Das Freigeben, die Deallokation, geschieht mit dem Operator **delete**.

Entsprechend den zwei Formen von **new** (mit und ohne **[]**) gibt es diese auch bei **delete**.

Beispiele:

```
Person*   person = new Person;
Person*   group  = new Person[20];
...
delete    person;
delete[]  group;
```

Die Form des **delete**-Operators muss unbedingt mit derjenigen des **new**-Operators übereinstimmen.

Das Freigeben von alloziertem Speicherplatz darf nicht vergessen werden.

Dies ist besonders wichtig, wenn innerhalb einer Schleife Speicher alloziert wird. Der verfügbare Speicherplatz würde sonst mit jedem Durchgang schrumpfen (sog. *memory leak*).

Wenn einem Zeiger ohne vorherige Deallokation ein neuer Wert zugewiesen wird, ist der alte Speicher-bereich nicht mehr zugreifbar (es sei denn die Adresse wurde noch an eine zweite Zeigervariable zugewiesen).

Einige andere Sprachen wie Java oder Lisp erkennen dies und lassen solchen Speicherplatz von einem sog. *garbage collector* wieder „einsammeln".

Wie erhält man *mehrdimensionale* dynamische Arrays?
Mit **new** kann ein solcher erzeugt werden:

```
int (*matrix)[nSpalten] =
                new int[nZeilen][nSpalten];
```

sofern alle Dimensionen **const** sind, ausser der ersten (hier **nZeilen**). Sonst müsste man so vorgehen:

```
        // Deklaration und Allokation
    int** matrix = new int*[nZeilen];
    for (int i = 0; i < nZeilen; i++)
        matrix[i] = new int[nSpalten];
        // Deallokation
    for (int i = 0; i < nZeilen; i++)
        delete[] matrix[i];
    delete[] matrix;
```

Bequemer geht es später mit speziellen C++ Klassen.

Zeiger und Verbund

Bei Verbund-Datentypen spielen die Zeiger ebenfalls eine grosse Rolle. Die Zeiger ermöglichen den Informationsaustausch ohne dass der ganze Verbund kopiert werden muss. Beispiel:

```
Person b = { "Charles Babbage", 26, … };
Person* p = &b;
```

Via den Zeiger **p** hat man nun Zugriff auf alle Komponenten des Verbundes, z.B. auf das Geburtsdatum mit

```
(*p).geburtstag
```

Für die oft gebrauchte Kombination von Dereferenzierung ***** und Auswahloperator **.** existiert der *Auswahloperator* **->** als Abkürzung:

```
p->geburtstag
```


In einem Verbund dürfen als Komponenten auch Zeiger auf den gleichen Verbund vorkommen.

Beispiel: der Verbund **Person** kann um die Komponenten

```
Person* vater;  Person* mutter;
```

erweitert werden. Damit könnte beispielsweise eine Datenstruktur für einen Stammbaum aufgebaut werden.

Dagegen darf der Verbund sich selber nicht "rekursiv" enthalten, eine Komponente **Person ehepartner;** darf also nicht innerhalb des Verbundes **Person** vorkommen.

Zeiger als Parameter

Bei Funktionsparametern werden oft Zeiger verwendet, vor allem in C, wo es keine Variablenparameter gibt.

Alternativ kann man Arrays auch als solche übergeben, beispielsweise:

```
double det(double mat[3][3]);
```

Die Funktion verlangt dann exakt diese Arraylänge.

Bei *mehrdimensionalen* Arrays hat Übergabe als Array den Vorteil, dass dann innerhalb der Funktionsdefinition die Schreibweise mit Klammern benützt werden darf:

mat[i][j] statt ***(mat + 3*i + j)**

Bei *eindimensionalen* Arrays ist die Klammerschreibweise immer erlaubt. Die Funktion

```
char* strcpy(char* dest, const char* src);
```

darf in ihrer Definition also via `dest[i]`auf das i-te Array-Element zugreifen.

In der Parameterliste ist die Schreibweise mit leeren Array-Klammern erlaubt. Man kann diese brauchen um Zeiger für Arrays visuell zu unterscheiden von Zeigern auf einfache Variablen:

```
char* strcpy(char dest[],
             const char src[]);
```


Für Zeiger gilt wie für andere Variablen: Bei der Parameter-Übergabe wird der Wert kopiert.

Bei einem Zeiger bedeutet dies:

- Man kann die Zeiger-Variable selbst nicht verändern.
- Man kann aber via den Zeiger den Speicher verändern.

Wenn der Zeiger für einen Array steht, heisst dies:

- Der Array bleibt an Ort und Stelle (die Anfangsadresse ist unveränderlich).
- Der *Inhalt* des Arrays kann verändert werden.
- Das Schlüsselwort `const` garantiert, dass der Inhalt des Arrays unverändert bleibt (siehe Beispiel `strcpy`).

Zeiger als Parameter sind nicht nur bei Arrays, sondern auch bei einfachen Variablen möglich.
Das Polarkoordinaten-Beispiel sieht mit Zeigern anstelle von Variablenparametern so aus:

```
void polar(double x, double y,
           double* rho, double* phi)
{
     *rho = sqrt(x*x + y*y);
     *phi = atan2(x, y);
}
```

Aufruf:

```
double x = 4., y = 3., laenge, winkel;
polar(x, y, &laenge, &winkel);
cout << "  Laenge:" << laenge <<
        ", Winkel:" << winkel << endl;
```


Mit Zeigern wird explizit ausgeführt, was Variablen-parameter implizit leisten:

- Anwendung des Adressoperator `&` vor dem Aufruf (aus `laenge` wird `&laenge`)
- Übergabe als Zeiger (aus Typ `double` wird `double*`)
- Anwendung des Dereferenzierungsoperators `*` (aus `rho` wird `*rho`).

Zeiger als Rückgabewerte

Funktionen dürfen auch Zeiger zurückgeben. Ein Fehler wäre aber, einen Zeiger auf eine lokale Variable (z.B. Array oder Verbund) zurückzugeben. Der Speicher-platz muss mit `new` dynamisch alloziert werden (oder schon vor dem Aufruf alloziert sein).

```
double* sinusTabelle()
{
   double* s = new double[360];
    // nicht moeglich mit: double s[360];
   for (int i; i < 360; i++)
      s[i] = sin(i*M_PI/180.);
   return s;
}
…
double* sinTab = sinusTabelle();
```


Suchbäume

Beim Verwalten eines *dynamischen Datenbestandes* möchte man in beliebiger Folge Datensätze

- *suchen* (zum Lesen oder Ändern),
- *einfügen*,
- *löschen*.

Wir nehmen an dass, dieser aus *n* gleichartigen Datensätzen besteht. Die einzelnen Datensätze werden sinnvollerweise als Verbund realisiert.

In welcher Datenstruktur sind diese drei Operationen effizient implementierbar?

Naheliegende Lösung: den Datenbestand als *Array* organisieren.

- Einfügen: Zeitaufwand *O(1)* weil der Datensatz am Schluss angehängt werden kann. Aber: Doppeltes Einfügen wird nicht erkannt!
- Suchen, Löschen: Zeitaufwand *O(n)* weil *sequentielles Suchen* nötig.

Nächstbessere Lösung: den Datenbestand als *sortierten Array* zu organisieren. Eine Komponente des Verbundes wird als *Sortierschlüssel* gewählt. Der Schlüssel muss den Datensatz *eindeutig* identifizieren.

- Suchen: Zeitaufwand *O(log n)* weil *binäres Suchen* möglich. Man findet Datensätze nach (aufgerundet) $log_2\, n + 1$ Vergleichen.
- Einfügen, Löschen: Zeitaufwand *O(n)* weil durchschnittlich $n/2$ Array-Elemente verschoben werden müssen.

Ziel: alle drei Operationen in *O(log n)* Zeit realisieren.

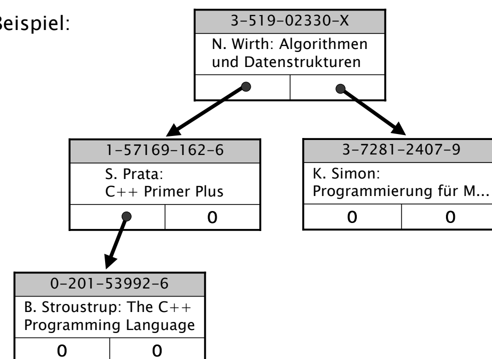
Datenstruktur: *binärer Baum*.

Die Knoten des Baumes sind die Datensätze, erweitert um zwei Zeiger.

```
struct TreeNode {
    int key;          // Sortierschluessel
    Data data;        // weitere Daten
    TreeNode* left;   // linker Teilbaum
    TreeNode* right;  // rechter Teilbaum
};
```

Der Sortierschlüssel kann von einem beliebigen Datentyp sein, der eine *lineare Ordnung* besitzt.

Beispiel:



Jeder (gültige) Zeiger vom Typ `TreeNode*` repräsentiert einen Baum. Im Fall eines `NULL`-Zeigers (Zeigers mit Wert 0) ist dies ein leerer Baum.

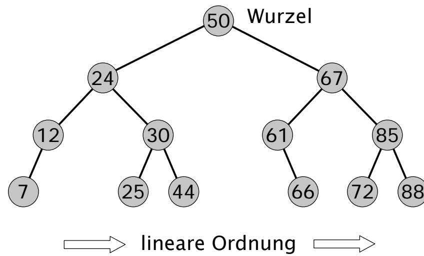
Bedingungen für einen gültigen nichtleeren Baum:
- Die Zeiger `left` und `right` sind entweder 0 oder zeigen auf einen Knoten.
- Keine zwei Zeiger zeigen auf denselben Knoten.
- Es gibt einen Wurzel-Knoten, auf den kein Zeiger zeigt.

Ein *Suchbaum* ist ein binärer Baum, bei dem die Knoten von links nach rechts sortiert sind:

Für jeden Knoten eines Suchbaums gilt:

- Der Schlüssel ist grösser als alle Schlüssel im linken Teilbaum
- Der Schlüssel ist kleiner als alle Schlüssel im rechten Teilbaum

Beispiel eines Suchbaumes:



Suchen

Das Suchen ist die einfachste der drei Operationen.

Die Funktion **search** sucht im Baum **r** einen Datensatz mit Schlüssel **z**. Rückgabewert ist ein Zeiger auf den entsprechenden Knoten, resp. 0 wenn kein solcher existiert.

```
TreeNode* search(int z, TreeNode* r)
{
  if (!r)               return 0; // leerer Baum
  else if (z < r->key)  return search(z, r->left);
  else if (z == r->key) return r;
  else /* z > r->key */ return search(z, r->right);
}
```


Einfügen

Die Funktion **insert** fügt einen neuen Datensatz mit Schlüssel **z** ein, sofern der Schlüssel noch nicht im Baum vorkommt. Via zurückgegebenen Zeiger kann der Datensatz anschliessend noch ergänzt werden.

```
TreeNode* insert(int z, TreeNode* &r)
{
  if (!r) {         // leerer Baum: fuege hier ein
    r = new TreeNode; r->key = z;
    r->left = 0; r->right = 0;
    return r;
  }
  else if (z < r->key)  return insert(z, r->left);
  else if (z == r->key) return 0; // exist. schon
  else /* z > r->key */ return insert(z, r->right);
}
```


Löschen

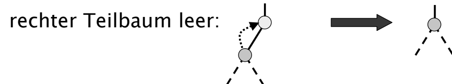
Das Löschen ist die schwierigste der drei Operationen:

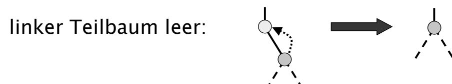
Sie beginnt mit einer Suchoperation. Ergebnis ist dann ein Baum mit dem zu löschenden Knoten in der Wurzel.

Das reduzierte Problem ist nun:

Wie entfernt man die Wurzel ohne die Suchbaum-Eigenschaft zu verletzen?

Dies ist leicht wenn (mindestens) einer der beiden Teilbäume leer ist: Man ersetzt den Baum einfach durch den zweiten Teilbaum.



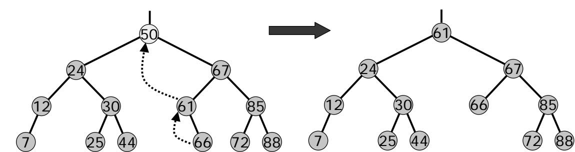


beide Teilbäume leer: (leerer Baum)

Seien nun beide Teilbäume nichtleer.

Die Idee ist nun:

- im rechten Teilbaum den linksäussersten Knoten (den Knoten mit dem kleinsten Schlüssel) entfernen, und
- diesen anstelle der Wurzel verwenden.



Die Hilfsfunktion `detachMin` entfernt im Baum `r` den Knoten mit dem kleinsten Schlüssel (ohne den Datensatz zu löschen). Rückgabewert ist ein Zeiger auf diesen "abgehängten" Datensatz.

```
TreeNode* detachMin(TreeNode* &r)
{
  if (r->left) return detachMin(r->left);
  else {
      TreeNode* p = r;
      r = r->right;
      return p;
  }
}
```

Dies erlaubt nun die Funktion `remove` zu schreiben:


```
void remove(int z, TreeNode* &r)
{
  if (!r) return;  // Fehler: nicht gefunden
  else if (z < r->key) remove(z, r->left);
  else if (z == r->key) {
    TreeNode* left  = r->left;
    TreeNode* right = r->right;
    delete r;
    if (!left) r = right;
    else if (!right) r = left;
    else {
      r = detachMin(right);
      r->right = right; r->left = left;
    }
  }
  else /* z > r->key */ remove(z, r->right);
  return;
}
```


Laufzeitanalyse

Die Laufzeit der Operationen Suchen, Einfügen und Löschen ist proportional zur *Höhe* des Suchbaumes.

Die Höhe eines Suchbaums mit n Knoten ist

- im Idealfall $log_2 (n+1)$ (aufgerundet),
- im schlimmsten Fall n

Der schlimmste Fall tritt ein, wenn ausgehend vom leeren Baum, n Datensätze mit monoton steigenden (oder fallenden) Schlüsseln eingefügt werden.

Der Idealfall liegt vor, wenn der Baum *vollständig ausgeglichen* ist:

Definition: Ein binärer Baum heisst *vollständig ausgeglichen*, wenn in jedem Knoten gilt: Die *Anzahl Knoten* der beiden Teilbäume unterscheidet sich höchstens um Eins.

Eigenschaft der vollständig ausgeglichenen Bäume: Die Länge aller Äste (Wurzel bis Blatt) unterscheidet sich höchstens um Eins.

Problem: Unsere Einfüge- und Löschoperationen zerstören die vollständige Ausgeglichenheit.

Das vollständige Ausgleichen nach *jeder* solchen Operation wäre zu aufwendig.

Eine praktikable Lösung ist dagegen das periodische Neuanlegen des gesamten Suchbaumes.

Besser ist es, eine schwächere Eigenschaft zu verwenden:

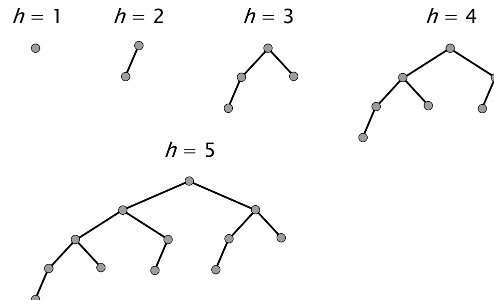
Definition: Ein binärer Baum heisst *ausgeglichen* (oder *AVL-Baum*), wenn in jedem Knoten gilt: Die *Höhe* der beiden Teilbäume unterscheidet sich höchstens um Eins.

Vorteil: diese Eigenschaft ist leichter wiederherzustellen.

Zudem wird Einfügen und Löschen nur unwesentlich aufwendiger, denn:

Adelson-Velskii und Landis haben gezeigt, dass ausgegliche Bäume maximal 45% höher sind als die entsprechenden vollständig ausgeglichenen Bäume.

Beispiele ausgeglichener Bäume: Die Fibonacci-Bäume



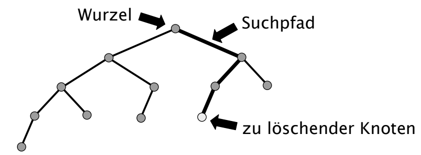
Ausgleichen in AVL-Bäumen

Um die Ausgeglichenheit eines (Teil-)Baumes festzustellen resp. wiederherzustellen, muss man die *Höhe* eines Teilbaumes ermitteln können.

Damit dies genügend schnell geht, wird in jedem Knoten zusätzlich die Höhe seines Teilbaumes gespeichert.

Beim Einfügen oder Löschen ändert sich die Höhe nur für Knoten entlang dem *Suchpfad*. Beim "Abbauen" der Rekursion in `insert` resp. `remove` klettert man aber genau dem Suchpfad nach zurück. Man kann dabei die Höhe wo nötig korrigieren.

Beispiel:



Nach einer einzigen Einfüge- oder Löschoperation ist die Ausgeglichenheit höchstens für Knoten entlang dem Suchpfad verletzt.

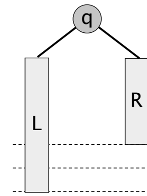
In diesem Fall unterscheiden sich die Höhen der beiden Teilbäume um 2.

Idee: in den Funktionen `insert` und `remove` zusätzlich vor der `return`-Anweisung:

- falls sich die Höhen der beiden Teilbäume `r->left` und `r->right` um 2 unterscheiden, dies durch sog. Rotationen (s.u.) korrigieren,
- die gespeicherte Höhe des Baumes `r` korrigieren.

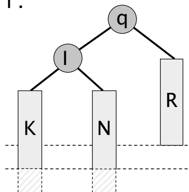
Annahme (oBdA.): der linke Teilbaum ist um 2 höher als der rechte.

In Skizze dargestellt:



Die beiden Rechtecke L und R symbolisieren Teilbäume der gezeichneten Höhe. Dank der Rekursion darf man L und R als ausgeglichen voraussetzen.

Sei nun l die Wurzel von L und seien K und N die beiden Teilbäume von l .



Nun sind die beiden Fälle

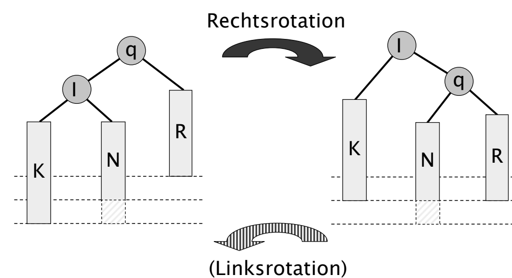
Höhe(K) $\geq$ Höhe(N)

Höhe(K) $<$ Höhe(N)

getrennt zu behandeln.

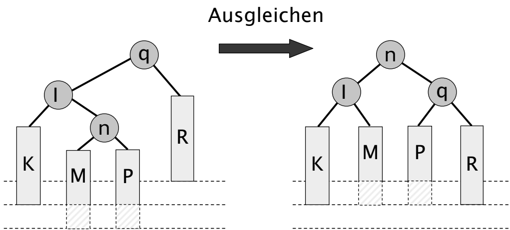
Fall I: Höhe(K) $\geq$ Höhe(N)

Hier besteht das Ausgleichen in einer *Rechtsrotation:*



Fall II: Höhe(K) < Höhe(N)

Hier besteht das Ausgleichen in einer Linksrotation um l gefolgt von einer Rechtsrotation um q.



Die Rotationsoperationen bestehen nur aus dem "Umhängen" von zwei Zeigern sowie dem Austauschen der Wurzel:

```
void rotateRight(TreeNode* &r)
{
  TreeNode* l = r->left;
  r->left = l->right;
  l->right = r;
  r = l;
  return;
}
```

Die Linksrotation funktioniert analog. Beide wären noch zu ergänzen um das Nachführen der gespeicherten Höhe.

Durchlaufstrategien

Wie erhält man eine geordnete Aufzählung aller Schlüssel in einem Suchbaum?

Am einfachsten natürlich mit einer rekursiven Funktion:

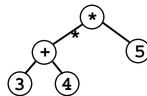
```
void printTree(TreeNode* &r)
{
  if (r) {
    printTree(r->left);
    cout << r->key << endl;
    printTree(r->right);
  }
  return;
}
```

Die Reihenfolge wie die Wurzel und die zwei Teilbäume behandelt werden ist hier wichtig.

Es gibt folgende sinnvolle Strategien, einen (allgemeinen) binären Baum zu traversieren:

- *inorder*-Reihenfolge:
  - linker Teilbaum
  - Wurzel
  - rechter Teilbaum
- *preorder*-Reihenfolge:
  - Wurzel
  - linker Teilbaum
  - rechter Teilbaum
- *postorder*-Reihenfolge:
  - linker Teilbaum
  - rechter Teilbaum
  - Wurzel

Stellt der Baum einen arithmetischen Ausdruck dar wie



dann ergibt

inorder: `(3+4)*5` die gewohnte Infix-Schreibweise (Klammern sind nötig!),

preorder: `*(+(3,4),5)` die Präfix- oder "Funktions"-Schreibweise (Klammern und Kommas sind nicht nötig wenn die Anzahl Operanden klar ist),

postorder: `3 4 + 5 *` die Postfix-Schreibweise, wie z.B. von PostScript und von früheren Taschenrechnern verwendet.

Klassen

Objektorientiertes statt *prozedurales* Programmieren ist eine neuere "Philosophie" der Software-Entwicklung.

Prozedural:

- Das (Unter-) Programm steht im Mittelpunkt.
- Es transformiert Eingabedaten in Ausgabedaten.
- Daten selbst sind "passiv".
- Der Zugriff auf die Daten wird nicht kontrolliert.

Objektorientiert:
- Daten und deren Verwaltung werden zusammengefasst zu *Objekten*.
- Das Objekt bietet kontrollierte Zugriffe auf die Daten an um diese zu lesen oder zu manipulieren.
- Objekte sind Variablen.
- Die Datentypen von solchen Variablen nennt man *Klassen*.

Auch die Mathematik kennt "Datentypen" die aus einer Menge und darauf definierten Operationen bestehen. Beispiele sind Gruppe, Vektorraum, etc.

In C++ sind Klassen als *Erweiterungen des Verbundes* realisiert.

Beispiel "Stack"

Wir wollen einen Datentyp für einen *Stack* (auch: Stapel, Keller) definieren.

Ein Stack
- enthält Datensätze, hier der Einfachheit halber nur **char**-Zeichen, und er
- erlaubt die drei Operationen:
  - **empty** : abfragen ob der Stapel leer ist
  - **push** : einen neuen Datensatz auf den Stack laden
  - **pop** : den obersten Datensatz wegnehmen

Ziel: Ein Datentyp **Stack**, der die Implementation "versteckt".

Der Stack kann z.B. als *Array* oder als *verkettete Liste* implementiert sein.

Die Implementation soll später ersetzbar sein durch eine effizientere, ohne dass alle Anwendungsprogramme geändert werden müssen.

Nach aussen "sichtbar" sein sollen einzig die Spezifikationen (d.h. die Deklarationen) der drei Funktionen **empty**, **push** und **pop**.

Erste Implementation in C++:

```
struct Stack {
    static const int SIZE = 100;
    char data[SIZE];
    char* top;
    void init()          { top = data; }
    bool empty()         { return top == data; }
    void push(char item){ *top++ = item; }
    char pop()           { return *--top; }
};
```


Die Klasse **Stack** hat 7 Elemente, davon 4 *Element-funktionen* (auch "*Methoden*" genannt).

Das Element **SIZE** ist ebenfalls speziell weil als **static** deklariert.

Die verbleibenden Elemente **data** und **top** sind herkömmliche Verbund-Komponenten. Nur diese belegen effektiv Speicherplatz pro Variable vom Typ **Stack** .

Beispiel für Benützung der Klasse:


```
// Testprogramm: Woerter umkehren
int main()
{
  Stack st;
  st.init();
  while (!cin.eof()) {
    char c = cin.get();
    if (isalnum(c)) st.push(c);
    else {
      while (!st.empty()) {
        cout.put(st.pop());
      }
      if (!cin.eof()) cout.put(c);
    }
  }
}
```