerufen wird.

Beispiel Fakultät:

Fakultät ist das Produkt aller natürlichen Zahlen von einer bestimmten Zahl n ausser Null.

Schreibweise n!

Berechnung:  $1 * 2 * 3 * 4 * 5 = 120$

```
#include <iostream>
using namespace std;

unsigned int fakultaet (unsigned int n){
    if (n<= 1) return 1;
    return n * fakultaet (n-1);
}

Int main ( int argc, char **argv){
    cout << fakultaet (5) << endl,
    return 0;
```

}

Regelausnahmen für Fakultät:

0! = immer 1 (ist so definiert!)

1! = immer 1

Heisst wenn 0 oder 1 vom Main an die Funktion übergeben wird, wird immer direkt 1 an das Main zurückgegeben ohne dass irgendwelche Zeit verloren geht.

Formel

$n * fakultaet(n - 1)$

Abarbeitung:

**Schritt**

fakultaet (5)

if (5 <= 1) return 1;

return 5 \* [ fakultaet ( 5 - 1 ) ]

if (4 <= 1) return 1;

return 4 \* [ fakultaet ( 4 - 1 ) ]

if (3 <= 1) return 1;

return 3 \* [ fakultaet ( 3 - 1 ) ]

if (2 <= 1) return 1;

return 2 \* [ fakultaet ( 2 - 1 ) ]

if (1 <= 1) return 1;

return 2 \* [ 1 ]

return 3 \* [ 2 ]

return 4 \* [ 6 ]

return 5 \* [ 24 ]

return 120

**Berechnung**

$5 - 1 = 4$

$4 - 1 = 3$

$3 - 1 = 2$

$2 - 1 = 1$

$2 * 1 = 2$

$3 * 2 = 6$

$4 * 6 = 24$

$5 * 24 = 120$

**Beschreibung**

Funktionsaufruf in Main und der Wert 5 wird als Variable n übergeben

False, heisst wird nicht ausgeführt

Resultat wird beim neuen Aufruf Funktion fakultaet mit dem return als Variabel n übergeben

False, heisst wird nicht ausgeführt

Resultat wird beim neuen Aufruf Funktion fakultaet mit dem return als Variabel n übergeben

False, heisst wird nicht ausgeführt

Resultat wird beim neuen Aufruf Funktion fakultaet mit dem return als Variabel n übergeben

False, heisst wird nicht ausgeführt

Resultat wird beim neuen Aufruf Funktion fakultaet mit dem return als Variabel n übergeben

True, heisst es wird der Wert 1 übergeben

Wert vorhergehenden Aufruf zurückgeben

Wert vorhergehenden Aufruf zurückgeben

Wert vorhergehenden Aufruf zurückgeben

Wert vorhergehenden Aufruf zurückgeben

Wert an Main zurückgeben

## 7. Rule of three

Heisst in ++C, dass wenn in einer Klasse eine der drei folgenden Definitionen gemacht wurde, immer alle drei Definitionen gemacht werden sollen.

Destruktor                    (Destructor)  
Kopierkonstruktor       (Copy constructor)  
Zuweisungsoperator   (Assignment operator)

### 7.1. Klassendeklaration

```
class Stack{  
    public:  
        Stack(int maxSize);           //Konstruktor  
        Stack(const Stack &robj);      //Kopierkonstruktor  
        Stack operator = (const Stack &robj); //Zuweisungsoperator-Copy assignment  
        ~Stack();                     //Destruktor  
    private:  
        int index;  
        int maxSize;  
        int *values;  
};
```

Wichtig: Pointer „\*values“ muss immer privat deklariert werden damit nicht als siamesischer Zwilling von einem anderen Objekt darauf zugegriffen werden kann.

### 7.2. Kopierkonstruktor

Macht eine „genaue“ Kopie vom Objekt mit dem Unterschied, dass dem neuen Objekt mit „new“ einen eigenen Speicherplatz zugewiesen wird.

Methode:

```
Stack::Stack(const Stack &rObj){  
    this -> index = rObj.index;  
    maxSize = rObj.maxSize;  
    values = new int [maxSize];           // Wichtig, sonst siamesischer Zwilling  
    for(int i = 0; i < index; i++){  
        values[i]= rObj.values[i];  
    }  
}
```

Main:

```
Stack newStack2(newStack1);           // Aufruf Kopierkonstruktor
```

Adresse	Zuordnung	Variable
0x000		
0x001	newStack1	index
0x002	newStack1	maxSize
0x003	newStack1	Values[ 0 ]
0x004	newStack1	Values[ 1 ]
0x005	newStack1	Values[ 2 ]
0x006		
0x007	newStack2	index
0x008	newStack2	maxSize
0x009	newStack2	Values[ 0 ]
0x00A	newStack2	Values[ 1 ]
0x00B	newStack2	Values[ 2 ]
0x00C		
0x...		

Beschreibung
Automatisch assoziierter Speicher weil Klassenvariable
Automatisch assoziierter Speicher weil Klassenvariable
mit „new“ assoziierter Speicher
mit „new“ assoziierter Speicher
mit „new“ assoziierter Speicher
Automatisch kopiert weil eine Klassenvariable
Automatisch kopiert weil eine Klassenvariable
mit „new“ assoziierter Speicher
mit „new“ assoziierter Speicher
mit „new“ assoziierter Speicher



Beispiel Siamesischer Zwilling:  
(ohne das mit „new“ Speicher assoziierte wird)

Adresse	Zuordnung	Variable
0x000		
0x001	newStack1	index
0x002	newStack1	maxSize
0x003	newStack1 & 2	Values[ 0 ]
0x004	newStack1 & 2	Values[ 1 ]
0x005	newStack1 & 2	Values[ 2 ]
0x006		
0x007	newStack2	index
0x008	newStack2	maxSize
0x009		
0x...		

Beschreibung
Automatisch assoziierter Speicher weil Klassenvariable
Automatisch assoziierter Speicher weil Klassenvariable
Doppelt benutzter Speicher, siamesischer Zwilling!
Doppelt benutzter Speicher, siamesischer Zwilling!
Doppelt benutzter Speicher, siamesischer Zwilling!
Automatisch kopiert weil eine Klassenvariable
Automatisch kopiert weil eine Klassenvariable

### 7.3. Zuweisungsoperator

Funktioniert auf die gleiche Weise wie der Kopierkonstruktor nur mit der Möglichkeit dass der assoziierte Speicher eine andere Grösse haben kann.

Methode:

```
Stack Stack:: operator =(const Stack &rObj) {
    delete[] values; // Nicht notwendig wenn der assoziierte Speicher gleich gross ist.
    maxSize = rObj.maxSize;
    values = new int [rObj.maxSize + 1]; // Wichtig, sonst siamesischer Zwilling, neues Array + 1 grösser als Vorlage!
    for(int i = 0; i < index; i++){ //
        values[i]= rObj.values[i];
    }
    return *this;
}
```

Zum Anwendung des Zuweisungsoperators muss zuerst ein neues Objekt erzeugt werden und danach wird ein Objekt zugewiesen und somit besteht die Möglichkeit alle Werte aus dem zugewiesenen Objekt zu kopieren. Die ganzen Adressen der Variablen werden am Ende dann durch den „This“ Zeiger an das Main zurückgegeben.

Main:

```
Stack newStack3(5); // neues Objekt erzeugen
newStack3 = newStack2; // Zuweisungsoperator ausführen
```

Wichtig: „return \*this“ heisst, dass um das Objekt zurückzugeben muss dazu zuerst eine Kopie erstellt (= Aufruf Kopierkonstruktor) werden und auch gelöscht (= Aufruf Destruktor) werden. Heisst an diesem Punkt wird Kopierkon- und Destruktor immer je einmal aufgerufen.

Adresse	Zuordnung	Variable
0x000		
0x001	newStack1	index
0x002	newStack1	maxSize
0x003	newStack1	Values[ 0 ]
0x004	newStack1	Values[ 1 ]
0x005	newStack1	Values[ 2 ]
0x006		
0x007	newStack2	index
0x008	newStack2	maxSize
0x009	newStack2	Values[ 0 ]
0x00A	newStack2	Values[ 1 ]
0x00B	newStack2	Values[ 2 ]
0x00C		
0x00D	newStack3	index

Beschreibung
Automatisch assoziierter Speicher weil Klassenvariable
Automatisch assoziierter Speicher weil Klassenvariable
mit „new“ assoziierter Speicher bei Aufruf Konst.
mit „new“ assoziierter Speicher bei Aufruf Konst.
mit „new“ assoziierter Speicher bei Aufruf Konst.
Automatisch assoziierter Speicher weil Klassenvariable
Automatisch assoziierter Speicher weil Klassenvariable
mit „new“ assoziierter Speicher bei Aufruf Kopierkonst.
mit „new“ assoziierter Speicher bei Aufruf Kopierkonst.
mit „new“ assoziierter Speicher bei Aufruf Kopierkonst.
Automatisch assoziierter Speicher weil Klassenvariable

0x00E	newStack3	maxSize
0x00F	newStack3	Values[ 0 ]
0x010	newStack3	Values[ 1 ]
0x011	newStack3	Values[ 2 ]
0x012	newStack3	Values[ 3 ]
0x013		
0x...		

Automatisch assoziierter Speicher weil Klassenvariable mit „new“ assoziierter Speicher bei Aufruf Zuw.operator
mit „new“ assoziierter Speicher bei Aufruf Zuw.operator
mit „new“ assoziierter Speicher bei Aufruf Zuw.operator
mit „new“ assoziierter Speicher bei Aufruf Zuw.operator

## 8. Vererbung mit Polymorphie

*(Inheritance with Polymorphism)*

Eine virtuelle Methode ist in der objektorientierten Programmierung eine Methode einer Klasse, deren Einsprung Adresse erst zur Laufzeit ermittelt wird. Dieses sogenannte dynamische Binden ermöglicht es, Klassen von einer Oberklasse abzuleiten und dabei Funktionen zu überschreiben bzw. zu überladen. Das Konzept der virtuellen Methoden wird von einem Compiler (Übersetzer) zum Beispiel mittels virtueller Tabellen umgesetzt.

**Polymorphie** = virtual  
**Abstrakte Klasse** = virtual .... = 0;

Polymorphie bedeutet Vielgestaltigkeit, das heisst es geht um das überladen bei der Vererbung. Es gelten dieselben Regeln wie bei der normalen Vererbung. Bei der Vererbung mit Überladen müssen die entsprechenden Methoden mit der Syntax „virtual“ gekennzeichnet werden. Dadurch weiss die IDE beim Compilieren dass die Einsprung Adresse erst während der Laufzeit ermittelt werden soll.

WICHTIG: Syntax „virtual“ heisst im Prinzip nur, dass alle abgeleiteten Klassen diese Methode implementiert haben müssen. Ohne „virtual“ können sie implementiert sein, müssen aber nicht.

Abstrakte Klassen sind Klassen:

- die nur aus einer Deklaration bestehen können.
- von denen niemals Objekte erstellt werden. Keine Instantiierung möglich.
- die oft als sog. Schnittstellen in umfangreichen Anwendungen verwendet werden.
- die oft als Startpunkt(e) einer Vererbungshierarchie gedacht sind.

Wird die Methode nur als Vorlage für untergeordnete (base) Klassen verwendet, so muss diese zusätzlich noch mit dem Syntax „ = 0;“ gekennzeichnet werden. Diese abstrakten Methoden können nur in den untergeordneten Klassen benutzt werden weil keine Objekt von der abstrakten Methode angelegt werden kann. Die Klasse mit mindestens einer Methode „virtual .... = 0;“ wird dann automatisch zu einer Abstrakten Klasse.

Nutzen Polymorphie:

- Auf diese Art kann dieselbe Methode mehrfach verwendet werden, heisst bei Anpassungen im Code dieser Methode muss diese nur einmal angepasst werden.

Nutzen abstrakte Methoden / Klassen:

- Auf diese Art können mehreren Unterklassen (base) dieselbe Methoden und Variablen zur Verfügung gestellt werden und nur die Abstrakten Methoden müssen entsprechend den Bedürfnissen angepasst werden.

Beispiel Körper, abgeleitet ist der Quader und davon ist der Würfel abgeleitet

Header:

```
// ULTIMATE BASE - SUPERKLASSE KOERBER
class Koerber{
    public:
        Koerber();
        virtual ~Koerber();
```

```

        virtual double getVolumen() = 0;
    };

// BASE 1 - SUBKLASSE QUADER, CHILD VON SUPERKLASSE KOERBER
class Quader : public Koerber{
public:
    Quader(int a, int b, int c);
    virtual ~Quader();
    virtual double getVolumen();

private:
    int a, b, c;
};

//BASE 2 - SUBKLASSE WUEREL, CHILD VON SUBKLASSE QUADER
class Wuerfel : public Quader{
public:
    Wuerfel(int a);
    ~Wuerfel();
};

```

Methoden:

```

ULTIMATE BASE - SUPERKLASSE KOERBER
Koerber::Koerber(){
}

Koerber::~Koerber(){
}

BASE 1 - SUBKLASSE QUADER, CHILD VON SUPERKLASSE KOERBER
Quader::Quader(int a, int b, int c): Koerber(){
    this -> a = a;
    this -> b = b;
    this -> c = c;
}

Quader::~~Quader(){
}

double Quader::getVolumen(){
    double V = a*b*c;
    return V;
}

BASE 2 - SUBKLASSE WUEREL, CHILD VON SUBKLASSE QUADER
Wuerfel::Wuerfel(int a):Quader(a, a, a){
}
Wuerfel::~~Wuerfel(){
}

int main(int argc, char** argv) {

    double a (3), b (5);

    Quader ObjektQuader(b, b, b);
    cout<<"\nQuader Volumen: " << ObjektQuader.Quader::getVolumen();

    Wuerfel ObjektWuerfel(a);
    cout<<"\nWuerfel Volumen: " << ObjektWuerfel.Wuerfel::getVolumen();

    return 0;
};

```

In diesem Beispiel ist die Klasse „Koerber“ eine abstrakte Klasse. Sobald in einer Klasse mindestens eine Methode mit dem Syntax „virtual ...= 0;“ versehen ist, gilt die komplette Klasse als „abstrakte Klasse“.

Eine Klasse „unterhalb“ der abstrakten Klasse wird als „abgeleitete Klasse“ bezeichnet.

Beim Aufruf der der Methode `getVolumen` mit dem Objekt `Wuerfel` wird automatisch die Methode `getVolumen` der Klasse `Quader` verwendet. Dies sieht man auch am Konstruktor vom `Wuerfel`:

`Wuerfel::Wuerfel(int a):Quader(a, a, a)`

`Wuerfel::Wuerfel (int a)` = Konstruktor vom der Klasse „`Wuerfel`“  
`: Quader` = wird abgeleitet von der Klasse „`Quader`“  
`(a, a, a)` ) = das `a` welches mit dem Konstruktor beim Erstellen im Main übergeben wird, wird der Klasse `Quader` als `a, b, c` übergeben

Wichtig:

- Ein Konstruktor kann NIE `virtual` oder `abstrakt` ( = 0 ) gesetzt werden
- Wenn in eine untergeordnete (base / child) Klasse „`abstrakt`“ ist, so wird automatisch auch die übergeordnete (ultimate base / Superklasse) Klasse „`abstrakt`“
- Wenn mit Syntax „`this ->`“ gearbeitet wird sind privat deklarierte Variablen besser geschützt als wenn man mit einer Initialisierungsliste arbeitet. Grund ist, dass man mit der Initialisierungsliste vor Eintritt in den Schleifenrumpf eben auch die privat und kontant deklarierten Variablen initialisieren kann!

## 9. Sortieren

### 9.1. Bubble Sort

Dabei wird das vollständige Array durchlaufen, und jedes Mal – wenn notwendig – werden die benachbarten Elemente miteinander vertauscht

Regel: if (  $n > n + 1$  ) dann austauschen (swap) der Zahl  $n$  und  $n + 1$  (Sortieren von klein-links zu gross- rechts)

Der 1.te Durchlauf erfolgt nach folgendem Prinzip:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	
	17	62	30	73	49	2	12	28	29	0	55	26	
0	17	62											
1	17	62	30	Swap weil $n > n + 1 = \text{True}$									
	17	30	62										
2	17	30	62	73									
3	17	30	62	73	49	Swap weil $n > n + 1 = \text{True}$							
	17	30	62	49	73								
4	17	30	62	49	73	2	Swap weil $n > n + 1 = \text{True}$						
	17	30	62	49	2	73							
5	17	30	62	49	2	73	12	Swap weil $n > n + 1 = \text{True}$					
	17	30	62	49	2	12	73						
6	17	30	62	49	2	12	73	28	Swap weil $n > n + 1 = \text{True}$				
	17	30	62	49	2	12	28	73					
7	17	30	62	49	2	12	28	73	29	Swap weil $n > n + 1 = \text{True}$			
	17	30	62	49	2	12	28	29	73				
8	17	30	62	49	2	12	28	29	73	0	Swap ..		
	17	30	62	49	2	12	28	29	0	73			
9	17	30	62	49	2	12	28	29	0	73	55	Swap ..	
	17	30	62	49	2	12	28	29	0	55	73		
10	17	30	62	49	2	12	28	29	0	55	73	26	Swap ...
	17	30	62	49	2	12	28	29	0	55	26	73	Fertig

Erkenntnis:

- Höchste Zahl ist neu an letzter Stelle, heisst ich muss im nächsten Schritt nur bis Stellen [10], also  $- 1$  sortieren

Der 2.te Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur Position [10] weil die höchste Zahl bereits an der letzten Stelle ist.

Alle weiteren Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur vorhergehenden Position  $- 1$  weil die höchste Zahl immer an der letzten Stelle ist.

Umsetzung

Mittels zwei for Schleifen:

- Schleife 1 (aussen) für die vertikalen Schritte, „Anzahl Variablen  $- 1$ “ entspricht der Anzahl Schleifenwiederholungen.
- Schleife 2 (innen) für die horizontalen Schritten, „Anzahl Variabel  $- 1$ “ zum Starten und danach mit jedem neuen Schleife 1 Durchgang um  $- 1$  reduzieren weil danach die hinterste die jeweils höchste Zahl des Durchlaufes ist.

Laufzeit  $O(n^2)$  (Quadratische Laufzeit weil 2 Schleifen verschachteln)

Bubble Sort ist ein ineffizientes Verfahren da es immer mindesten zwei Schritte braucht um zwei Zahlen auszutauschen und weil die die Anzahl Variablen  $n - 1$  mal alle Zahlen sortiert werden müssen.

## Code

```
bool flagAbbruch (true);

for (int i1 = 0; i1 < SIZE - 1; i1++){
    if (!flagAbbruch){
        return;
    }
    flagAbbruch = false;

    for (int i2 = 0; i2 < SIZE - i1 - 1; i2++){
        if (values[i2] > values[i2 + 1]){
            swap(values[i2], values[i2 + 1]);
            flagAbbruch = true;
        }
    }
}
```

## 9.2. Selection Sort

### Erklärung

Algorithmus sucht als Erstes das kleinste Element in der Liste, merkt es sich und tauscht es gegen das Element am Anfang aus, sodass sich dann das kleinste Element ganz am Anfang befindet. Als Nächstes wird das zweitkleinste Element in der Liste gesucht und wird gegen das an zweiter Stelle platzierte Element der Liste ausgetauscht usw.

Regel: if (  $n > n_{\text{COUNTER}}$  ) dann austauschen Zahl (Sortieren von klein-links zu gross-rechts)

Der 1.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4
Schritt	Prüfung					Aktion
1.1	kleinste Zahl von Array[0] bis Array[5] suchen und Nummer vom Arrayfeld speichern					Nummer 0 vom Arrayfeld speichern

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4
Schritt	Prüfung					Aktion
1.2	Swap Wert von Array[0] auf Array[0].					Swap

Der 2.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4
Schritt	Prüfung					Aktion
2.1	kleinste Zahl von Array[1] bis Array[5] suchen und Nummer vom Arrayfeld speichern. Es kann mit if / else abgefangen werden das kein Swap gemacht wird wenn $n = n_{\text{COUNTER}}$ ist.					Nummer 5 vom Arrayfeld speichern

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	7	6	8	9
Schritt	Prüfung					Aktion
2.2	Swap Wert von Array[5] auf Array[1]					Swap

Der 3.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	7	6	8	9
Schritt	Prüfung					Aktion
3.1	kleinste Zahl von Array[2] bis Array[5] suchen und Nummer vom Arrayfeld speichern					Nummer 3 vom Arrayfeld speichern

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	6	7	8	9
Schritt	Prüfung					Aktion
3.2	Swap Wert von Array[3] auf Array[2]					Swap

und so weiter... bis man bei den letzten zwei Zahlen angekommen ist! Fertig!

### Umsetzung

Mittels zwei for Schleifen:

- Schleife 1 (aussen) startet links und reduziert immer die Anzahl Positionen um -1 bis auf die Position  $\text{Size} - 1$ .
- Schleife 2 (innen) sucht jeweils den kleinsten Wert in den durchsuchten Positionen

Laufzeit  $O(n^2)$  (Quadratische Laufzeit weil 2 Schleifen verschachteln)

Leicht effizienteres Verfahren um Daten zu sortieren. Es wird jeweils die tiefste Zahl im Array gesucht und dann mit aktuellen Position `n_COUNTER` im Array ausgetauscht. Die Position wird links im Array gestartet und wird nach jedem Swap um + 1 verschoben bis `n_COUNTER = maxArraySize - 1` ist.

Im schlechtesten Fall ist die Liste von gross nach klein sortiert, was bedeutet jede Zahl bis auf die grösste muss verschoben werden.

## Code

```
int temp ( 0 ), min ( 0 );

for ( int i1 = 0; i1 < SIZE - 1; i1++ ) {
    min = i1;
    for ( int i2 = i1+1 ; i2 < SIZE ; i2 ++ ) {
        if ( values[ i2 ] < values[ min ] ) {
            min = i2;
        }
    }
    temp = values[ i1 ];
    values[ i1 ] = values[ min ];
    values[ min ] = temp;
}
```



### 9.3. Insertion Sort

Die einzelnen Elemente werden von vorne nach hinten durchlaufen. Von der aktuellen Position aus wird jedes Element von rechts nach links weitergereicht – und das so lange, bis das bewegte Element größer oder gleich dem Element ist, das an der im Augenblick abgefragten Position liegt.

Sortierverfahren (aus der Sicht for Schleife):

for i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	
	17	62	30	73	49	2	12	28	29	0	55	26	
0	17	Element [0] ist sortiert!											
1	17	62	Ist [0] > [1] = False, no action										
2	17	62	30	Ist [1] > [2] = Trues, suche Position = [1]									
	17	30	62	Insert [1] bis [2]									
3	17	30	62	73	Ist [2] > [3] = False, no action								
4	17	30	62	73	49	Ist [3] > [4] = Trues, suche Position = [2]							
	17	30	49	62	73	Insert [2] bis [4]							
5	17	30	62	49	73	2	Ist [4] > [5] = Trues, suche Position = [0]						
	2	17	30	49	62	73	Insert [0] bis [5]						
6	2	17	30	49	62	12	73	Ist [5] > [6] = Trues, suche Position = [1]					
	2	12	17	30	49	62	73	Insert [1] bis [6]					
7	2	12	17	30	49	62	73	28	Ist [6] > [7] = Trues, suche Position = [3]				
	2	12	17	28	30	49	62	73	Insert [3] bis [7]				
8	2	12	17	28	30	49	62	73	29	Ist [7] > [8] = Trues, ...			
	2	12	17	28	29	30	49	62	73	Insert [4] bis [8]			
9	2	12	17	28	29	30	49	62	73	0	Ist [7] > [8] = Trues, ...		
	0	2	12	17	28	29	30	49	62	73	Insert [0] bis [9]		
10	0	2	12	17	28	29	30	49	62	73	55	Ist [8] > [9] = ...	
	0	2	12	17	28	29	30	49	55	62	73	Insert [8]	
11	0	2	12	17	28	29	30	49	55	62	73	26	Ist [9] > [...]
	0	2	12	17	26	28	29	30	49	55	62	73	Insert [4] b.
<b>Fertig</b>													

Beispiel für einen Insert Zyklus (while Schleife):

[0]	[1]	[2]	[3]	[4]	
17	30	62	73	49	Prüfung: Ist [3] > [4] = Trues, suche richtige Position = [2]
17	30	62	49	73	Swap [3] & [4]
17	30	62	49	73	Swap [2] & [3]
17	30	49	62	73	<b>Fertig</b>

Erkenntnis

Nicht linear, je nachdem wie die Zahlen stehen kann es länger oder weniger lang gehen um einen Datensatz komplett zu sortieren.

Umsetzung

Mittels einer for Schleifen und einer while Schleife:

- Schleife for (aussen) läuft n mal durch die Schleife für jeden Zeile
- Schleife while (innen)
  - Prüfen (letzte Position - 1) > (letzte Position)
  - == False, keine Aktion
  - == True, Suchen [Insert Position]
  - while [Insert Position] < (letzte Position)
    - Swap [letzte Position] & [letzte Position - 1]
    - [letzte Position] --

Laufzeit  $O(n^2)$  (Quadratische Laufzeit weil 2 Schleifen verschachteln)

Jedoch ist die Laufzeit unregelmässig, weil es je nach Zahlenaufbau verschieden viele Schritte braucht um eine Zahl an die richtige Stelle zu bringen.

Im schlechtesten Fall ist die Liste von gross nach klein sortiert, was bedeutet jede Zahl muss immer von der aktuellen Position komplett an den Anfang verschoben werden.

## Code

```
int tmp ( 0 ), i2 ( 0 );
```

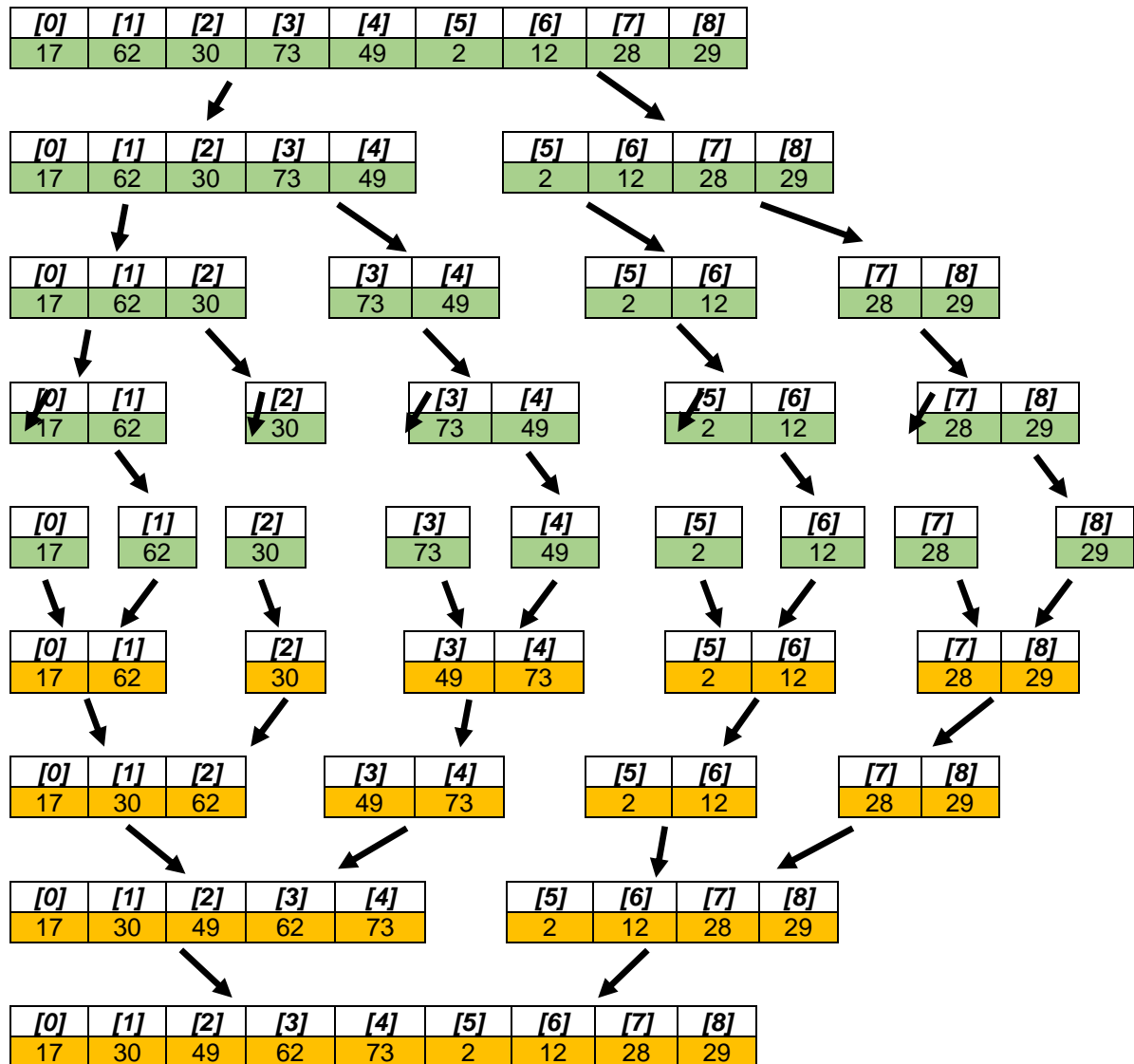
```
for( int i1 =1; i1 < SIZE; i1++ ) {  
    .      i2 = i1;  
    .  
    .      while ( i2 > 0 && values [ i2 - 1 ] > values [ i2 ] ) {  
    .          .      tmp = values [ i2 ];  
    .          .      values [ i2 ] = values [ i2 - 1 ];  
    .          .      values [ i2 - 1 ] = tmp;  
    .          .      i2 --;  
    .      }  
}
```

## 9.4. Merge Sort

Die Daten werden immer in zwei Teile zerlegt (halbiert). Diese zwei Teile werden wiederum jeweils in zwei Teile zerlegt bis nur noch jeweils Zahlenpaare und einzelne Zahlen bestehen. An diesem Punkt wird sortiert und danach Die Rekursion beendet sich, wenn das Teilstück aus nur noch einem Element besteht.

Bei einer Ungeraden Anzahl von Zahlen muss entschieden werden ob jeweils links oder rechts geteilt wird.

Sortierverfahren



Fertig

Erkenntnis

?

Umsetzung

?

Laufzeit  $O(n \cdot \log n)$

?

Code

```

void MergeSort ( int haelfte, int groesse){
.
.   if (groesse < 1 return;
.
. // Array aufteilen
.   int haelfte1 [groesse / 2];
.   int haelfte2 [(groesse + 1) / 2];
.
. // Aufteilen
.   int i;
.   for(i = 0; i < groesse / 2; ++i)
.       haelfte1 [i] = liste [i];
.   for (i = groesse / 2; i < groesse; ++i)
.       haelfte2 [i - groesse / 2] = liste[i];
.
. // ?
.   MergeSort (haelfte1,groesse / 2);
.   MergeSort (haelfte2,( groesse + 1) / 2);
.
.   int *pos1 = &haelfte1 [0];
.   int *pos2 = &haelfte2 [0];
.   for (i = 0; i < groesse; ++i){
.       if (*pos1 <= *pos2){
.           liste [i] = *pos1;
.           if (pos1 != &haelfte2[(groesse+1)/2 - 1]) { // pos1 nicht verändern,
.                                                       // wenn der größte Wert
.                                                       // mehrmals vorkommt
.               if (pos1 == &haelfte1 [groesse / 2 - 1]){
.                   pos1 = &haelfte2 [(groesse + 1) / 2 - 1];
.               }
.               else {
.                   ++pos1;
.               }
.           }
.       }
.       else {
.           liste [i] = *pos2;
.           if (pos2 == &haelfte2 [ (groesse + 1) / 2 - 1] ){
.               pos2 = &haelfte1 [groesse / 2 - 1];
.           }
.           else {
.               ++pos2;
.           }
.       }
.   }
. }
}

```

## 9.5. Quick Sort

Diese Funktion ist in die ANSI-C-Bibliothek mit aufgenommen (qsort). Quicksort funktioniert nach dem Prinzip »Teile und herrsche«, also rekursiv.

Die Daten werden immer in zwei Teile zerlegt und wieder sortiert. Diese zwei Teile werden wiederum jeweils in zwei Teile zerlegt und sortiert usw., bis die Daten sortiert sind. Die Rekursion beendet sich, wenn das Teilstück aus nur noch einem Element besteht.

Erklärung:

Das Pivotelement (von französisch pivot ‚Dreh-/Angelpunkt‘) ist dasjenige Element einer Zahlenmenge, das als Erstes von einem Algorithmus ausgewählt wird, um bestimmte Berechnungen durchzuführen.

In unserem Fall ist das Pivotelement immer in der Mitte zwischen Variabel varMin und varMax, wenn es nicht aufgeht immer rechts von der Mitte.

In der C++ Bibliothek <algorithm> gibt es die Funktion „swap“ die zwei Werte austauscht.

```
#include <algorithm>
swap( intVal1, intVal2 );           // Syntax Normalwerte
swap( Array[intVal1], Array[intVal2] ); // Syntax Arraywerte
```

Array mit den Werten

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
17	62	30	73	49	2	12	28	29

### 1.1 Bestimmen der aktuellen Grösse vom Array

Übergabeparameter bezogen auf das Array beim dem Aufruf der Funktion

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
*sizeArray									
Value	17	62	30	73	49	2	12	28	29
	left								right

Die Werte left = 0, right = 8, \*sizeArray = 0xF001(Adresse Array [0]) werden als Integer beim Aufrufen an die Funktion übergeben.

Der Wert valPiv wird je nachdem ob Gesamtgrösse eine gerade oder ungerade Zahl in der Mitte oder rechts + 1 von der Mitte genommen.

```
Regel:      int indMin = left, indMax = right;

            if ( ( ) % 2 == 0 ) {
                .      indPiv = sizeArray / 2;
            }
            else {
                .      indPiv = sizeArray + 1 / 2;
            }
            valPiv = Array [indPiv];
```

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	62	30	73	49	2	12	28	29
	Left				indPiv				right
	Min								Max

val
49
valPiv

## 1.2 Sortieren

```

Regel:  whileAUSSEN (indMin <= indMax) {
        .   whileMIN (Array[indMin] < Array[indPiv]) Index ++;
        .   whileMAX (Array[indMax] > Array[indPiv]) Index ++;
        .   if indMin <= indMax {
        .       .   swap (Array[indMin], Array[indMax]);
        .       .   indMin ++;
        .       .   indMax --;
        .   }
    }

```

[indMin] <= [indMax] → True, while<sub>AUSSEN</sub> ausführen

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	62	30	73	49	2	12	28	29	49
		Min							Max	Piv

[indMin] (17) < valPiv (49) ? True → Min ++


Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	62	30	73	49	2	12	28	29	49
		Min							Max	Piv

[indMin] (62) < valPiv (49) ? False → Abbruch while<sub>MIN</sub>

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	62	30	73	49	2	12	28	29	49
		Min							Max	Piv

[indMax] (29) > valPiv (49) ? False → Abbruch while<sub>MAX</sub>  
 [indMin] <= [indMax] ? True → swap Array [1] und Array [8]

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	29	30	73	49	2	12	28	62	49
		Min							Max	Piv



Inkrement indMin ++ und indMax --  
 [indMin] <= [indMax] → True, while<sub>AUSSEN</sub> ausführen

Index	[0]	[1]	[2]	[3]	[4]		[6]	[7]	[8]	val
Value	17	29	30	73	49	2	12	28	62	49
			Min						Max	Piv

[indMin] (30) < valPiv (49) ? True → indMin ++

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	29	30	73	49	2	12	28	62	49
				Min					Max	Piv

[indMin] (73) < valPiv (49) ? → Abbruch while<sub>MIN</sub>

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	val
Value	17	29	30	73	49	2	12	28	62	49
				Min					Max	Piv

[indMax] (28) > valPiv (49) ? False → Abbruch while<sub>MAX</sub>  
 [indMin] <= [indMax] ? True → swap Array [3] und Array [7]

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	49	2	12	73	62

Min Max

Inkrement indMin ++ und indMax --

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	49	2	12	73	62

Min Max

[indMin] (28) < valPiv (49) ? True → indMin ++  
[indMin] <= [indMax] → True, while<sub>AUSSEN</sub> ausführen

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	49	2	12	73	62

Min Max

[indMin] (28) < valPiv (49) ? False → Abbruch while<sub>MIN</sub>

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	49	2	12	73	62

Min Max

[indMax] (12) < valPiv (49) ? False → Abbruch while<sub>MAX</sub>  
[indMin] <= [indMax] ? True → swap Array [4] und Array [12]

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	12	2	49	73	62

Min Max

Inkrement indMin ++ und indMax --;  
[indMin] <= [indMax] → True, while<sub>AUSSEN</sub> ausführen

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	12	2	49	73	62

Min  
Max

[indMin] (2) < valPiv (49) True → indMin ++

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	12	2	49	73	62

Max Min

[indMin] (49) < valPiv (49) False → Abbruch while Schleife

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Value	17	29	30	28	12	2	49	73	62

Max Min

[indMax] (2) > valPiv (49) False → Abbruch while Schleife  
[indMin] <= [indMax] ? False → Schleifenrumpf "if" nicht ausführen

[indMin] <= [indMax], False → Abbruch while<sub>AUSSEN</sub>

Erkenntnis:

Alle Wert < 49 sind jetzt auf der linken Seite

Alle Wert  $> 49$  sind jetzt auf der rechten Seite

### 1.3 Rekursionsaufruf

An diesem Punkt wird die Regel ZWEIMAL rekursiv aufgerufen und zwar für einen Linken Array und für einen rechten Array.

Dies wird für ein rechtes Array gemacht solange der Wert left kleiner ist als valMax.

Dies wird für ein linkes Array gemacht solange der Wert valMin kleiner ist als right

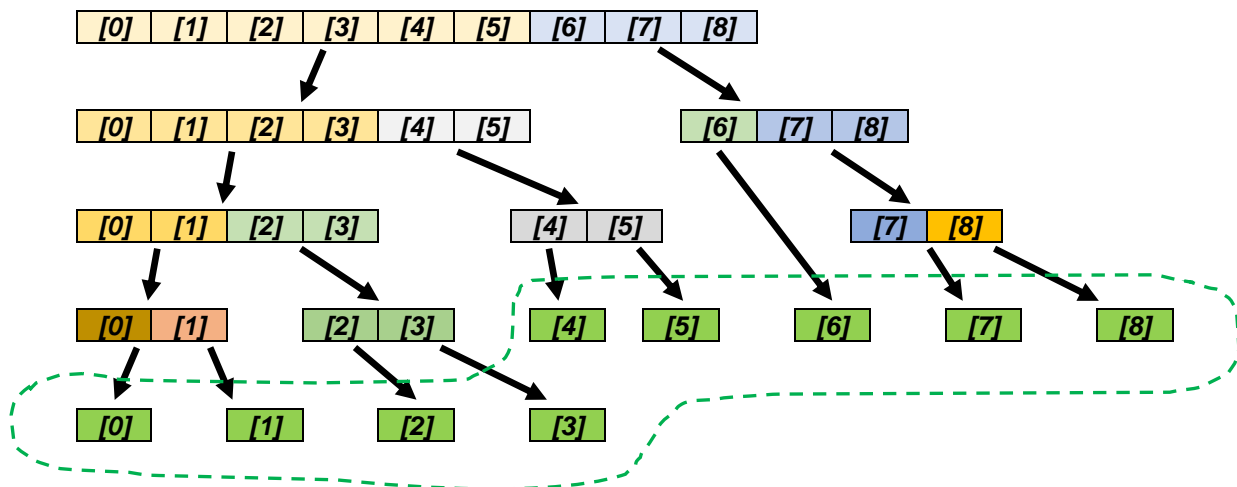
WICHTIG:

Nicht vergessen, die Werte Array [valMin] und Array [valMax] sind (und werden) immer an einem Schnidpunkt übereinander laufen bevor die while<sub>AUSSEN</sub> abgebrochen wird, ausser das Array besteht nur noch aus einem Wert!

<b>Index</b>	<b>[0]</b>	<b>[1]</b>	<b>[2]</b>	<b>[3]</b>	<b>[4]</b>	<b>[5]</b>	<b>[6]</b>	<b>[7]</b>	<b>[8]</b>
<b>Value</b>	17	29	30	28	12	2	49	73	62
	left					Max	Min	right	

```
Regel:    if Array [left] < Array [valMax] { // 0 < 5 → True!
          .   QuickSort (intMin, intMax, sizeArray);
          }

          if valMin < right { // 6 < 8 → True!
            .   QuickSort (intMin, intMax, sizeArray);
            }
        }
```



Laufzeit  $O(n \cdot \log n)$

Schlechtester Fall wäre, wenn genau in umgekehrter Richtung sortiert ist.

Code

```
Void QuickSort (intMin, intMax, sizeArray) {  
    .  
    .      int valMin = left, valMax = right;  
    .  
    .      if ( () % 2 == 0 ) {  
    .          .      indPiv = sizeArray / 2;  
    .      }  
    .      else {  
    .          .      indPiv = sizeArray + 1 / 2;
```



```

.         }
.     valPiv = Array [indPiv];
.
.     while_AUSSEN (valMin <= valMax) {
.         .     while_MIN (Array[valMin] < Array[valPiv]) Index ++;
.         .     while_MAX (Array[valMax] > Array[valPiv]) Index ++;
.         .     if valMin <= valMax {
.         .         .     swap (Array[valMin], Array [valMax]);
.         .         .     valMin ++;
.         .         .     valMax --;
.         .         }
.     }
.
.     If left < valMax {
.         .     QuickSort (intMin, intMax, sizeArray);
.     }
.
.     If calMin < right {
.         .     QuickSort (intMin, intMax, sizeArray);
.     }
. }

```

## 9.6. Randomisierter Quick Sort

Je besser das Array bereits vor dem Anwenden des Quick Sort sortiert ist, desto schlechter ist die Laufzeit. Im Schlechtesten Fall ist das Ganze Array bereits sortiert was der schlechteste Fall wäre. Funktioniert gleich wie der normale Quick Sort, einfach wird der Wert vom Pivot Element durch eine Zufallszahl ersetzt, indem einfach immer die letzte Position vom Array als Pivot Wert genommen wird. Auf diese Weise fällt der schlechteste Fall weg.

Laufzeit  $O(n \cdot \log n)$

## 10. Suchen

### 10.1. Binary Search

Voraussetzung ist, dass das Array bereits sortiert ist!

Laufzeit:  $O(n \log n)$

Das Array wird in der Mitte halbiert. Es gilt dann:

$\text{searchValue} < \text{valueCenter} \rightarrow$  so muss im linken Array Teil weitergesucht werden

$\text{searchValue} > \text{valueCenter} \rightarrow$  so muss im rechten Array Teil weitergesucht werden

$\text{searchValue} == \text{valueCenter} \rightarrow$  Wert gefunden!

- WICHTIG:
- Nicht geeignet wenn man die Position vom Wert vom Wert im Array sucht, da die Position für das Suchen stetig geändert wird.
  - Nicht geeignet wenn ein Wert mehrfach vorhanden ist, der Wert muss Unique sein. Es kann zwar abgefangen werden, aber dann ist die Laufzeit des Algorithmus nicht mehr  $O(n \log n)$

Code

```
int left = startPos;
int right = size;
while ( left <= right ) {
    .   int index = ( ( left + right ) / 2);
    .   if ( data [index] == value ) return index;
    .   if ( data [index] < value ) left = index + 1;
    .   else right = index - 1;
}
return -1;
```

### 10.2. Lineares suchen

Mit der linearen Suche können mehrere Variabel mit demselben Wert gefunden werden.

Funktionsparameter können mit Defaultwerten belgt werden. Parameter, für die ein Defaultwert vorgegeben ist, können beim Aufruf einer Funktion weggelassen werden. In diesem Fall erhält dann der Parameter den Defaultwert.

Code

```
Class Search{
    .   public:
    .   int Linear::SearchStdValue (int value; int *data; int sizeArray; int startPosition = 0);
}
```

// Funktionsaufruf mit Standard Parameter

```
int Linear::Search (int value, int *data; int sizeArray; int startPosition ) {
    .   for ( int i = 0; i < sizeArray; i++ ) {
    .       .   if (data [i] == value){
    .           .       return i;
    .           .   }
    .       }
    .   return -1;
}
```

## Binary Tree

Wird ein Element mit zwei Nachfolger entfernt, so kann diverser ersetzt werden:

- Vom rechten Teilbaum das Element ganz links
- Vom linken Teilbaum das Element ganz rechts

Im Normalfall kann in einem binary tree jeder Wert nur 1 x vorkommen. Wenn dies nicht der Fall ist, so muss man entweder ein Array zum einzelnen Wert anlegen, in dem alle Mehrfachwerte gespeichert werden (dynamisch!) oder man macht hinter jedem Wert einen counter der entsprechend inkrementiert oder dekrementiert wird.

Print all values binary tree

1. Preorder  
Regel Wurzel, links, rechts
2. Inorder  
Regel: links, Wurzel, rechts (sortierte Reihenfolge)
3. Postorder  
Regel: links, rechts, Wurzel

## 11. Linked List

Array benutzen wenn:

- man braucht indexierten Zugriff lesen / schreiben (über Iteration, nicht über Funktion)
- man braucht Direktzugriff lesen / schreiben ( cout << bspArray[15]; )
- wenn man weiss wie viele Elemente gespeichert werden müssen, heisst Liste ist statisch und unflexibel (Arrays sind Anspruchsvoll bei Erweiterungen und brauchen viele Ressourcen)
- schnell mittels Iteration schnell durch den Array gehen muss
- wenn Speicherplatz eine Rolle spielt (Array braucht weniger als Linked List)

Linked List benutzen wenn:

- konstante Laufzeit für einfügen und löschen gebraucht wird, heisst es müssen immer wieder neue Elemente eingefügt oder gelöscht werden
- wenn man nicht weiss wie viele Elemente gespeichert werden müssen, heisst Liste ist dynamisch.
- Kein Direktzugriff notwendig ist, heisst schreiben / lesen wird über Funktionen zur Verfügung gestellt.

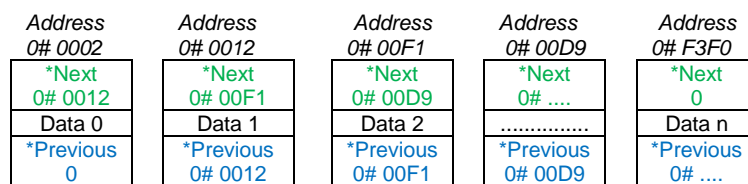
Funktionsbeschreibung:

Ein Element besteht immer mindestens aus folgenden Daten:

- Element, irgend ein Datentyp
- Pointer auf nächstes Element. Das letzte Element hat hier den Eintrag 0

Erweiterung der Linked List:

- Pointer auf vorhergehende Element. Das erste Element hat hier den Eintrag 0



Klasse des Elementes:

Diese Klasse ist das eigentliche Element. Jedes Mal wenn ein neues Element in die Linked List eingefügt wird, wird ein neues Objekt von dieser Klasse erstellt.

```
class Data {  
    private :  
        int value ;  
        Data *next ;  
        Data *previous ;  
  
    public :  
        Data( int value , Data *next = 0, Data *previous = 0 ) ;  
        void setValue( int value ) ;  
        void setNext(Data *next ) ;  
        void setNext(Data *previous ) ;  
        int getValue ( ) const ;  
        Data *getNext ( ) const ;  
        Data *getPrevious ( ) const ;  
};
```

Klasse der Linked List:

Diese Klasse ist die Schnittstelle nach aussen. Hier werden dem Benutzer der linked List die Funktionen zur Verfügung gestellt, welche den Zugriff für das Lesen, das Schreiben, das Einfügen, das Löschen, .. usw. der einzelnen Datenpunkte ermöglicht.

```
class LinkedList {
    private :
        Data *start ;
    public :
        LinkedList ( ) ;
        LinkedList( const LinkedList & obj ) ;
        LinkedList operator= ( const LinkedList &obj ) ;
        ~LinkedList ( ) ;
        // methods to use the l i s t
};
```

Wichtig:

Beim Einfügen und Löschen der Elemente muss man daran denken, dass die jeweiligen Adressen der vorhergehenden und nachfolgenden Elemente zuerst zwischengespeichert werden müssen bevor das entsprechende Objekt „Data“ gelöscht wird.

Wenn dies nicht gemacht wird kann die jeweilige Adresse nicht mehr gefunden werden.

## 12. Containers

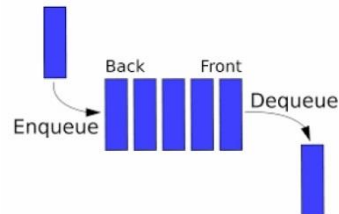
Folgende Container können mit ihren jeweiligen Vor und Nachteilen verwendet werden:

### 12.1. Queue

Der Queue (Schlange, Reihe) ist ein FIFO (first in, first out) Container.

Enqueue = neues Element hinzufügen

Dequeue = Element entnehmen



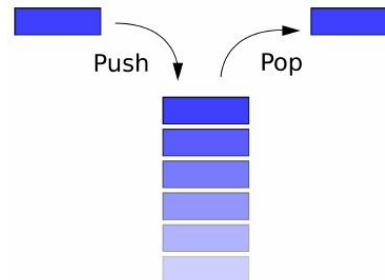
```
class Queue {
    private :
        int *values ; // array based queue
        int maxNumberOfElement ;
        int index ;
    public:
        Queue ( ) ;
        Queue( const Queue & obj ) ;
        Queue operator= ( const Queue & obj ) ;
        ~Queue ( ) ;
        bool deque( int & value ) ;
        bool enqueue( int value ) ;
        int size ( ) ;
        bool isEmpty ( ) ;
};
```

### 12.2. Stack

Der Stack (Stapel) ist ein LIFO (last in, first out) Container.

Push = neues Element hinzufügen

Pop = Element entnehmen

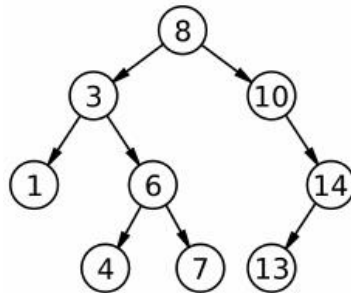


Klasse:

```
class Stack {
    private :
        int *values ; // array based stack
        int maxNumberOfElement ;
        int index ;
    public :
        Stack ( ) ;
        Stack( const Stack & obj ) ;
        Stack operator= ( const Stack & obj ) ;
        ~Stack ( ) ;
        bool pop( int & value ) ;
        bool push( int value ) ;
        bool top( int & value ) ;
        int size ( ) ;
        bool isEmpty ( ) ;
};
```

### 12.3. Binary Tree

Ist eine Baum Daten Struktur in welcher jeder Knoten meistens zwei Nachfolger haben kann. Die Elemente sind unsortiert abgelegt. Jeder Baum hat einen Root (Wurzel) wo der Baum beginnt.



Mögliche Probleme bei Funktionen:

- wenn Werte  $< 0$  sind
- wenn es keine Zahlenwerte sind die gespeichert werden

Klasse TElement:

Diese Klasse ist das eigentliche Element. Jedes Mal wenn ein neues Element in den Binary Tree eingefügt wird, wird ein neues Objekt von dieser Klasse erstellt.

```
class TElement {
    private :
        float data ;
        TElement *left ; //Pointer (Adresse) auf das linke Element
        TElement *right ; //Pointer (Adresse) auf das rechte Element
    public :
        TElement( float data = 0 . 0 ) ;
        ~TElement ( ) ;
        void setLeft(TElement *left ) ;
        void setRight(TElement *right ) ;
        void setData( f l o a t data ) ;
        TElement *getLeft ( ) ;
        TElement *getRight ( ) ;
        float getData ( ) ;
};
```

Klasse Tree:

Diese Klasse ist die Schnittstelle nach aussen. Hier werden dem Benutzer der Binary Tree die Funktionen zur Verfügung gestellt, welche den Zugriff für das Lesen, das Schreiben, das Einfügen, das Löschen, .. usw. der einzelnen Datenpunkte ermöglicht.

```
class Tree {
    private :
        TElement *root ;
    public :
        Tree ( ) ;
        Tree( const Tree & obj ) ;
        Tree operator= ( const Tree & obj ) ;
        ~Tree ( ) ;
        // a d d i t i o n a l methods to use the tree
};
```

Wichtig:

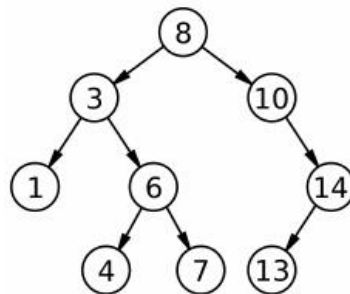
Beim Einfügen und Löschen der Elemente muss man daran denken, dass die jeweiligen Adressen der vorhergehenden und nachfolgenden Elemente zuerst zwischengespeichert werden müssen bevor das entsprechende Objekt „Element“ gelöscht wird.

Wenn dies nicht gemacht wird kann die jeweilige Adresse nicht mehr gefunden werden.

## 12.4. Binary Search Tree

Ist wie der Binary Tree aufgebaut aber er hat eine innere Ordnung. Jeder Knoten enthält einen Wert. Der Wert vom nächst „tieferen“ Knoten:

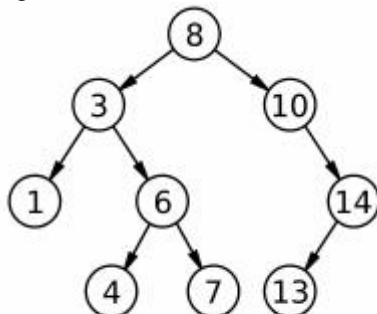
- links ist IMMER  $\leq$  dem vorangehenden Knotenwert
- rechts ist IMMER  $>$  dem vorangehenden Knotenwert



## 12.5. Binary Tree Traversal

Traversierung (durchkreuzen), heisst also es besteht ein genauer systematischer Prozess wie der Baum durchlaufen wird. Für die Traversierung sind drei Formen verfügbar:

- Preorder (root - left - right)  
Start an Wurzel (8) = Ausgeben
  1. gibt es einen weiter Node Links? Ja, gehe zu Node Links = Ausgeben, Nein gehe zu Node rechts = Ausgeben.
  2. solange ein Node ein Child links hat geht man links = Ausgeben. Wenn nicht geht man rechts= Ausgeben
  3. ist man am Ende eines Astes angekommen geht man solange die Nodes zurück, bis man eine Verzweigung nach rechts hat.
  4. gehe zu Punkt 2 zurück

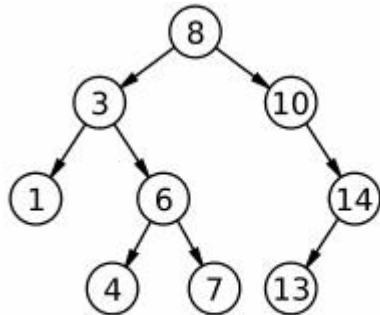


8, 3, 1, 6, 4, 7, 10, 14, 13

- Inorder (left - root - right)  
Start an Wurzel (8)

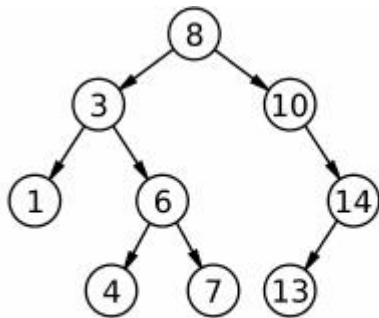


1. gibt es einen weiter Node Links? Ja, gehe solange Links bis Ende erreicht =Ausgeben, Nein gehe zu Node rechts und schaue ob es einen Node links gibt.
2. Gehe einen Node nach oben = Ausgeben und schaue ob es rechts geht. Wenn ja gehe rechts und dann starte wieder bei 1.



1, 3, 4, 6, 7, 8, 10, 13, 14

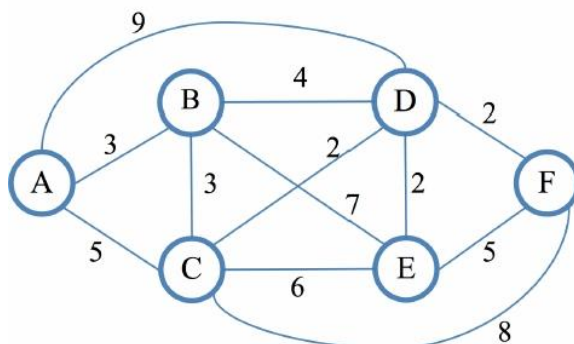
- Postorder (left - right - root)  
Start an Wurzel (8)
1. links ans Ende gehen = Ausgeben und dann zurück bis nächste Abzweigung rechtes
  2. wenn nicht nach links möglich gehe rechtes. Wenn rechts Ende erreicht, = Ausgeben.
  3. wenn Links und rechts eines Node besucht Node Ausgeben.



1, 4, 7, 6, 3, 13, 14, 10, 8

## 12.6. Graph

Ein Graph ist eine Daten Struktur, die aus vielen Knoten mit einem Wert besteht wobei jeder Knoten mit x-vielen anderen Knoten verbunden sein kann.



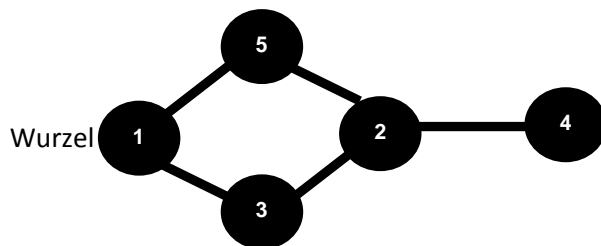
Folgende Probleme bestehen:

- wie wird sichergestellt, dass ich den Knoten nicht mehr als einmal besuche
- was ist der kürzeste Weg
- mehr als zwei Wege können pro Knoten möglich sein

In Computertechnologien sind Graphen ein wichtiger Bestandteil. Typischerweise anspruchsvolle Operation ist das Finden eines Pfades zwischen zwei Konten. Bekannte Algorithmen sind um die kürzesten Wege zu einem Zielknoten zu finden sind:

- depth-first search (Tiefensuche, basierend auf Queue)
- breadth-first search (Breitensuche, basierend auf Stack)

### 12.7. Graph ohne Gewichtung



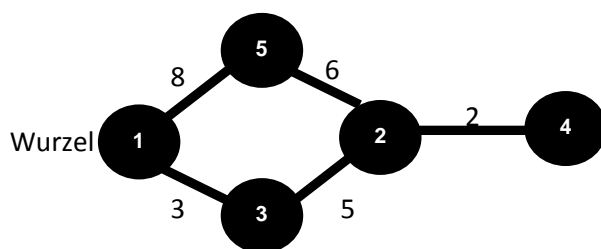
Beispiel Matrix:

	Verbindung zu 1	Verbindung zu 2	Verbindung zu 3	Verbindung zu 4	Verbindung zu 5
Knoten 1			1		1
Knoten 2			1	1	1
Knoten 3	1	1			
Knoten 4		1			
Knoten 5	1		1		

jede 1 in der Matrix entspricht einer Verbindung!

### 12.8. Graph mit Gewichtung

Jede Verbindung besitzt ebenfalls einen Wert



Beispiel Matrix:

	Verbindung zu 1	Verbindung zu 2	Verbindung zu 3	Verbindung zu 4	Verbindung zu 5
Knoten 1			3		8

Knoten 2			5	2	6
Knoten 3	3	5			
Knoten 4		2			
Knoten 5	8		6		

jede Zahl in der Matrix entspricht der Gewichtung einer Verbindung!

Klasse Graph:

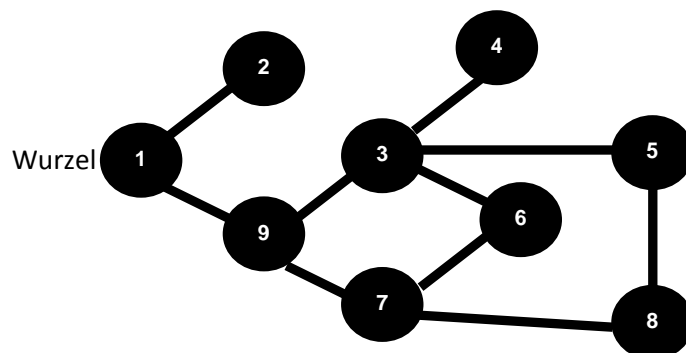
```

class Graph {
    private :
        int **matrix; //Doppelpointer um Matrix zu erstellen
        int value;
    public :
        Graph ( ) ;
        Graph ( const Tree & obj ) ;
        Graph operator= ( const Tree & obj ) ;
        ~ Graph ( ) ;
        int addNode (int value);
        void setConnection(int nodeID1, int nodeID2, int nodeWheight);
        int numberOfNode();
        int numberOfEdges()
        void bfs(); //Breitensuche
        void dfs(); //Tiefensuche
};

```

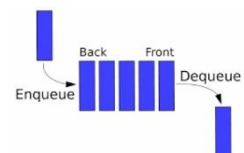
## 12.9. Breath First Search für Graphen

Die Breitensuche beruht auf Queue Prinzip.



Regeln:

- Tiefere Nachbarn zuerst schreiben
- Zahl die im Queue beim Ausgang dequeue steht ist aktueller Node
- Beim Besuch Node den Wert auswerfen mit cout
- Beim Besuch Node wird dieser als „besucht“ markiert



Beispiel:

Schritt	Node IST	Aktion	EN – Queue - DE	Ausgabe
1		Bei Wurzel starten, Enqueue (1)	1	
2	1	Enqueue (2,9) Nachbarn von 1 in Queue schreiben	9,2,1	1,2,9
	1	Dequeue (1) weil keine unbesuchten Nachbarn	9,2	1,2,9
	2	keine unbesuchten Nachbarn, Dequeue (2)	9	1,2,9
	9	Enqueue (3,7) Nachbarn von 9 in Queue schreiben	7,3,9	1,2,9,3,7
	9	9 keine unbesuchten Nachbarn, Dequeue (9)	7,3	1,2,9,3,7
	3	Enqueue(4,5,6) Nachbarn von 3 in Queue schreiben	6,5,4,7,3	1,2,9,3,7,4,5,6
	3	3 keine unbesuchten Nachbarn, Dequeue (3)	6,5,4	1,2,9,3,7,4,5,6
	4	keine unbesuchten Nachbarn, Dequeue (4)	6,5	1,2,9,3,7,4,5,6
	5	Enqueue(8) Nachbarn von 5 in Queue schreiben	8,6,5	1,2,9,3,7,4,5,6,8
	5	5 keine unbesuchten Nachbarn, Dequeue (5)	8,6	1,2,9,3,7,4,5,6,8
	6	6 keine unbesuchten Nachbarn, Dequeue (6)	8	1,2,9,3,7,4,5,6,8

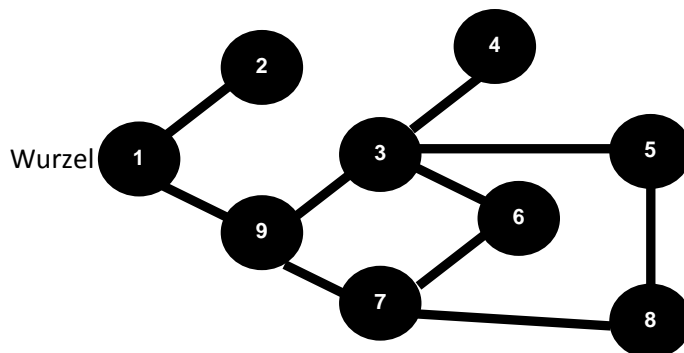
	8	8 keine unbesuchten Nachbarn, Dequeue (8)		1,2,9,3,7,4,5,6,8
		Queue leer		1,2,9,3,7,4,5,6,8
		FERTIG		1,2,9,3,7,4,5,6,8

Pseudo Code Funktion FBS:

```
// Parameter G is the graph
// Parameter n is the starting node
function breath-first-search(G, n) {
    Queue q;
    q.enqueue(n)
    while (q is not empty) {
        u = q.dequeue();
        mark u as discovered;
        for (each node v in G.adjacentNodes(u)) {
            q.enqueue(v);
        }
    }
}
```

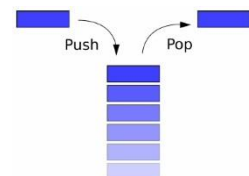
## 12.10. Depth First Search für Graphen

Die Tiefensuche beruht auf Stack Prinzip.



Regeln:

- Immer zuerst zu dem tieferen Node gehen
- Oberster Wert in Stack ist aktueller Node
- Beim Besuch Node den Wert auswerfen mit cout
- Beim Besuch Node wird dieser als „besucht“ markiert



Beispiel:

Schritt	Node IST	Aktion	push-pop -STACK	Ausgabe
1		Bei Wurzel starten, push 1	1	
2	1	Besuchen kleineren Nachbar unbesucht (2), push 2	2,1	1,2
3	2	2 hat keine unbesuchten Nachbarn, pop	1	1,2
4	1	Besuchen kleineren Nachbar unbesucht (9), push 9	9,1	1,2,9
5	9	Besuchen kleineren Nachbar unbesucht (3), push 3	3,9,1	1,2,9,3
6	3	Besuchen kleineren Nachbar unbesucht (4), push 4	4,3,9,1	1,2,9,3,4
7	4	4 hat keine unbesuchten Nachbarn, pop	3,9,1	1,2,9,3,4
8	3	Besuchen kleineren Nachbar unbesucht (5), push 5	5,3,9,1	1,2,9,3,4,5
9	5	Besuchen kleineren Nachbar unbesucht (8), push 8	8,5,3,9,1	1,2,9,3,4,5,8
10	8	Besuchen kleineren Nachbar unbesucht (7), push 7	7,8,5,3,9,1	1,2,9,3,4,5,8,7
11	7	Besuchen kleineren Nachbar unbesucht (6), push 6	6,7,8,5,3,9,1	1,2,9,3,4,5,8,7,6
12	6	6 keine unbesuchten Nachbarn, pop	7,8,5,3,9,1	1,2,9,3,4,5,8,7,6
13	7	7 keine unbesuchten Nachbarn, pop	8,5,3,9,1	1,2,9,3,4,5,8,7,6
14	8	8 keine unbesuchten Nachbarn, pop	5,3,9,1	1,2,9,3,4,5,8,7,6
15	5	5 keine unbesuchten Nachbarn, pop	3,9,1	1,2,9,3,4,5,8,7,6
16	3	3 keine unbesuchten Nachbarn, pop	9,1	1,2,9,3,4,5,8,7,6
17	9	9 keine unbesuchten Nachbarn, pop	1	1,2,9,3,4,5,8,7,6
18	1	1 keine unbesuchten Nachbarn, pop = Stack leer		1,2,9,3,4,5,8,7,6

Pseudo Code Funktion DFS:

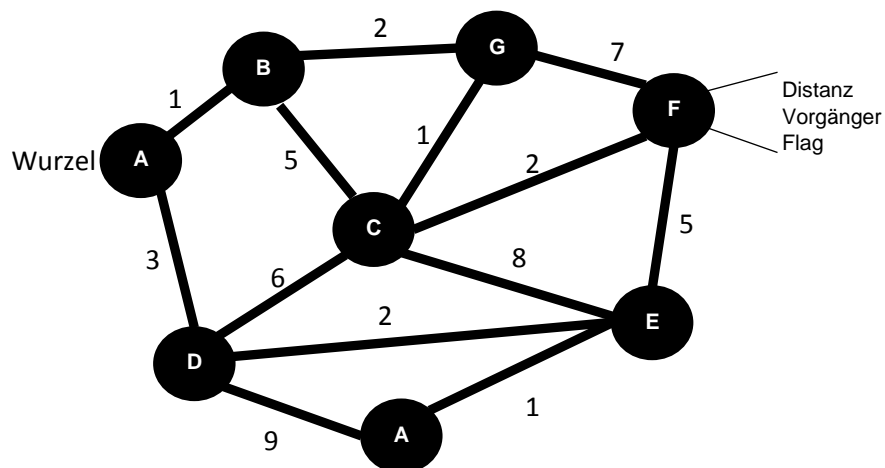
```
// Parameter G is the graph
// Parameter n is the starting node
function depth-first-search(G, n) {
    Stack s;
    s.push(n);
    while (s is not empty) {
        n = s.pop();
        if (n is not discovered) {
            mark n as discovered;
            for (each unvisited node w in G.adjacentNodes(n)) {
                s.push(w);
            }
        }
    }
}
```

### 12.11. Dijkstra's Algorithmus

Mit dem Dijkstra's Algorithmus wird der kürzeste Weg in einem Graph gefunden. Jedem Node werden die zusätzlichen drei Attribute zugeordnet:

- Distanz
- Vorgänger
- Flag „Besucht“

Anhand einer Matrix kann zu jedem Zeitpunkt immer zwischen zwei Nodes die Kosten berechnet werden.



Regeln:

- Alle Nodes sind als „nicht besucht“ markiert
- Alle Attribute auf
  - o Distanz unendlich
  - o Vorgänger 0
  - o Flag 0
- Startpunkt definieren
- Ein besuchter Node wird nicht mehr geändert
- Wenn zwei Node die gleichen Kosten haben wird der erste in der Tabelle genommen
- Matrix mit allen Knoten erstellen

Beispiel:

	Distanz	Vorgänger	Flag
<b>A</b>	$\infty$	0	false
<b>B</b>	$\infty$	0	false
<b>C</b>	$\infty$	0	false
<b>D</b>	$\infty$	0	false
<b>E</b>	$\infty$	0	false
<b>F</b>	$\infty$	0	false
<b>G</b>	$\infty$	0	false
<b>H</b>	$\infty$	0	false

Lösung:

	Distanz	Vorgänger	Flag
<b>A</b>	0	A	1
<b>B</b>	1	A	1
<b>C</b>	4	G	1
<b>D</b>	3	A	1
<b>E</b>	5	D	1
<b>F</b>	6	C	1
<b>G</b>	3	B	1
<b>H</b>	6	E	1

## 13. Templates

Templates erlauben es einen „Platzhalter“ anstelle eines Datentypen einzusetzen und dann während der Laufzeit den Datentypen beim Erstellen des Objekts zu übergeben.

**Wichtig:** Wenn Template verwendet werden wird es kein Code für die Implementierung in das Methoden File geschrieben sondern alles unterhalb der Klassendefinition im Header File.

### Klassen Beispiel:

```
template <class T>
class Stack{
    private:
        T *values;
        int index;
        int maxSize;
    public:
        Stack(int maxSize);
        Stack(const Stack T & obj);
        Stack operator= (const Stack & obj);
        virtual ~Stack();
        bool push(T value);
        bool pop(T &value);
        bool top (T &value);
        int size();
        bool isEmpty();
        void printValues();
        T getRandomValue();
};

template <class T>
Stack<T>::Stack(int maxSize) {
    ...}

template <class T>
Stack<T>::Stack(const Stack T & obj){
    ...}

template <class T>
Stack<T>& Stack<T> ::operator =(const Stack<T> & obj){
    ...}

template <class T>
Stack<T>::~~Stack(){
    ...}

template <class T>
bool Stack<T>::pop(T &value){
    ...}

template <class T>
bool Stack<T>::top(T &value){
    ...}

template <class T>
int Stack<T>::size(){
    ...}

template <class T>
bool Stack<T>::isEmpty(){
    ...}

template <class T>
void Stack<T>::printValues(){
    ...}

template <class T>
T Stack<T>::getRandomValue(){
    ...}
```

### Main Beispiel:

```
//Deklarieren und Initialisieren Variablen
int sizeReturn (-1);
int intSize (10), intPopReturn (0), intTopReturn (0);
long longSize (15), longPopReturn (0), longTopReturn (0);
short shortSize (15), shortPopReturn (0), shortTopReturn (0);

//Erzeugen neue Objekte
Stack<int> intStack1(intSize);
Stack<long> longStack1(longSize);
Stack<int> intStack2(intStack1);
Stack<short> shortStack1(shortSize);
```

### Funktion Beispiel schlecht:

```
template <class T>
    T min(T a , T b) {
        i f (a < b) return a ;
        return b ;
    }

class A {
    // some implementation
};

int main( int argc , char __argv) {
    // use function for type A
    A obj1 , obj2 ;
    A result = min<A>(obj1 , obj2 ) ;
    return 0;
}
```

Compilerfehler weil die Klasse A nicht den Operator < überladet!

### Funktion Beispiel gut:

```
template <class T>
    T min(T a , T b) {
        i f (a < b) return a ;
        return b ;
    }

class A {
    // some implementation
public :
    bool operator < ( const A & obj ) ;
};

int main( int argc , char __argv) {
    // use function for type A
    A obj1 , obj2 ;
    A result = min<A>(obj1 , obj2 ) ;
    return 0;
}
```



## 14. Containers in C++ STL Library

(STL → Standard Template Library)

- Container (er. List, container, queue,...)
- Iteration Operatoren
- Algorithmen

Link EN: <http://www.cplusplus.com/reference/stl/>

Link DE: <http://de.cppreference.com/w/cpp/container>

Container Einteilung:

1. Sequenzielle Container (werden in der Reihenfolge abgelegt in der sie gespeichert werden)
2. Adaptive Container (können nicht mit einem Iteration Operator durchlaufen werden, zum Bsp. Stack -> Push / Pop oder Queue -> enqueue, dequeue)
3. Assoziative Container (mit innerer Ordnung, Daten werden bereits beim ablegen sortiert)

Sequenzielle Container:

- Array
- List (hat keinen Zugriff direkt auf Element in der Liste ausser mit Iteration der Bibliothek)
- Vektor (nur bei letzten Element dynamisch)
- Deque (wie Vektor aber am Anfang und am Ende dynamisch)

Adaptive Container:

- Stack
- Queue

Assoziative Container:

- Set (Sortierter Container mit eindeutigen Inhalten)
- Multiset (wie map nur ist es hier möglich gleiche Werte mehrfach zu speichern)
- Map (Sortierter Container mit zwei Werten die eindeutig sind)
- Multimap (wie map, nur ist es hier möglich gleiche Werte mehrfach zu speichern)

### 14.1. Beispiel Container Deque, Vector, List

```
#include <deque>
using namespace std;

int main (int argc, char **argv){

    deque<int> container;
    // Anstelle dequeue kann auch vector oder list stehen!

    // Elemente füllen wie folgt: 8, 6, 4, 2, 0, 1, 3, 5, 7, 9
    for (int i =0; i < 10; i++){
        if (i%2 == 0) container.push_front(i);
        else container.push_back(i);
    }

    // Elemente ausgeben alt, funktioniert für Vektor, deque, array ABER nicht für List
    for (int i =0; i < container.size(); i++){
        cout << container[i] << endl;
    }

    // Elemente ausgeben Iterator (funktioniert für jeden Container in der Bibliothek)
```

```

deque<int>::iterator iter;
for (iter = container.begin(); iter != container.end(); iter++){
    //wenn in der Mitte Zugriff dann einfach „begin() + 10“ einfügen
    //iter => ist ein pointer auf Element
    cout << *iter << endl;
}

// Element am Ende einfügen bei Dequeue, List, Vektor
objekt.push_back(rand ( ) );

// Element am Start einfügen, funktioniert nicht mit Vektor
objekt.push_front(rand ( ) );

//Element an spezifischer Stelle einfügen
list<int > : : iterator iter3 = c3 . begin ( ) ;
objekt.insert(iter3 , rand ( ) );

// Element löschen
vector<int > : : iterator iter2 = c2 . begin ( ) ;
objekt.erase(iter2 ) ;

// For Schleife
deque<int > : : iterator iter1 ;
for (iter1=c1 . begin ( ) ; iter1!=c1 . end ( ) ; iter1++) {
    cout << *_iter1 << endl ;
}

```

## 14.2. Beispiel Set / Map

```

#include <iostream>
#include <map>
#include <set>

Using namespace std;

int main (int argc, char **argv){

    int intsize = 10000000;
    int intErase = 10000000;

    //Grundlagen
    map<string, int> ageTable;

    //Wert einfügen 1
    ageTable["B.Pitt"] = 48;
    ageTable["G.Bush"] = 76;
    ageTable["M.Mislin"] = 38;
    ageTable["D.Erhart"] = 5;
    ageTable["B.Pitt"] = 50; //überschreibt 48 bei B.Pitt

    //Wert einfügen 2
    pair<map<string, int>::iterator, bool> ret;

    ret = ageTable.insert (pair<string, int> ("G.Bush", 76)); //wird nicht eingefügt weil schon vorhanden,
                                                                //Rückgabe 0

    cout << "Return Val: " << ret.second << endl;

    ret = ageTable.insert (pair<string, int> ("B.Obama", 72)); //wird eingefügt weil nicht vorhanden, Rückgabe 1
    cout << "Return Val: " << ret.second << endl;

    //Container durchlaufen
    map<string,int>::iterator iter;
    for(iter=ageTable.begin(); iter != ageTable.end(); iter++){
        cout << iter->first << " , " << iter->second << endl;
    }

    //Andere Beispiele zu Map / Set
    map<int, int> mapTable;
    set<int> setTable;
}

```

```

// Map, Element einfügen
for (int i = 0; i < intsize; i++){
    mapTable.insert (pair<int, int> (i, i));
}

// Set, Element einfügen
for (int i = 0; i < intsize; i++){
    setTable.insert (i);
}

// Set, Element löschen
for(int i = 0; i < intErase ;i++){
    mapTable.erase(i);
}
cout << "\n\n";
return 0;
}

```

## 15. Algorithmen in C++ STL Library

Link EN: <http://www.cplusplus.com/reference/algorithm/?kw=algorithm>

Link DE: <http://de.cppreference.com/w/cpp/algorithm>

Es stehen verschiedene Algorithmen in der Bibliothek die zur Verfügung stehen um damit auf Container zu arbeiten. Es gibt folgende Beispiele:

count(begin, end, gesuchterWert);	//funktioniert nicht bei list weil kein Index besteht sondern ein pointer auf das nächste Element.
find(begin,end);	//gibt eine Fefernz auf das erste Element vom bereich begin end zurück
find_it(begin, end, Funktion);	
usw.	

## 16. String Matching Algorithmen

Ein String Matching Algorithmus besteht immer aus den folgenden zwei Teilen:

- String der geprüft werden soll
- Pattern (Muster, Vorlage) auf den der String durchsucht werden soll

### 16.1. Naives String Matching

Ein String wird auf ein Pattern geprüft indem auf jeder Position der Textes gegen den Pattern verglichen wird solange ein Match auftritt. Bei einem Miss Match wird automatisch der Pattern um eine Position nach rechts verschoben.

H	E	R	E		I	S		A	S		S	I	M	P	L	E		E	X	A	M	P	L	E		
E	X	A	M	P	L	E																			1 Abbruch bei Vergleich 1	
	E	X	A	M	P	L	E																		2 Abbruch bei Vergleich 2	
		E	X	A	M	P	L	E																	1 Abbruch bei Vergleich 1	
			E	X	A	M	P	L	E																2 Abbruch bei Vergleich 2	
				E	X	A	M	P	L	E															1 Abbruch bei Vergleich 1	
					E	X	A	M	P	L	E														1 Abbruch bei Vergleich 1	
						E	X	A	M	P	L	E													1 Abbruch bei Vergleich 1	
							E	X	A	M	P	L	E												1 Abbruch bei Vergleich 1	
								E	X	A	M	P	L	E											1 Abbruch bei Vergleich 1	
									E	X	A	M	P	L	E										1 Abbruch bei Vergleich 1	
										E	X	A	M	P	L	E									1 Abbruch bei Vergleich 1	
											E	X	A	M	P	L	E								1 Abbruch bei Vergleich 1	
												E	X	A	M	P	L	E							1 Abbruch bei Vergleich 1	
													E	X	A	M	P	L	E						1 Abbruch bei Vergleich 1	
														E	X	A	M	P	L	E					1 Abbruch bei Vergleich 1	
															E	X	A	M	P	L	E				1 Abbruch bei Vergleich 1	
																E	X	A	M	P	L	E			1 Abbruch bei Vergleich 1	
																	E	X	A	M	P	L	E		2 Abbruch bei Vergleich 2	
																		E	X	A	M	P	L	E	1 Abbruch bei Vergleich 1	
																			E	X	A	M	P	L	E	7 Match

28

Klasse:

```
class Naive {
public :
    static int naive(const string &Text, const string &Pattern , int startpos=0) ;
};
```

Methode:

```
void Naive::naiv(const string &Text, const string &Pattern, int startpos){
.    int index (0);
.    while(index + pattern.size() < text.size() ){
.        .    bool flag = true;
.        .    for (int i = 0; i < pattern.size; i++){
.        .        .    if (pattern-at( i ) != text.at ( index + i ) {
.        .        .        .    flag = false;
.        .        .        .    index ++;
.        .        .        .    break;
.        .        .    }
.        .    }
.    }
```

```

.      .      If ( flag ) {
.      .      .      index ++;
.      .      }
.      }
}

```

## 16.2. Boyer Moore String Matching

Der Vergleich vom Text und dem Pattern wird für den Vergleich von der rechten Seite her gestartet.

- Ist letztes Zeichen im vom Pattern ein Match? Wenn Miss Match, prüfen ob das Zeichen vom Text im Pattern vorkommt.
  - o Wenn das Zeichen vom Text nicht im Pattern vorkommt, dann wird der komplette Pattern um Pattern.size() nach rechts verschoben.
  - o Wenn das Zeichen vom Text im Pattern vorkommt, verschieben Pattern bis die beiden Zeichen auf der gleichen Position sind.

**B O Y E R M O R E**

H E R E	I S	A S	S I M P L E	E X A M P L E		
E X A M P L	E				1	
		E X A M P L	E		1	
			E X	A M P L E	5	
				E X A M P L	E	1
					E X A M P L E	7
						15

Klasse:

```
class BoyerMoore {
public:
    static int bm( const string & text, const string & pattern , int startpos=0 ) ;
};
```

**Methode:**

```
void BoyerMoore::naiv(const string &Text, const string &Pattern, int startpos){
    ...
}
```

### 16.3. Knuth Morris Pratt (KMP) String Matching

Der KMP baut auf dem Naiven Suchalgorithmus auf mit dem Unterschied dass das Pattern nicht nur um eine Position sondern bei Bedarf um mehrere verschoben wird. Dazu wird vorab der Pattern auf der ganzen Länge auf einen gleichen Präfix == Suffix vom Pattern geprüft und alle Möglichkeiten zwischengespeichert.

**B e i s p i e l**   **P r ä f i x**   /   **S u f f i x**

```

P a t t e r n :  0 1 0 1 1 0 1 0 1 1
                  0
                  0 1
                  0 1 0
                  0 1 0 1
                  0 1 0 1 1
                  0 1 0 1 1 0
                  0 1 0 1 1 0 1
                  0 1 0 1 1 0 1 0
                  0 1 0 1 1 0 1 0 1
                  0 1 0 1 1 0 1 0 1 1

```

Der Vergleich vom Text und dem Pattern wird für den Vergleich von der linken Seite her gestartet.

- Ist auf der Anzahl Zeichen bei denen vom Pattern bei welchen ein Match war **kein** Präfix == Suffix, dann denn Pattern so weit nach rechts schieben, dass der Präfix genau unter dem Suffix sthet.
- Ist auf der Anzahl Zeichen bei denen vom Pattern bei welchen ein Match war **ein** Präfix == Suffix, dann denn Pattern nach rechts auf Mismatch verschieben

**K N U T     M O R I S E     ( K N P )**  
**P r ä f i x     /     S u f f i x**

A B C	A B C D A B	A B C D A B C D A B D E . . . . .	
<b>A B C D</b>	A B D		4
<b>A</b>	B C D A B D		1
<b>A B C D A B D</b>			7
<b>A B</b>	<b>A B</b>	<i>P r ä f - S u f</i>	
<b>A B C</b>	D A B D		3
<b>A</b>	B C D A B D		1
<b>A B C D A B D</b>			7
<b>A B</b>	<b>A B</b>	<i>P r ä f - S u f</i>	
<b>A B C D A B D</b>			7
<b>A</b>	B C . . .		1
	A . u s w		31

Klasse:

```

class KMP {
public :
    static int kmp( const string & text, const string & pattern , int startpos=0) ;
};

```

Methode:

```

void KMP::kmp( const string & text, const string & pattern , int startpos=0){
    ...
}

```