2 Praktikum 2: Ringpuffer und Binärbaum

Szenario - Sie wurden als Systemarchitekt bei „Data Fuse Inc.“, einem führenden Anbieter von BigData- und Swarm-Informationslösungen, eingestellt. In der Entwicklungsabteilung „advanced distributed systems“ sind Sie nun für die Konzeption und Realisierung von experimentellen Lösung zuständig. Zeigen Sie, dass Sie erlernte Algorithmen in ein neues Zusammenspiel bringen können.

2.1 Aufgabenstellung

Die Aufgaben des Praktikums bestehen aus folgenden Teilen:

- Teil 1 - Backup-Rotation mittels Ringpuffer
- Teil 2 - Binärbaum - Datenhaltung und Operation

2.2 Backup mittels Ringpuffer

Anwendungsbeispiel - Die Erstellung von Backups im industriellen Umfeld ist äußerst komplex und eine Sicherung kann schnell mehrere Terabyte umfassen. Eine aufwändige Langzeitarchivierung ist jedoch nicht immer erforderlich, da die Änderungsrate der Daten zu hoch und damit zu teuer ist. Man bedient sich hier des Tricks der Ring-Sicherung, bei der nur ein begrenzter Horizont von Backup-Zyklen vorgehalten wird und ältere Sicherungen gelöscht oder überschrieben werden.

2.2.1 Aufgabenstellung

Schreiben Sie ein Programm zur Erstellung und Verwaltung von Backups mittels Ringpuffer.
Erstellen Sie hierfür eine Hauptklasse **Ring**, die die Verwaltung des Rings und damit der verknüpften Backups ermöglicht. Diese Klasse realisiert alle erforderlichen Operationen auf der Datenstruktur (z.B. Neue Elemente hinzufügen, Inhalte ausgeben, Elemente suchen, usw.).
Der eigentliche Ring besteht aus insgesamt 6 Knoten der Klasse **RingNode**, die untereinander wie eine einfach verketten Liste verknüpft sind. Ein leerer Ring wächst mit jedem hinzugefügten RingNode, bis er seine maximale Größe von 6 erreicht hat. Um die Alterung eines Datensatzes zu simulieren/kennzeichnen (siehe Bild 1), 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 (Abb. 1) dargestellten Klassen (einschließlich aller Attributen und Methoden) in der bereitgestellten Datei-Vorlage. Bei Bedarf können Sie die Klassen um weitere Hilfsattribute und -funktionen erweitern. Lassen Sie aber vorgegebene Strukturen und friend-Hilfsfunktionen unberührt. *Beachten Sie bitte die Lösungshinweise unten,*

da diese weitere Details zur Implementierung geben!

Schreiben Sie in Ihrer main()-Funktion eine GUI, über die der Benutzer folgende Möglichkeiten hat:

- Neuen Datensatz eingeben. Dieser besteht aus Daten für das Backup (SymbolicData:string) sowie einer Beschreibung (Description:string)
- Suchen nach Backup-Daten. Auf der Konsole sollen die Informationen OldAge, Beschreibungs- und Datentext des betreffenden Nodes ausgegeben werden. Sonst eine Fehlermeldung.
- Alle Backup-Informationen ausgeben. Aufsteigende Liste aller Backups, Format siehe Beispiel

```
 OneRingToRuleThemAll v0.1, by Sauron Schmidt   // Beispiel: Menü der Anwendung
================================
1) Backup einfuegen
2) Backup suchen
3) Alle Backups ausgeben
?>
```

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

```
?> 2                                  // Beispiel: suche Datensatz
+Nach welchen Daten soll gesucht werden?
?> echtWichtig1
+ Gefunden in Backup: OldAge 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
OldAge: 0, Descr: sechstes Backup, Data: 0118999
--------------------------
OldAge: 1, Descr: fuenftes Backup, Data: 1337
--------------------------
OldAge: 2, Descr: viertes Backup, Data: 007
--------------------------
OldAge: 3, Descr: drittes Backup, Data: 789
--------------------------
```

```
OldAge: 4, Descr: zweites Backup, Data: 456
-------------------------
OldAge: 5, Descr: erstes Backup, Data: echtWichtig1
```

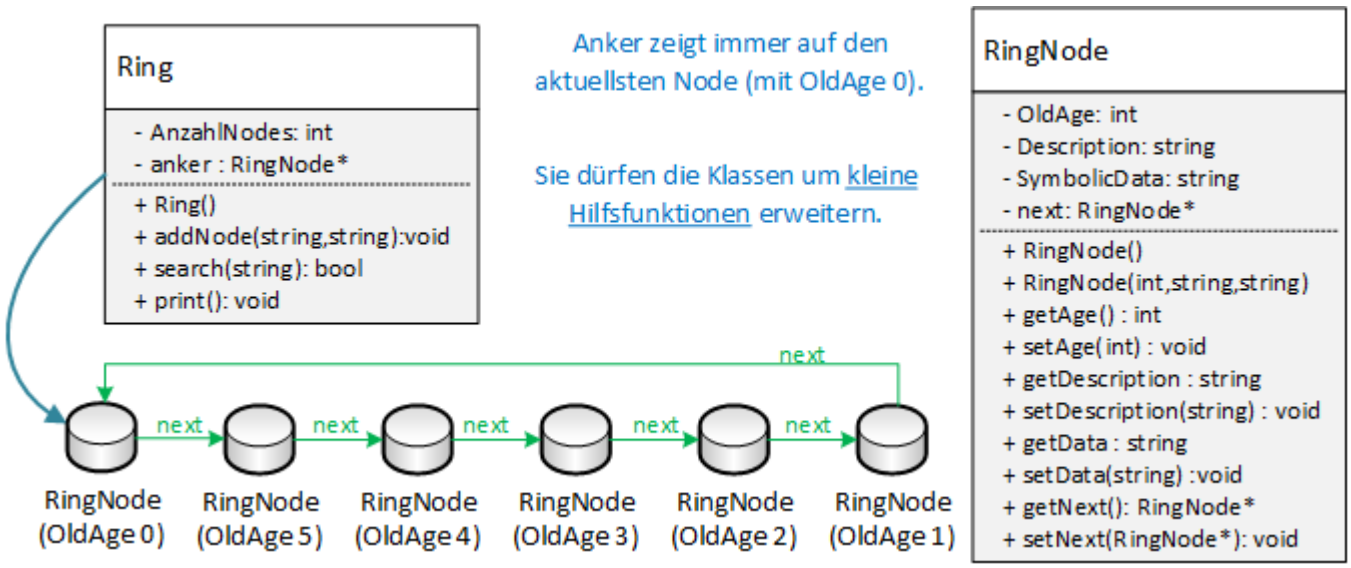

Abbildung 1: Datensicherung mittels Ringpuffer

2.2.2 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 1. Die erwartete Funktionalität der Methoden ist anhand des Namens erkennbar (z.B. addNode soll einen neuen Knoten, mit den als Parameter übergebenen Daten, anlegen und im Ring platzieren.) Kern der testgetriebenen Entwicklung ist die Headerdatei catch.hpp, sowie die RingTest.cpp Datei, über die ausführliche Unit-Tests bereitgestellt werden. Ein Datei-Vorlage für Ihre Entwicklungsumgebung (Visual Studio) wird Ihnen über Ilias bereitgestellt. **Sie dürfen die Test-Headerdatei und Unittests nicht verändern.**

2.2.3 Lösungshinweise

- Nutzen Sie Ihr Wissen über die verkettet Liste, da sich die Datenstrukturen stark ähneln.
- Achten Sie darauf, dass Sie die Operationen in der richtigen Reihenfolge vornehmen.
- Sie müssen die OldAge-Informationen der Nodes immer aktualisieren und den neuen Node richtig positionieren.
- In der Abbildung 1 folgt auf OldAge 0 - next - OldAge 5. Es wird damit die Schreibrichtung angezeigt, und dass der älteste Node mit OldAge5 als erster überschrieben werden soll.

- Sie dürfen die Klassen um kleine Hilfsmethoden erweitern (wenn Sie es begründen können).

Implementierung Klasse **Ring**

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

Implementierung 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 Node.
- Ringnode muss eine eigene Klasse sein und darf nicht einfach als Struct realisiert werden.

2.3 Binärbaum - Datenhaltung und Operation

Anwendungsbeispiel - Ihnen liegt ein großer Datensatz aus der letzten Zielgruppenanalyse vor, der für Ihre Analysten nutzbar gemacht werden muss. 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.

2.3.1 Aufgabenstellung

Entwickeln Sie eine Hauptklasse **Tree**, die als übergeordnete Klasse (keine Vererbung gemeint) die Kontrolle über den Baum hat und für alle Operationen verantwortlich ist (z.B. Node hinzufügen, Node löschen, Node suchen, Node ausgeben, usw.). Der eigentliche Baum besteht aus **TreeNodes**, die jeweils die in der Abbildung (2) angegebenen Attribute, Funktionen, sowie die erforderlichen Referenzen auf den links/rechts folgenden TreeNode bzw. Nullpointer besitzen. Den zu realisierenden Aufbau und die zu implementierenden Funktionalität der beiden Klassen, können Sie der Abbildung 2 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 *NodePosID*. Dieser Integer-Wert errechnet sich aus den Attributen *Alter*, *PLZ* und *Einkommen* des Nodes.

$$Alter(int) + PLZ(int) + Einkommen(double) = Positionsindikator(int)$$

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 *NodeID* des TreeNodes. Beachten Sie, dass die *NodeID* eine fortlaufende Seriennummer ist und nicht mit der *NodePosID* zu verwechseln ist. Ihr Baum soll ferner in der Lage sein, Nodes anhand des Attributes *Name* zu finden und auszugeben. Berücksichtigen Sie Mehrfachvorkommen von Namen. Weiter sollen Nodes über ihre *NodePosID* 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 2.3.4.

Beachten Sie die Lösungshinweise. Beachten Sie die Lösungshinweise. 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***
- Löschen eines vorhandenen Datensatzes anhand der ***PositionsID***.
- Suchen eines Datensatzes anhand des ***Personennamens***.
- Anzeige des vollständigen Baums nach ***Preorder***.

```
==================================    // Beispiel: Menü der Anwendung
Person Analyzer v19.84, by George Orwell
1) Datensatz einfuegen, manuell
2) Datensatz einfuegen, CSV Datei
3) Datensatz löschen
4) Suchen
5) Datenstruktur anzeigen
?>
```

```
?> 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 eingefügt
```

```
?> 5                         // Beispiel: Anzeigen eines Trees mit mehreren Einträgen

ID | Name       | Alter | Einkommen |  PLZ  | Pos
---+------------+-------+-----------+-------+-------
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
```

```
?> 4                          // Beispiel:  Datensatz suchen
+ Bitte geben Sie den zu suchenden Datensatz an
Name ?> Schmitt
+ Fundstellen:
NodeID: 4, Name: Schmitt, Alter: 1, Einkommen: 500, PLZ: 2, PosID: 503
```

```
?> 3                          // Beispiel: Datensatz löschen
+ Bitte geben Sie den zu löschenden Datensatz an
PosID ?> 502
+ Datensatz wurde gelöscht.
```

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

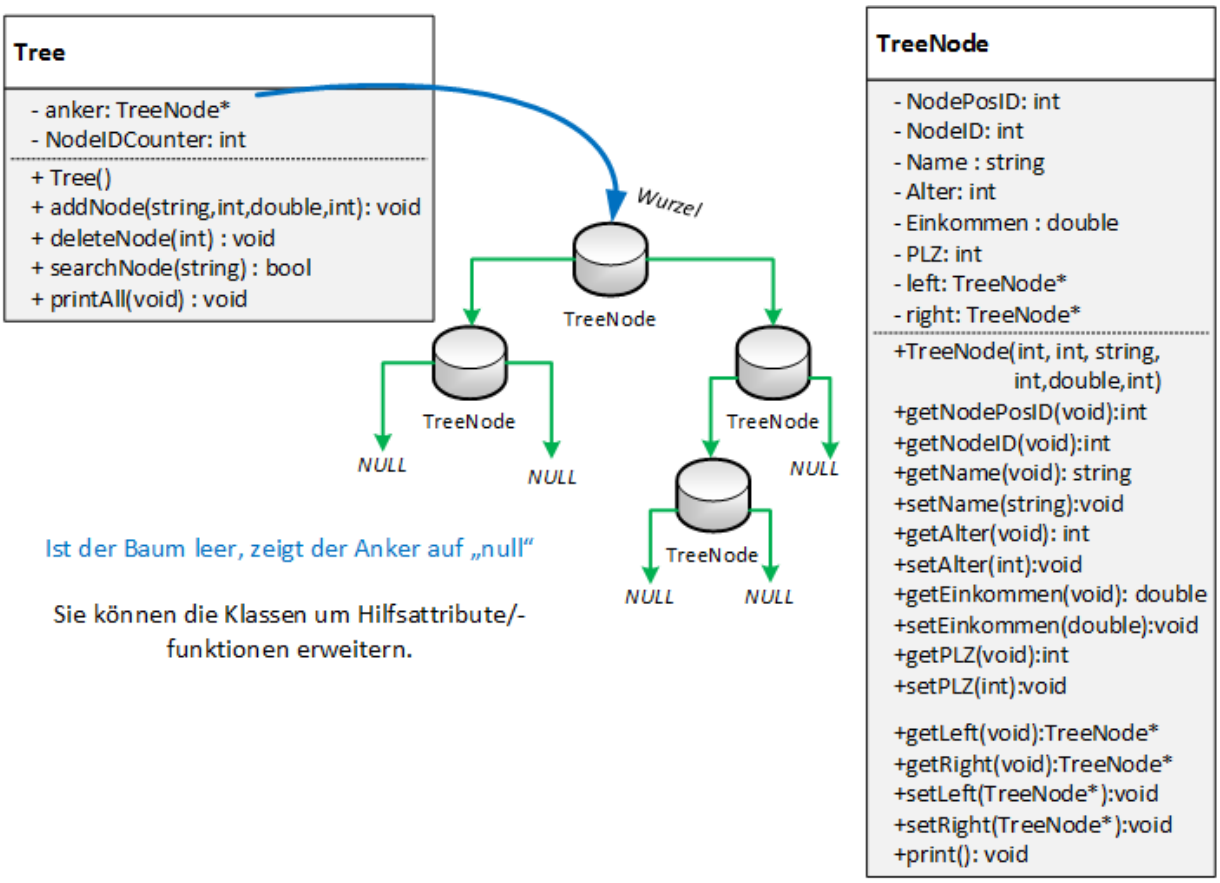

Abbildung 2: Aufbau der Baumstruktur und beteiligten Klassen

2.3.2 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 2 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.

2.3.3 Verständnisfragen

Beantworten Sie folgende Fragen nach erfolgreicher Implementierung:

- Die NodeID eignet sich nicht als Positionsindikator. Warum? Wie würde sich das auf den Baum auswirken?

- Was würde passieren, wenn die NodePosID 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?

2.3.4 Lösungshinweise

2.3.5 Klassen, Methoden und Attribute

- Der Baum **Tree** und die Knoten/Blätter **TreeNodes** sind je eigenständige Klassen, die nicht untereinander vererben.
- Die *NodeID* ist eine fortlaufende Seriennummer für jeden Node, die beim Anlegen eines neuen Datensatzes einmalig vergeben wird und die Einfüge-Reihenfolge nachvollziehbar macht.
- Die *NodePosID* ist ein errechneter Positionsindikator für jeden Node, die beim Anlegen eines neuen Datensatzes einmalig vergeben wird und aus dem *Alter*, dem *Einkommen* und der *PLZ* 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 Poiner *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 nachfolgende rechten/linken TreeNodes verweisen. Gibt es keine Nachfolger, so müssen die Referenzen auf den *Nullpointer* verweisen.
- Verdeutlichung der UML Parameterfolge von Klasse *Tree* (siehe Abbildung 2):
  - *addNode(string Name,int Alter, double Einkommen, int PLZ)*, kein Rückgabewert
  - *deleteNode(int NodePosID)*, kein Rückgabewert
  - *searchNode(string Name)*, Rückgabewert entsprechend true oder false
  - *printAll(void)*, kein Rückgabewert
- Verdeutlichung der UML Parameterfolge der Klasse *TreeNode* (siehe Abbildung 2):
  - TreeNode(int NodePosID, int NodeID, string Name, int Alter, double Einkommen, int PLZ), Konstruktor der Klasse
  - print(void), kein Rückgabewert
  - Getter/Setter wie beschrieben

2.3.6 Tree und Tree-Operationen

- Das Einfügen eines neuen TreeNodes erfolgt anhand der errechneten Positions-ID (*NodePosID*). Laufen Sie ab Wurzel die bestehenden TreeNodes ab und vergleichen Sie die Positions-ID, in deren Abhängigkeit dann in den linken oder rechten Teilbaum gewechselt wird. Sie können davon ausgehen, dass der Benutzer nur gültige Werte eingibt, die nicht zu einer Mehrfachvergabe der PositionsID führen.

- Bei der Suchfunktion soll nach dem Namen einer Person gesucht werden und alle zugehörigen Daten ausgegeben werden. Beachten Sie hierbei, dass *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 u.a. 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 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 2 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.

2.3.7 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.

  – 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.

- Jede Zeile der Datei stellt einen Datensatz dar, der seine Elemente mittels Semikolon trennt. (Hilfestellung: `https://www.c-plusplus.net/forum/281529-full` und `https://de.wikipedia.org/wiki/CSV_(Dateiformat)`)
- Beachten Sie, dass Sie die gelesenen Werte in den richtigen Zieldatentyp übertragen müssen.
- 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.

2.3.8 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 'Pre-Order' Reihenfolge erfolgen.
- Formatieren Sie die Ausgaben sinnvoll (siehe Beispiel).