Aufgabe Programmierung

Scheinkriterien: Zum Bestehen des Praktikums ist es notwendig, insgesamt mindestens 37 Punkte zu erreichen. Zusätzlich gilt, dass pro Theorieblatt mindestens 1 Punkt und pro Praxisblatt mindestens 2 Punkte erreicht werden müssen!

1 Praxis

1.1 Baumdatenstruktur (5 Pt)

1.1.1 Grundgerüst der Baumdatenstruktur (2.5 Pt)

Erstellen Sie ein neues Projekt `ecg_tree` mit zwei neuen Dateien `node.h` und `node.cpp` und implementieren Sie eine Baumdatenstruktur wie folgt.

Die Knotenklasse `node` soll einen Namen vom Typ `std::string` speichern können und folgende Methoden bereitstellen (Typen der Argumente und Rückgabewerte sind bewusst weggelassen und müssen erschlossen werden)

- Konstruktor mit einem Argument vom Typ `const std::string&`, das den Knotenname initialisiert
- Destruktor zum Löschen aller Kindknoten mit dem `delete`-Operator. Deklarieren Sie den Destruktor als virtuelle Methode.
- `get_name() const` ... gibt den Namen des Knotens zurück
- `set_name(new_name)` ... setzt den Namen des Knotens auf einen neuen Namen
- `get_nr_children() const` ... gibt die Anzahl der direkten Kindknoten an
- `get_child(i) const` ... gibt einen Zeiger auf den i-ten direkten Kindknoten zurück
- `add_child(child)` ... fügt am Ende einen neuen direkten Kindnoten hinzu

Nutzen Sie zum Speichern der Kindknotenzeiger die Template Klasse `std::vector` der Standard Template Library, die im Header `<vector>` deklariert ist.

1.1.2 Programmstruktur (1.5 Pt)

Erstellen Sie eine dritte Datei `main.cpp`. Binden Sie `node.h` mit einem entsprechenden `#include`-Befehl ein. Implementieren Sie eine `main`-Funktion, die einen Baum mit einem Wurzelknoten namens `"root"` und zwei Kindern namens `"left child"` und `"right child"` erzeugt und danach den ganzen Baum, mit dem `delete`-Operator angewendet auf den Wurzelknoten, wieder löscht.

1.1.3 Debugging (1 Pt)

Setzen Sie im Destruktor der Knotenklasse einen Break-Point und starten Sie die Anwendung in der Debug-Konfiguration im Debug-Modus. Beobachten Sie, wie der Destruktor rekursiv aufgerufen wird. Erweitern Sie den Destruktor so, dass folgende Ausgabe entsteht (nicht vergessen, `<iostream>` zu inkludieren):

```
enter ~node() of "root"
enter ~node() of "left child"
leave ~node() of "left child"
enter ~node() of "right child"
leave ~node() of "right child"
leave ~node() of "root"
```

1.2 Rekursive Traversierung (5 Pt)

1.2.1 Globale Knotenzählung (1 Pt)

Erweitern Sie die Knotenklasse um eine statische Variable `node_id`, die eine globale Knotennummer mitzählt. Initialisieren Sie diese in `node.cpp` auf `0` und zählen Sie sie im Knotenkonstruktor um eins hoch.

Geben Sie dem Namensparameter im Knotenkonstruktor einen leeren String als Defaultparameterwert und setzen Sie im Konstruktor den Knotennamen auf "`node_`<*node_id*>", falls kein Knotenname angegeben wurde. Dabei sollen die automatisch erzeugten Knotennamen mit "`node_1`" beginnen. Nutzen Sie zur Umwandlung der Knotennummer in einen String die `std::stringstream`-Klasse aus dem Header `<sstream>`, die wie folgt verwendet wird:

```
std::stringstream str_sm;
str_sm << node_id;
std::string node_id_str = str_sm.str();
```

1.2.2 Rekursive Baumerstellung (2 Pt)

Implementieren Sie in `node.cpp` eine Funktion `create_complete_tree( nr_child_nodes, tree_depth)`, die rekursiv einen Baum erstellt, bei dem alle Knoten bis auf die Blattknoten genau `nr_child_nodes` Kindknoten haben und bei dem die Pfade von der Wurzel bis zu den Blättern genau `tree_depth` Knoten enthalten (dabei ist der Wurzelknoten mitzuzählen). Deklarieren Sie die Methode in `node.h` und rufen Sie die Methode mit den Parameterwerten `(2,4)` in der `main`-Funktion auf.

1.2.3 Stream-Ausgabe (2 Pt)

Implementieren Sie in der Knotenklasse eine Methode `print(std::ostream str, ...)`, die einen Baum in dem angegebenen Stream ausgibt. Bei `std::cout` handelt es sich um einen solchen Stream. Folgende Ausgabe soll entstehen, wenn man den Wurzelknoten des von `create_complete_tree(2,4)` erzeugten Baumes ausgibt:

```
node_1
   node_2
      node_3
         node_4
         node_5
      node_6
         node_7
         node_8
   node_9
      node_10
         node_11
         node_12
      node_13
         node_14
         node_15
```

Überlegen Sie sich eine Strategie, wie Sie die Informationen über die Einrückungen mitführen können (bspw. über Funktionsparameter oder statische Variablen) (1.5 Pt).

Überladen Sie den `<<`-Operator für die Knotenklasse so, dass die `print`-Methode aufgerufen wird (0.5 Pt). Dadurch soll es möglich sein, einen Knoten und seine Kindelemente mittels `std::cout << node;` auszugeben. Eine mit `extern` versehene Deklaration des überladenen Operators soll wieder in `node.h` erscheinen. Die Implementierung kommt in `node.cpp`.

1.3 Zusatzaufgaben (maximal 5 Pt)

- Erstellen Sie einen Baum, der einen Zyklus enthält und implementieren Sie eine neue Traversierungsmethode zur Ausgabe ähnlich zum <<-Operator, die Zyklen detektiert, markiert ausgibt und eine unendliche Rekursion vermeidet. Gestalten Sie die Implementierung so, dass Sie keine zusätzlichen Elemente in die Knotenklasse einfügen müssen. Zur Lösung können Sie zum Beispiel eine Map-Datenstruktur verwenden, die Informationen enthält, ob ein bestimmter Knoten bereits besucht wurde (max +5 Pt).

- Implementieren Sie eine iterative Traversierungsmethode, die dieselbe Ausgabe erzeugt, jedoch ohne Rekursion auskommt. Legen Sie dazu einen Stack an, der die zu bearbeitenden Elemente enthält. Dafür können Sie z.B. die Klasse `std::stack` verwenden. Anstelle eines rekursiven Funktionsaufrufes fügen Sie nun neue Elemente an den Beginn des Stacks ein. In einer Schleife werden die Elemente vom Stack geholt und bearbeitet. Das wird solange wiederholt, bis der Stack leer ist (max +3 Pt).

- Extrahieren Sie aus Ihrem Code eine statische Bibliothek namens `tree`, in der die notwendigen Funktionalitäten für die Baumstruktur zu finden sind. Um eine statische Bibliothek zu erstellen müssen Sie ein neues Projekt in derselben Projektmappe anlegen und die relevanten Code-Dateien verschieben. Fügen Sie einen Verweis aus `ecg_tree` auf die statische Bibliothek hinzu, damit der Code beim Kompilieren zur Verfügung steht. (max +2 Pt)

- Erstellen Sie ein Visual Studio Projekt namens `tree_dll`, das aus `node.cpp` eine dynamische Bibliothek erstellt. Beachten Sie, dass beim Erstellen der Dll die Deklarationen in `node.h` mit `__declspec(dllexport)` gekennzeichnet werden müssen, beim Nutzen der Dll im Projekt `ecg_tree` jedoch stattdessen mit `__declspec(dllimport)`. Lesen Sie die Dokumentation zu `__declspec` und ändern Sie Ihre Deklarationen in `node.h` so ab, dass der Header sowohl im statischen Bibliotheksprojekt `tree` wie auch im dynamischen Fall in `tree_dll` verwendet werden kann. Beispiellösungen finden sich in freien Bibliotheken wie `glew` oder `fltk`. (max +3 Pt)

2 Abgabe

2.1 Was

Die Lösung muss in den Laborräumen der Abnahme lauffähig sein. Für die Abnahme setzen wir ein MS Visual Studio (2010, bzw. 2012) Projekt voraus. Stellen Sie also sicher, dass der Quellcode unter diesen Voraussetzungen kompiliert. Es wird ausschließlich Quellcode akzeptiert, der zuvor wie unter „Wie“ beschrieben abgegeben wurde.
Wenn Sie die entsprechenden Software auch zu Hause verwenden wollen, können diese über Dreamspark (`https://www.dreamspark.com`) beziehen. Für Informatikstudenten der TU besteht zusätzlich die Möglichkeit, weitere Software über das Dreamspark Premium-Abonnement der Fakultät zu beziehen.

2.2 Wie

Archivieren Sie alle Quellcode-Dateien und die Visual-Studio-Projektdateien Ihrer Lösung in einer ZIP-Datei. Bitte fügen Sie *nicht* die Kompilate oder die VS-Projekt-Datenbank (*.sdf, *.opensdf) hinzu. Die Abgabe erfolgt via OPAL. Dazu nutzen Sie die Praktikumsseite der entsprechenden Übung.
Über den Punkt „Abgabeordner“ laden Sie Ihre Ergebnisse als ZIP-Datei auf den OPAL Server. Dies können Sie bis zum Abgabetermin (siehe Seitenkopf) mehrfach vornehmen z.B. um Korrekturen an Ihrer Lösung durchzuführen.
Jeder Student muss die Lösung seines Teams über *seinen* OPAL-Account hochladen!