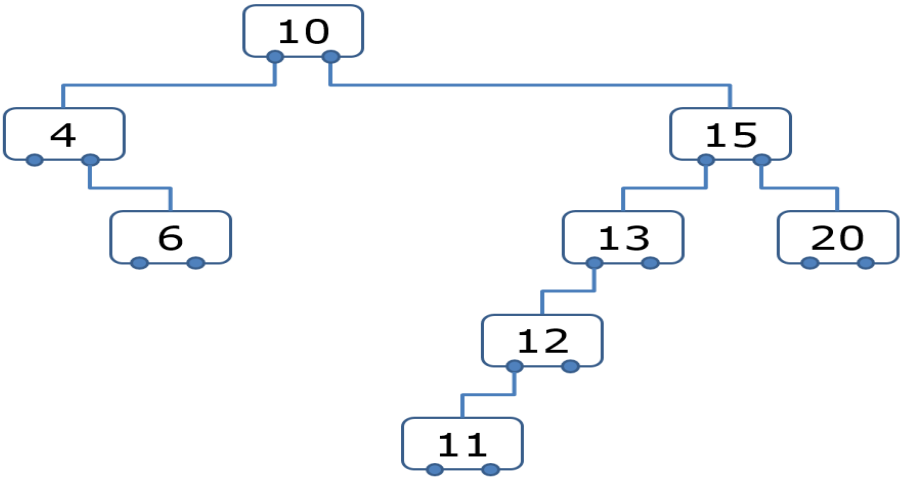
# Praktikumstermin Nr. 09, INF
# Dynamische Datenstrukturen - Binärer Suchbaum

*Abgabe im GIP-INF Praktikum der Woche 11.12.-15.12.2023.*

## (Pflicht-) Aufgabe INF-09.01: Dynamische Datenstruktur: Binärer Suchbaum (duplikatfrei) über `int` Werten

Ein *Binärer Suchbaum (ohne Duplikate) über `int` Werten* ist eine Datenstruktur, in der `int` Werte in den Knoten der Datenstruktur nach den im folgenden beschriebenen Regeln gespeichert werden.




Jeder Knoten der Datenstruktur speichert genau einen `int` Wert und besitzt höchstens zwei *Kindknoten*.

Der *erste* in den Baum einzufügende `int` Wert wird im neu zu erzeugenden Wurzelknoten des Baums abgelegt.

Jeder weitere einzufügende `int` Wert wird nach folgendem Prinzip in den Baum eingefügt: Ausgehend vom Wurzelknoten wird der neue Wert mit dem im jeweiligen Knoten gespeicherten Wert verglichen.

1. Ist der neue Wert *gleich* dem Wert im Knoten, so wird der neue Wert nicht erneut in den Baum eingefügt (*duplikatfreier* Baum).

2. Ist der neue Wert *kleiner* dem Wert im Knoten und besitzt der Knoten *keinen* linken Kindknoten, so wird der neue Wert in einen neu zu erzeugenden linken Kindknoten eingefügt.

3. Ist der neue Wert *kleiner* dem Wert im Knoten und besitzt der Knoten einen linken Kindknoten, so wird die Prüfung ab Fall 1. für den linken Kindknoten erneut vorgenommen.

   *Fälle 4. und 5. sind analog zu 2. und 3.:*

4. Ist der neue Wert *größer* dem Wert im Knoten und besitzt der Knoten *keinen* rechten Kindknoten, so wird der neue Wert in einen neu zu erzeugenden rechten Kindknoten eingefügt.

5. Ist der neue Wert *größer* dem Wert im Knoten und besitzt der Knoten einen rechten Kindknoten, so wird die Prüfung ab Fall 1. für den rechten Kindknoten erneut vorgenommen.

Programmieren Sie (***noch keine Klassen nutzen / definieren!***) eine `struct` Datenstruktur `BaumKnoten` mit einem Attribut `int data` sowie zwei Attributen `BaumKnoten* links` und `BaumKnoten* rechts`. Programmieren Sie ferner zwei Funktionen `einfuegen()` und `ausgeben()`, um einen duplikatfreien Binärbaum über `int` Werten gemäß den Testläufen zu realisieren. Die Funktionen sollen den `anker`, d.h. einen Pointer auf den Wurzelknoten des Baums, als Parameter nehmen. Die Funktion `einfuegen()` soll außerdem einen `int` Wert `wert` als zweiten Parameter nehmen, der dann in den Baum eingefügt werden soll (falls noch nicht dort vorhanden).

Verwenden Sie keine globalen oder `static` Variablen.

Deklarieren Sie die `struct` Datenstruktur `BaumKnoten` sowie die Funktionsprototypen für `einfuegen()` und `ausgeben()` in einer Headerdatei `binaerer_suchbaum.h` und innerhalb eines Namespaces `suchbaum`.

Implementieren Sie die Funktionen `einfuegen()` und `ausgeben()` in einer Datei `binaerer_suchbaum.cpp`. Sie können gerne zusätzliche Hilfsfunktionen definieren, dann aber innerhalb des Namespaces `suchbaum`.
Die Ausgabefunktion `ausgeben()` rückt die Knotenwerte entsprechend

ihrer Tiefe im Baum (d.h. Abstand vom Wurzelknoten) ein, mit zwei Pluszeichen pro Tiefenstufe. Der Baum ist bei der textuellen Ausgabe „um 90 Grad gegen den Uhrzeigersinn gedreht" im Vergleich zur Diagrammdarstellung.
D.h. zu einem Baumknoten wird erst der rechte Teilbaum ausgegeben, dann der Wert des Knotens selbst, dann der linke Teilbaum.
Wegen der Selbstähnlichkeit (Teilbaum sieht von der Struktur aus wie der gesamte Baum): Realisieren Sie die Ausgabe über eine rekursive Funktion

```
void suchbaum::knoten_ausgeben(BaumKnoten* knoten,
                               unsigned int tiefe);
```

… die aus der Funktion `ausgeben()` aufgerufen wird und genau das obige Ausgabeprinzip umsetzt (lassen Sie sich von der „Türme von Hanoi" Funktion inspirieren, falls nötig…).

Legen Sie im Projekt eine leere Headerdatei `gip_mini_catch.h` an und kopieren Sie den Inhalt der in Ilias gegebenen gleichnamigen Datei in diese Headerdatei.

Legen Sie im Projekt eine Datei `test_binaerer_suchbaum.cpp` an und kopieren Sie den Inhalt der in Ilias gegebenen gleichnamigen Datei in diese Datei.

Legen Sie im Projekt eine Datei `suchbaum_main.cpp` an und kopieren Sie den Inhalt der in Ilias gegebenen gleichnamigen Datei in diese Datei.

## Testläufe (Benutzereingaben sind unterstrichen):

```
Alle Tests erfolgreich (71 REQUIREs in 7 Test Cases)
Leerer Baum.
Naechster Wert (99 beendet): ? 10
Naechster Wert (99 beendet): ? 4
Naechster Wert (99 beendet): ? 6
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 13
Naechster Wert (99 beendet): ? 12
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 20
Naechster Wert (99 beendet): ? 11
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 99
++++20
++15
++++13
++++++12
++++++++11
10
++++6
++4
Drücken Sie eine beliebige Taste . . .
```

```
Alle Tests erfolgreich (71 REQUIREs in 7 Test Cases)
Leerer Baum.
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 2
Naechster Wert (99 beendet): ? 99
3
++2
Drücken Sie eine beliebige Taste . . .
```

```
Alle Tests erfolgreich (71 REQUIREs in 7 Test Cases)
Leerer Baum.
Naechster Wert (99 beendet): ? 99
Leerer Baum.
Drücken Sie eine beliebige Taste . . .
```

# (Pflicht-) Aufgabe INF-09.02: Templates

Programmieren Sie einen `struct` Datentyp Tupel, der zwei Komponentenwerte `komponente1` und `komponente2` von beliebigen, ggfs. unterschiedlichen Datentypen speichert. Definieren Sie Tupel als Template-Datentyp.

Programmieren Sie außerdem eine Template-Funktion …

```
int vergleiche( … p1 … , … p2 … )
```

… die zwei Werte des Tupel-Typs miteinander vergleicht und …
**–1** zurückgibt, wenn beide Komponentenwerte des ersten Tupels kleiner sind als die jeweiligen Parameterwerte des zweiten Tupels, …
**+1** zurückgibt falls beide Komponentenwerte des ersten Tupels größer sind als die jeweiligen Parameterwerte des zweiten Tupels …
und ansonsten den Wert **0** zurückgibt.

Die beiden Parameterwerte der Funktion seien von einem beliebigen, aber identischen Tupel-Typ (also Typen der Komponenten der beiden zu vergleichenden `struct` Werte jeweils identisch, so dass die `struct` Werte auch vergleichbar sind, siehe Hauptprogramm zum Testlauf).

Für die Definition des Tupel-Typs und der Template-Funktion sollen zwei Dateien `tupel.h` und `tupel.cpp` genutzt werden.

In eine Datei `tupel_main.cpp` soll das im Folgenden angegebene Hauptprogramm eingefügt werden. Die Templates sollen die Methodik der expliziten Instanziierung nutzen (damit sollte klar sein, was in die Datei `tupel.h` gehört und was in die Datei `tupel.cpp`).

```
// Datei: tupel_main.cpp

#include <string>
#include <iostream>

#include "tupel.h"

int main()
{
    Tupel<std::string, int> hansi = { "Hansi", 8 };
    Tupel<std::string, int> willi = { "Willi", 77 };

    std::cout << vergleiche<std::string, int>(hansi, willi) << std::endl;

    Tupel<int, int> t1 = { 3 , 4 };
    Tupel<int, int> t2 = { 1 , 2 };

    std::cout << vergleiche<int, int>(t1, t2) << std::endl;

    Tupel<int, int> t3 = { 9 , 1 };
    Tupel<int, int> t4 = { 3 , 5 };

    std::cout << vergleiche<int, int>(t3, t4) << std::endl;

    system("PAUSE");
    return 0;
}
```

## Testlauf *(keine Benutzereingaben)*:

```
-1
1
0
Drücken Sie eine beliebige Taste . . .
```