*Dieses Praktikum beschäftigt sich mit den Inhalten aus der Vorlesung **ADS_K2_Lineare_Datenstrukturen** und **ADS_K2_Datenstrukturen_v3_Trees**.*

1 Praktikum 1: Ringpuffer und Binärbaum

Szenario - Sie beginnen ein Praktikum bei „Data Fuse Inc.“, einem führenden Anbieter von BigData-Lösungen. In der Entwicklungsabteilung unterstützen Sie bei Konzeption und Realisierung von experimentellen Lösungen. Bringen Sie erlernte Algorithmen in ein neues Zusammenspiel.

1.1 Backup mittels Ringpuffer

1.1.1 Grundidee

Anwendungsbeispiel - Die Erstellung von Backups im industriellen Umfeld ist äußerst komplex und ein Sicherungssatz kann schnell mehrere Terabyte umfassen. Eine aufwändige Langzeitarchivierung ist jedoch nicht immer erforderlich, da z.B. die Änderungsrate der Daten zu hoch und damit zu teuer ist. Man bedient sich hier des Tricks der Ring-Sicherung, bei der nur eine begrenzte Anzahl Backup-Zyklen vorgehalten werden müssen. Wird ein aktuelles Backup erstellt, muss die jeweils älteste Sicherung gelöscht bzw. überschrieben werden. Diese Grundidee lässt sich auf eine Vielzahl von Problemen anwenden.

1.1.2 Aufgabenstellung

Schreiben Sie ein Programm zur Erstellung und Verwaltung von Backups mittels einer Queue, die Sie als Ringpuffer implementieren. Erstellen Sie hierfür eine Hauptklasse **Ring**, die die Verwaltung des Rings ermöglicht. Diese Klasse realisiert alle erforderlichen Operationen auf der Datenstruktur (z.B. neue Elemente hinzufügen, Inhalte ausgeben, Elemente suchen). Der eigentliche Ring besteht aus insgesamt 6 Knoten der Klasse **RingNode**, die ähnlich einer verketteten Liste verknüpft sind (Abb. 2, 3, 4). Ein leerer Ring wächst mit jedem hinzugefügten RingNode, bis er seine maximale Größe von 6 erreicht hat. Um das Alter eines Datensatzes zu kennzeichnen, besitzt jeder RingNode das Attribut oldAge (Aktuellste: ’0’, der Älteste ’5’). Erreicht der Ring seine maximale Größe, ersetzt ein neu hinzugefügter RingNode immer den Ältesten - das älteste Backup wird überschrieben. Implementieren Sie die im Bild auf der nächsten Seite (Abb. 4) dargestellten Klassen (einschließlich aller Attribute und Methoden) in der bereitgestellten Datei-Vorlage. Bei Bedarf können Sie die Klassen um weitere Hilfsattribute und Hilfsfunktionen erweitern. Lassen Sie aber vorgegebene Strukturen und friend-Hilfsfunktionen unberührt. *Beachten Sie bitte die Lösungshinweise unten, da diese weitere Details zur Implementierung geben!*

Visuelle Darstellung

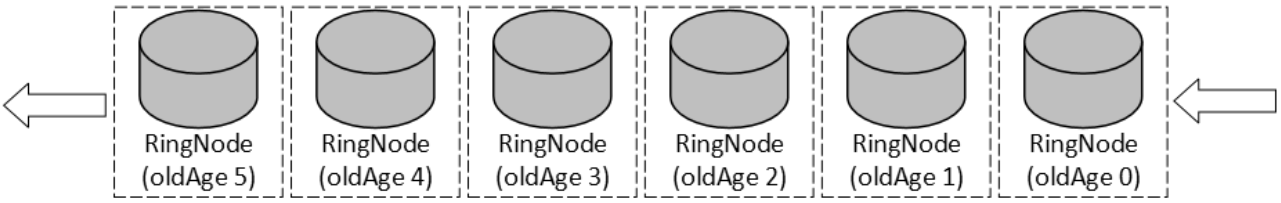

Abbildung 1: Theoretische Warteschlange (Queue) mit fixer Größe. Ältestes Element steht vorne und kann entnommen werden, jüngstes/neues Element wird hinten angestellt

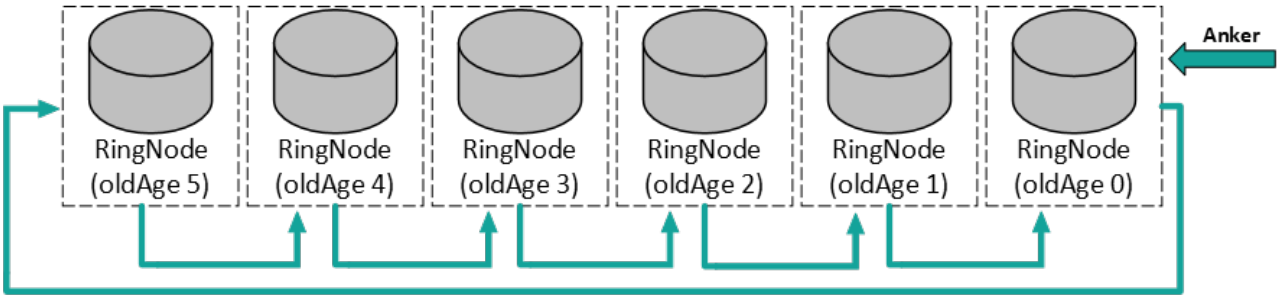

Abbildung 2: Umbau als zyklisch, einfach-verkettete Liste mit max. Knoten bzw. Backups von 6, Verknüpfung mit next-Pointern, jüngstes Element zeigt auf das älteste Element

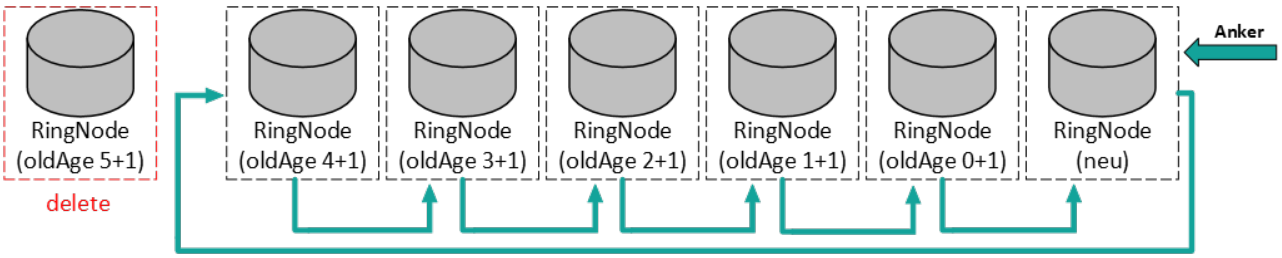

Abbildung 3: Zu implementierender Ausbau als Ringpuffer mit max. Größe von 6. Anker zeigt immer auf das jüngste Element. Beim Hinzufügen altern Nodes (oldAge+1) und Node mit oldAge 6 muss entfernt werden.

Abbildung 4: Datensicherung mittels Ringpuffer, mit Klassen

Der Benutzer soll über die Konsole folgende Möglichkeiten haben:

1. Neuen Datensatz eingeben. Dieser besteht aus Daten des Backups (symbolicData:string) sowie einer Beschreibung (description:string).
2. Suchen nach Backup-Daten. Auf der Konsole sollen Alter, Beschreibungs- und Datentext des betreffenden Backups ausgegeben werden. Sonst eine Fehlermeldung
3. Alle Backup-Informationen ausgeben. Aufsteigende Liste aller Backups, Format siehe Beispiel
4. Programm beenden

```
========================================          // Beispiel: Menü der Anwendung
SuperBackUp-Organizer over 9000, by. Son Goku
========================================
1) Backup anlegen
2) Backup suchen
3) Alle Backups ausgeben
4) Programm beenden
?>
```

```
?> 1                                              // Beispiel: neuer Datensatz
+Neuen Datensatz anlegen
Beschreibung ?>  erstes Backup
Daten ?> echtWichtig1
+Ihr Datensatz wurde hinzugefuegt.
```

```
?> 2                                    // Beispiel: suche Datensatz
+Nach welchen Daten soll gesucht werden?
?> echtWichtig1
+ Gefunden in Backup: Alter 0, Beschreibung: erstes Backup, Daten: echtWichtig1
```

```

?> 2                                  // Beispiel 2: suche Datensatz
+Nach welchen Daten soll gesucht werden?
?> megaWichtig1
+ Datensatz konnte nicht gefunden werden.
```

```
?> 3                  // Beispiel: Ausgabe aller Backups  nachdem weitere Daten
                      //  eingegeben wurden. Sortierung nach Alter
Alter: 0, Beschreibung: sechstes Backup, Daten: insolvenzDaten6
--------------------------
Alter: 1, Beschreibung: fuenftes Backup, Daten: kritischWichtig5
--------------------------
Alter: 2, Beschreibung: viertes Backup, Daten: unfassbarWichtig4
--------------------------
Alter: 3, Beschreibung: drittes Backup, Daten: unglaublichWichtig3
--------------------------
Alter: 4, Beschreibung: zweites Backup, Daten: echtSuperwichtig2
--------------------------
Alter: 5, Beschreibung: erstes Backup, Daten: echtWichtig1
```

Listing 1: Ausgabevorgabe des zu entwickelnden Programms

1.1.3 Testgetriebene Entwicklung

Um Ihnen die Entwicklung und Abgabe des Praktikums zu erleichtern, verfolgen wir einen testgetriebenen Ansatz. Implementieren Sie die Klassen, Attribute und Methoden nach den Vorgaben aus Abbildung 4. Die erwartete Funktionalität der Methoden ist anhand des Namens erkennbar (z.B. addNewNode soll einen neuen Knoten, mit den als Parameter übergebenen Daten, anlegen und im Ring platzieren.) Kern der testgetriebenen Entwicklung ist die Headerdatei catch.h, sowie die Ring-Test.cpp Datei, über die ausführliche Unit-Tests bereitgestellt werden. Eine Datei-Vorlage für Ihre Entwicklungsumgebung (Visual Studio Code) wird Ihnen über Ilias bereitgestellt. **Sie dürfen die Test-Headerdatei und Unittests nicht verändern.**

1.1.4 Lösungshinweise

- Nutzen Sie Ihr Wissen über die verkettete Liste und allgemeine Warteschlange (Queue), da sich die verwendeten Datenstrukturen stark ähneln (siehe Abbildungen 1, 2, 3, 4). Die Verwendung von STL-Containern wie Queue oder List ist allerdings nicht gestattet.
- Um den Aufbau der Datenstruktur in Abbildung 4 besser zu verstehen, sehen Sie sich den schrittweisen Umbau der Struktur von einer Queue als Grundidee (Abbildung 1), über die erste Umstellung als zyklische, 1-fach verkettete Liste (siehe Abbildung 2), über Operationsbeispiele

der umgebauten Liste (Abbildung 3), hin zur ausgebauten Datenstruktur mit Klassen (Abbildung 4) an.

- Achten Sie darauf, dass Sie die Operationen in der richtigen Reihenfolge vornehmen.
- Aktualisieren Sie stets das Attribut m_oldAge-Informationen der Knoten. Platzieren Sie den neuen Knoten richtig in der verketteten Liste.
- Überprüfen Sie, ob Sie das Attribut m_anker aktualisieren müssen. Der Anker zeigt immer auf das jüngste Element.
- In der Abbildung 4 folgt auf m_oldAge 0→next der Knoten mit m_oldAge 5. Es wird damit die Schreibrichtung angezeigt, und dass der älteste Knoten mit m_oldAge 5 als erster überschrieben werden soll.
- Sie dürfen die Klassen um kleine Hilfsmethoden erweitern (wenn Sie es begründen können).

Implementierungshinweise zur Klasse **Ring**

- Erstellen Sie die Klasse Ring als "übergeordnete" (keine Vererbung gemeint) Klasse für die Kontrolle über den eigentlichen Ring.
- Die Klasse Ring soll sich nur um das Handling der Datenstruktur kümmern und keine Menü-Funktionen besitzen (z.B. Menü ausgeben, Menüauswahl einlesen, etc.). Schreiben Sie für die Darstellung des Menüs eigenständige Methoden in der main.cpp oder realisieren Sie dies in der main() selbst.
- Der Datenring besteht aus max. 6 Knoten der Klasse RingNode. Die Knoten sind von außen nicht direkt zu erreichen, sondern nur über die Klasse Ring.
- Der Anker des Ringpuffers zeigt immer auf den aktuellsten RingNode (m_oldAge 0).
- Zu Beginn ist Ihr Ring noch leer und der Anker zeigt auf "null". Anschließend verfahren Sie folgenderweise: Lassen Sie den Ring mit jeder Eingabe dynamisch wachsen bis die maximale Anzahl von 6 Knoten erreicht ist. Erst wenn der Benutzer einen neuen Datensatz speichern will, legen Sie einen neuen Knoten an und fügen ihn dem Ring hinzu. Ist die maximale Anzahl von 6 erreicht, identifizieren Sie den ältesten Knoten, ersetzen ihn durch den neuen Knoten und verfahren weiter wie oben beschrieben. Sie müssen sich hier um die Einhaltung der Obergrenze (6 Knoten) sowie um korrekte Einfüge-Operationen (umbiegen der next-Pointer) kümmern.
- Bei der ersten Einfüge-Operation hat Ihr neuer Knoten den Attributwert m_oldAge=0 und es muss keine Aktualisierung vorgenommen werden, da es noch keine weiteren Einträge gibt.
- Ab der zweiten Einfüge-Operation muss das m_oldAge Attribut aller schon bestehenden Knoten um 1 erhöht werden, da es einen neuen/aktuelleren Knoten mit dem Attributwert m_oldAge=0 gibt und damit alle anderen Knoten "älter"werden.

- Wenn Sie den ältesten Knoten ersetzen, so müssen Sie ihn richtig entfernen (inkl. Änderung der betroffenen Pointer, d.h. Knoten aus der Liste entfernen und dann löschen). Sie dürfen nicht einfach die neuen Attributwerte in den alten Node schreiben.
- Hinweise zu den Attributen und Methoden der Klasse Ring
  - *m_countNodes: int*, Anzahl der Knoten, die bereits im Ring sind.
  - *m_anker: RingNode**, Einstiegspunkt (Anker) zur bestehenden Struktur. Bei leerer Struktur gilt m_anker == nullptr.
  - *addNewNode(string Beschreibung, string Data)*, legt einen neuen Knoten mit den Parameterwerten im Ring an, kümmert sich ggf. um das Überschreiben und Aktualisierung der oldAge Informationen. Kein Rückgabewert.
  - *search(string Data)*, sucht nach Übereinstimmung mit dem Parameterwert im Ring. Es wird nach Daten, nicht nach der Beschreibung gesucht. Konsolenausgabe wie beim Beispiel oben, Rückgabewert entsprechend true oder false.
  - *print( )*, Ausgabe des bestehenden Rings, mit aufsteigendem Alter auf der Konsole. Siehe Beispiel oben.

Implementierungshinweise zur Klasse **RingNode**

- Die Klasse RingNode ist der „dumme“ Datencontainer, in dessen Attribute die Daten der aktuellen Sicherung geschrieben werden.
- Einzige Intelligenz ist der „next“ Pointer auf den nächsten Knoten.
- RingNode muss eine eigene Klasse sein und darf nicht einfach als struct realisiert werden.

1.2 Binärbaum - Datenhaltung und Operation

Anwendungsbeispiel - Aus der letzten Zielgruppenanalyse wird Ihnen ein großer Datensatz übergeben, der für Ihre Analysten nutzbar gemacht werden soll. Stellen Sie Ihren Kollegen ein Programm zur Verfügung, welches den Datensatz einlesen, aufbereiten und verändern kann. Zur hierarchischen Organisation der Datensätze, haben Sie sich für die Verwendung eines Binärbaums entschieden.

1.2.1 Aufgabenstellung

Entwickeln Sie eine Hauptklasse **Tree** für den Baum, die als übergeordnete Klasse (keine Vererbung gemeint) die Kontrolle über den Baum hat und für alle Operationen verantwortlich ist (z.B. Knoten hinzufügen, löschen, suchen, ausgeben, usw.). Knoten des Baumes bestehen aus **TreeNodes**, die die in der Abbildung (5) angegebenen Attribute, Funktionen, sowie die erforderlichen Referenzen auf den links/rechts folgenden TreeNode bzw. Nullpointer besitzen. Den zu realisierenden Aufbau und die zu implementierenden Funktionalitäten der beiden Klassen können Sie der Abbildung 5 entnehmen. Sie dürfen die Klassen um kleine Hilfsmethoden und -attribute erweitern, wenn es für die Umsetzung zwingend erforderlich ist. Wo ein TreeNode im Tree platziert wird, entscheidet sich anhand des Attributs *m_NodeOrderID*. Dieser Integer-Wert errechnet sich aus den Attributen *m_Age*, *m_PostCode* und *m_Income* des Knotens.

$$m\_Age(int) + m\_PostCode(int) + m\_Income(double) = m\_NodeOrderID(int)$$

Um den chronologischen Ablauf der Operationen später nachvollziehen zu können, benötigen Sie für jeden TreeNode eine zusätzliche Seriennummer (ID). Inkrementieren Sie hierfür das Integer-Attribut *m_NodeChronologicalID* des TreeNodes. Beachten Sie, dass die *m_NodeChronologicalID* eine fortlaufende Seriennummer ist und nicht mit der *m_NodeOrderID* zu verwechseln ist. Ihr Baum soll ferner in der Lage sein, Nodes anhand des Attributes *m_Name* zu finden und auszugeben. Berücksichtigen Sie Mehrfachvorkommen von Namen. Weiter sollen Nodes über ihre *m_NodeOrderID* identifiziert und aus dem Baum gelöscht werden können. Löschoperationen müssen den Baum in korrektem Zustand hinterlassen. Weitere Details zu den Operationen (z.B. Löschen und Ausgabe) entnehmen Sie bitte den Lösungshinweisen in Kapitel 1.2.4. *Beachten Sie die Lösungshinweise.*

Der Benutzer soll über ein Menü folgende Möglichkeiten haben :

- Hinzufügen neuer Datensätze als ***Benutzereingabe***
- Importieren neuer Datensätze aus einer ***CSV Datei*** , (Funktion wird vorgegeben)
- Löschen eines vorhandenen Datensatzes anhand der ***NodeOrderID***.
- Suchen eines Datensatzes anhand des ***Personennamens***.
- Anzeige des vollständigen Baums nach ***Preorder/Postorder/Inorder***.
- Anzeige des vollständigen Baums nach ***Levelorder*** mit Level-Angabe.

- Programm beenden.

```
========================================       // Beispiel: Menü der Anwendung
 ADS - ELK - Stack v1.9, by 25th Bam
========================================
1) Datensatz einfuegen, manuell
2) Datensatz einfuegen, CSV Datei
3) Datensatz loeschen
4) Suchen
5) Datenstruktur anzeigen (pre/post/in)
6) Level-Order Ausgabe
7) Beenden
?>
```

```
?>1                      // Beispiel: manuelles Hinzufügen eines Datensatzes
+ Bitte geben Sie die den Datensatz ein:
 Name ?> Mustermann
 Alter ?> 1
 Einkommen ?> 1000.00
 PLZ ?> 1
+ Ihr Datensatz wurde eingefuegt
```

```
?>5         // Beispiel: Anzeigen eines Trees mit mehreren Einträgen  in Preorder
Ausgabereihenfolge ?>pre
 ID | Name             | Age   | Income    | PostCode | OrderID
----+------------------+-------+-----------+----------+-------
   0|        Mustermann|      1|       1000|         1|   1002
   3|              Hans|      1|        500|         1|    502
   5|           Schmitz|      1|        400|         2|    403
   4|           Schmitt|      1|        500|         2|    503
   1|            Ritter|      1|       2000|         1|   2002
   2|            Kaiser|      1|       3000|         1|   3002
```

```
?>6                                          // Beispiel: Anzeige nach Levelorder
 ID | Name             | Age   | Income    | PostCode | OrderID | Level
----+------------------+-------+-----------+----------+---------+------
   0|        Mustermann|      1|       1000|         1|     1002|    0
   3|              Hans|      1|        500|         1|      502|    1
   1|            Ritter|      1|       2000|         1|     2002|    1
   5|           Schmitz|      1|        400|         2|      403|    2
   4|           Schmitt|      1|        500|         2|      503|    2
   2|            Kaiser|      1|       3000|         1|     3002|    2
```

```
?>4                          // Beispiel:  Datensatz suchen
+ Bitte geben Sie den zu suchenden Datensatz an
 Name ?> Ritter
+ Fundstellen:
NodeID: 1, Name: Ritter, Alter: 1, Einkommen: 2000, PLZ: 1, PosID: 2002
```

```
?>3                          // Beispiel: Datensatz löschen
+ Bitte geben Sie den zu loeschenden Datensatz an
 NodeOrderID ?> 503
+ Eintrag wurde geloescht.
```

```
?> 2                              // Beispiel: CSV Import
+ Möchten Sie die Daten aus der Datei "ExportZielanalyse.csv" importieren (j/n) ?> j
+ Daten wurden dem Baum hinzugefügt.
```

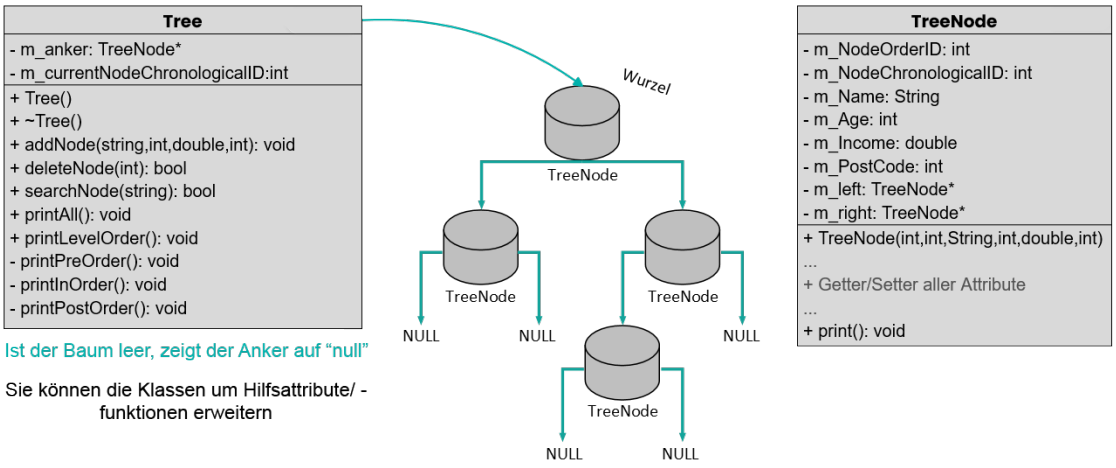

Abbildung 5: Aufbau der Baumstruktur und beteiligten Klassen

Programmieren Sie zusätzlich mehrere Klassen-Methoden ***levelOrder(void)*** in der Klasse Tree, welche die Knoten des Baumes in Level-Order ausgibt. Da die Anzahl der Knoten im Baum selber nicht gespeichert wird, reicht es einfach alle Knoten bis zum Level 10 auszugeben. (Falls es ab einem bestimmten Level keine Knoten mehr gibt, kann abgebrochen werden oder nichts ausgegeben werden). Dafür dürfen Sie gerne rekursive Hilfsmethoden benutzen. Implementieren Sie außerdem die Methoden ***printInOrder, printPreOrder)*** und ***printPostOrder)***, die die Knoten in der entsprechenden Reihenfolge ausgeben.

1.2.2 Testgetriebene Entwicklung

Um Ihnen die Entwicklung und Abgabe des Praktikums zu erleichtern, werden Testroutinen zur Verfügung gestellt. Es ist daher zwingend erforderlich, dass Sie Klassen, Attribute und Methoden nach den Vorgaben aus Abbildung 5 und den Lösungshinweisen implementieren - auch die Benennung ist relevant. Die erwartete Funktionalität der Methoden ist anhand der Namen erkennbar (z.B.

addNode soll einen neuen Knoten, mit den als Parameter übergebenen Daten, anlegen und im Tree platzieren.) Kern der testgetriebenen Entwicklung sind die Dateien catch.h, sowie TreeTest.cpp, über die die ausführlichen Unit-Tests bereitgestellt werden. Sie dürfen diese beiden Dateien, sowie die friend-Deklarationen in den Klassen, nicht verändern! Als Vorlage werden Ihnen wieder die erforderlichen .h und .cpp Dateien zur Verfügung gestellt, die Sie z.B. in VisualStudio importieren können.

1.2.3 Verständnisfragen

Beantworten Sie folgende Fragen nach erfolgreicher Implementierung:

- Die m_NodeChronologicalID eignet sich nicht als Positionsangabe. Warum? Wie würde sich das auf den Baum auswirken?
- Die Erstellung und Verwendung der m_NodeOrderID ist nicht unproblematisch. Warum?
- Was würde passieren, wenn die m_NodeOrderID mehrfach vergeben wird?
- Welche Interpretation lassen die sortierten Daten später zu? Was könnten Sie aus den Zusammenhängen schließen?
- Kennen Sie eine Datenstruktur, die sich für eine solche Aufgabe besser eignen würde?
- Wie kann man die Level-, In-, Pre-, Postorder Ausgabe überprüfen?

1.2.4 Lösungshinweise

Klassen, Methoden und Attribute

- Der Baum **Tree** und die Knoten/Blätter **TreeNodes** sind je eigenständige Klassen, die nicht untereinander vererben.
- Erstellen Sie für die Klasse Tree einen Destruktor, der die Datenstruktur vernünftig aufräumt und löscht. Tipp: Sie können rekursive Funktionen bzw. eine mod. Traversierungsmethode nutzen.
- Die *m_NodeChronologicalID* ist eine fortlaufende Seriennummer für jeden Node, die beim Anlegen eines neuen Datensatzes einmalig vergeben wird und die Einfüge-Reihenfolge nachvollziehbar macht.
- Die *m_NodeOrderID* ist eine errechnete Positionsangabe für jeden Node, die beim Anlegen eines neuen Datensatzes einmalig vergeben wird und aus dem *m_Age*, dem *m_Income* und der *m_PostCode* des Datensatzes errechnet wird. Zweck ist die korrekte Platzierung im Baum.
- Nutzen Sie die Datenkapselung und schützen Sie alle Attribute vor Zugriff. Erstellen Sie Setter/Getter, wenn es sinnvoll ist.
- Der **Tree** besitzt einen Pointer *m_anker* auf den ersten **TreeNode**. Ist der Baum leer, muss dieser Pointer auf den *Nullpointer* zeigen.

- Alle TreeNodes haben zwei Zeiger vom Typ **TreeNode**, die auf die nachfolgenden linken/rechten Knoten vom Datentyp TreeNode verweisen. Gibt es keine Nachfolger, so müssen die Referenzen auf den *Nullpointer* verweisen.
- Verdeutlichung der UML Parameterfolge von Klasse *Tree* (siehe Abbildung 5):
  - *addNode(string Name,int Age, double Income, int PostCode)*, kein Rückgabewert
  - *deleteNode(int NodeOrderID)*, bool Rückgabewert entsprechend true oder false
  - *searchNode(string Name)*, bool Rückgabewert entsprechend true oder false
  - *printAll(void)*, kein Rückgabewert
  - *levelOrder(void)*, kein Rückgabewert (Ausgabe der Knoten)
- Verdeutlichung der UML Parameterfolge der Klasse *TreeNode* (siehe Abbildung 5):
  - *TreeNode(int NodeOrderID, int NodeChronologicalID, string Name, int Age, double Income, int PostCode)*, Konstruktor der Klasse
  - *print(void)*, kein Rückgabewert
  - *Getter/Setter* wie beschrieben

Baum-Operationen

- Das Einfügen eines neuen Knotens vom Datentyp TreeNode erfolgt anhand der errechneten Positionsangabe (*m_NodeOrderID*). Laufen Sie ab der Wurzel die bestehenden TreeNodes ab, vergleichen Sie die Positionsangabe und wechseln in deren Abhängigkeit dann in den linken oder rechten Teilbaum. Sie können davon ausgehen, dass der Benutzer nur gültige Werte eingibt, die nicht zu einer Mehrfachvergabe der *m_NodeOrderID* führen.
- Bei der Suchfunktion soll nach dem Namen einer Person gesucht werden und alle zugehörigen Daten ausgegeben werden. Beachten Sie hierbei, dass *m_Name* kein eindeutiges Schlüsselelement ist und es mehrere Datensätze mit dem Personennamen z.B. „Schmitt“ geben kann. Sie müssen alle passenden Einträge finden und ausgeben. Überlegen Sie, ob hier eine rekursive oder iterativer Vorgehensweise besser ist.
- Löschen eines vorhandenen Datensatzes ist die anspruchsvollste Operation im Binärbaum. Haben Sie die Position des TreeNodes im Tree ausgemacht, müssen Sie auf folgende 4 Fälle richtig reagieren. Der zu löschende TreeNode...
  1. ... ist die Wurzel.
  2. ... hat keine Nachfolger.
  3. ... hat nur einen Nachfolger (rechts oder links).
  4. ... hat zwei Nachfolger.

Vorgabe: verwenden Sie bei Löschoperationen mit zwei Nachfolgern das Verfahren 'Minimum des rechten Teilbaums'. Siehe Vorlesung zum Binärbaum. Denken Sie daran, dass Sie bei einer Löschoperation auch immer die Referenz des Vorgängers auf die neue Situation umstellen müssen.

- Die erwartete Funktionalität der Methoden aus Bild 5 ergibt sich aus der Benennung. Erweitern Sie die Klassen um eigene Methoden und Attribute wenn es zwingend erforderlich ist.
- Sie müssen den Baum **nicht** ausbalancieren.
- Die Klasse zum Knoten TreeNode darf keine Referenz zum Vorgängerknoten enthalten. Dadurch würde ihre Implementierung komplizierter werden.
- Die Attribute der Klasse TreeNode dürfen keine Referenz zum Vorgängerknoten enthalten.

Eingabe der Daten

- Manuelle Eingabe - Wie im Schaubild oben zu erkennen soll der Benutzer über die Konsole einen neuen Datensatz anlegen können. Dabei werden die benötigten Positionen der Reihe nach als Benutzereingabe aufgenommen.
- CSV-Datei - Um einen schnelleren Import zu ermöglichen, soll der Benutzer auch eine CSV Datei einlesen können. Dieser Teil wird Ihnen als Funktion in der main.cpp fertig zur Verfügung gestellt. Beachten Sie dabei folgende Punkte:
  - Die CSV Datei "ExportZielanalyse.csv" liegt im gleichen Verzeichnis wie das Programm. Sie müssen dem Benutzer keine andere Datei zur Auswahl geben oder eine Eingabe für den Dateinamen realisieren. Fragen Sie lediglich, ob die Datei mit dem Namen wirklich importiert werden soll.
  - Die Reihenfolge der Spalten in der CSV Datei, entspricht der manuellen Eingabe (siehe Beispiel oben).
  - Der CSV Import darf die bereits vorhandenen, manuell eingetragenen, Datensätze nicht überschreiben, sondern nur den Tree damit erweitern.

Ausgaben und Benutzerinteraktion

- Funktionen zur Realisierung des Menüs dürfen nicht Teil der Klasse sein. Schreiben Sie diese separat in Ihrer main.cpp.
- Übernehmen Sie das Menü aus dem Beispiel oben.
- Fehlerhafte Eingaben im Menü müssen abgefangen werden.
- Bei fehlgeschlagenen Operationen wird eine Fehlermeldung erwartet.

- Die Ausgabe des gesamten Baums (alle TreeNode Daten, siehe Beispiel) soll nach der ausgewählten Traversierungsmethode erfolgen.
- Formatieren Sie die Ausgaben sinnvoll (siehe Beispiel).