Technische Universität Hamburg-Harburg
Vision Systems

Prof. Dr.-Ing. R.-R. Grigat

Objektorientierte Programmierung, Algorithmen und Datenstrukturen Praktikumsaufgabe 2019

Max Schürenberg
Sven Painer

21. Mai 2019

1 Einleitung

Diese Praktikumsbeschreibung ist relativ ausführlich gehalten um Ihnen die Bearbeitung zu erleichtern. Bitte verschaffen Sie sich zunächst einen groben Überblick, wesentlich ist Abschnitt 3.

- Abschnitt 1 beschreibt die Bewertung des Praktikums und Software-Werkzeuge (TortoiseSVN, Unit-Test, Doxygen).
- Abschnitt 2 beschreibt den LZW-Algorithmus mit den für dieses Praktikum spezifischen Vorgaben zu Datenstrukturen.
- Abschnitt 3 beschreibt die Implementierungsaufgabe auf Basis der Lösungen zu Übungen 1-7. Es sind zwei unterschiedliche Strategien des LZW-Algorithmus zu implementieren.

Zum Start müssen Sie sich zunächst bei uns registrieren und erhalten dann eine e-Mail mit den Zugangsdaten zum SVN Repository. Dies ist weiter unten beschrieben.

1.1 Subversion SVN und das Subversive Team Plug-In

Sie arbeiten mit SVN, einem *Versionsverwaltungssystem* auf unserem Server und laden dort bitte täglich Ihre Zwischenstände hoch, soweit Änderungen erfolgt sind (Commit-Befehl). Dies ist die professionelle Arbeitsweise in Teams, auch wenn das Team in diesem Fall nur aus Ihnen besteht. Und wenn Ihre Festplatte ausfällt oder gestohlen wird bleibt zumindest Ihr Projektstand erhalten.

Für die Kommunikation mit dem SVN Repository verwenden wir Tortoise SVN im Windows Explorer, dies müssen Sie separat herunterladen und installieren (`http://tortoisesvn.net/`).

Alternativ ist in Eclipse bereits das Plug-In *Subversive* installiert, so dass Sie Ihre Files auch direkt aus Eclipse heraus ins SVN Repository laden können. Allerdings gibt es bei der ersten Verbindung eine Zertifikatwarnung, obwohl unser Zertifikat aktuell ist. Für Linux wäre z.B. kdeSVN geeignet.

1.1.1 Zugangsdaten zum SVN

Melden Sie sich bitte auf dem Webformular unter folgendem Link zum Praktikum an.

`https://tux.ti1.tu-harburg.de/infing/anmelden.php`

User: student
Passwort: sommer
Bitte verwenden Sie Ihre **TUHH e-Mail Adresse**. Senden Sie das ausgefüllte Formular ab, dann erhalten Sie per e-Mail Ihre individuellen Zugangsdaten für Ihr persönliches SVN Repository.

1.1.2 Erstellung des Eclipse-Projekts für das Praktikum

- Kopieren Sie im *Project* Explorer von Eclipse (nicht im Windows Explorer!) das Projekt Blatt2 (Strg+C, Strg+V) und geben Sie der Kopie im sich öffnenden Fenster den Namen *Praktikum*. Dabei entsteht vermutlich eine Fehlermeldung Building workspace has encountered a problem, die aber nur mit Texlipse zu tun hat (LaTeX für die Doxygen Dokumentation) und ignoriert werden kann.

- Löschen Sie aus dem Verzeichnis Praktikum/src im Project Explorer alle (cpp und hpp) Files.

- Wechseln Sie nun im *Windows* Explorer in das Verzeichnis Praktikum/src in Ihrem Workspace-Verzeichnis im portableDevEnv. Führen Sie einen SVN Checkout durch und nennen als Zielverzeichnis Ihrer Arbeitskopie genau dieses src Verzeichnis Praktikum/src (falls es nicht schon automatisch eingetragen ist). Alle Details dazu finden Sie im folgenden Abschnitt. Das Quellverzeichnis, User und Password entnehmen Sie der e-Mail, die Sie nach Anmeldung zum Praktikum von webmaster@ti1.tu-harburg.de an Ihre TUHH e-Mail erhalten haben.

- Legen Sie für spätere eigene Tests eine Kopie der cpp und hpp Files des Verzeichnises src an, z.B. als Verzeichnis srcOriginal. Dieses sind nämlich die Files, mit denen später Ihre Lösung auch durch uns getestet wird.

- Im Projekt Explorer im Kontextmenü->Refresh, dann werden alle neuen Files angezeigt.

- Entwickeln Sie nun wie üblich Ihr Projekt Praktikum in Eclipse.

- Immer wenn Sie Ihre Arbeit für diesen Tag beenden, klicken Sie im Windows Explorer mit Rechtsklick auf das Verzeichnis src und führen erst *SVN Update* durch, dann *SVN Commit* und geben beim Commit einen inhaltlich aussagekräftigen Kommentar zur Änderung der Files ein.

1.1.3 Erstellen der Working Copy mit TortoiseSVN

Laden Sie TortoiseSVN herunter und installieren es auf Ihrem Rechner (`http://tortoisesvn.net/`).
Die Befehle finden Sie im Kontextmenü (Rechtsklick) des Windows Explorers.

Erstellen Sie auf Ihrem Computer eine *lokale Kopie* Ihres SVN-Bereichs aus dem Server, synonym auch *Arbeitskopie* oder *Working Copy* genannt. Gehen Sie im Windows Explorer in Verzeichnis, worin das Arbeitsverzeichnis angelegt werden soll. Rechtsklick -> SVN Checkout...-> URL of repository, tragen Sie Ihre Repository Adresse ein, siehe Ihre e-Mail (bitte Ihre Matrikelnummer einsetzen):

`https://tux.ti1.tu-harburg.de/repos/infing/<MatrikelNummer>`

Checkout directory ist das Verzeichnis Ihrer lokalen Arbeitskopie, den Pfad und Namen können Sie im Prinzip frei wählen. Wenn Sie dazu Ihren Eclipse Workspace wählen, dann haben Sie Ihre Arbeitskopie stets dabei, auch wenn Sie mit portableDevEnv (z.B. auf USB Stick) auf einen anderen Rechner wechseln. Vorschlag ist also <PfadZuPortableDevEnv>/workspace/Praktikum als Arbeitsverzeichnis.

Fully recursive und *HEAD revision* ist schon voreingestellt, die HEAD revision ist stets die neueste Version Ihrer archivierten Software. OK drücken, fertig.

Dateien können neu aufgenommen werden, dazu mit rechter Maustaste auf das File klicken, TortoiseSVN → add. Damit ist das File in der Arbeitskopie markiert, erkennbar am blauen Kreuz. Wählen Sie nur Quellfiles aus, *keine temporären* Files aus wie z.B. *.aux, *.log, oder Resultate, denn diese werden ohnehin alle bei jeder Übersetzung neu erzeugt und würden ohne Nutzen Kapazität im SVN belegen. Sie können Files oder File-Extensions von der Versionsverwaltung explizit ausschließen mit Rechtsklick-> TortoiseSVN -> Add to ignore list. Archiviert werden in userem Fall (nur) alle *.hpp und *.cpp Files.

Die Files unter SVN-Verwaltung erkennen Sie im Explorer am grünen Haken. Sobald Sie ein File modifizieren ändert sich der grüne Haken in ein rotes Ausrufezeichen, Ihre Arbeitskopie ist bei rotem Ausrufezeichen gegenüber dem SVN-Repository geändert.

Sie synchronisieren mit dem Server, indem Sie zunächst ein SVN->Update durchführen, also die aktuellste Information vom Server abrufen. Sollten dort inzwischen Änderungen vorgenommen worden sein, so werden diese mit Ihren eigenen Änderungen zusammengeführt (Merge). Dies könnte passieren, wenn Sie zu Hause und im CIP Pool auf unterschiedlichen Arbeitskopien parallel Änderungen vorgenommen haben, und zwar sobald Sie von einem Rechner ein Commit und vom anderen Rechner dann ein Update ausführen. Ohne das Update und Merge würde der zweite Rechner das Commit des ersten Rechners überschreiben, dessen Änderungen wären in der neuen Head-Revision nicht mehr enthalten.

Gelingt das Merge bei Konflikten nicht automatisch, so müssen Sie von Hand den Merge vornehmen, hierzu können Sie z.B. WinMerge installieren.

Nach dem Update und ggf. dem Merge folgt das Hochladen (Commit), also im Explorer mit rechter Maustaste auf das Verzeichnis der Arbeitskopie klicken und im Kontextmenü dann auf **SVN Commit...**. Es öffnet sich ein Fenster. Tragen Sie in das Feld **Message** bitte aussagekräftige Kommentare zu den aktuellen Änderungen ein (wichtig!). Über **Recent Messages** können Sie sehen, welche Änderungen Sie oder Ihre Kollegen in letzter Zeit vorgenommen haben, vorausgesetzt es wurden tatsächlich Kommentare in das Feld Message eingetragen. Im Feld **Changes made** können Sie Files wählen oder abwählen. Drücken Sie OK, nun wird das Repository auf dem Server auf den Stand Ihrer Arbeitskopie aktualisiert, die Head-Revision ist nun identisch zu Ihrer Arbeitskopie, in Ihrer Arbeitskopie ändern sich die roten Pfeile in grüne Haken, die Revisionnummer der Head-Revision wurde erhöht.

TortoiseSVN ist verbreitet, bei Bedarf finden Sie gute Tutorials zu TortoiseSVN im Internet.

1.1.4 Konflikte mit TortoiseSVN auflösen (Merge)

TortoiseSVN bietet für das obige Beispiel von Konflikten nach Update folgende Lösungen. Bei Konflikt in einem File test.cpp werden zwei Versionen erzeugt, test.cpp.mine (Ihre Arbeitsversion) und test.cpp.theirs (Version auf dem Server).

1. **Konflikte auflösen und eine Mischung beider Versionen verwenden**
   Die Version im Repository wird durch eine beliebige Kombination der Versionen ersetzt.
   - Bearbeiten Sie *test.cpp* nach Bedarf und entfernen Sie in der Datei alle von SVN hinzugefügten Markierungen. Die Funktion *Edit conflicts* bietet einen Editor zum Mergen der Dateien.
   - Wenden Sie *Resolved* auf die Datei an.
2. **Lokale Version verwenden**
   Die Version im Repository wird durch die lokale Version ersetzt.
   - Kopieren Sie den Inhalt von *test.cpp.mine* nach *test.cpp*
   - Wenden Sie den Befehl *Resolved* auf *test.cpp* an, um SVN zu signalisieren, dass Sie den Konflikt behoben haben
3. **Version des Repositories verwenden**
   Die lokalen Änderungen werden rückgängig gemacht und stattdessen die Version aus dem Repository verwendet.

- Wenden Sie den Befehlt *Revert* auf die konfliktverursachende Datei an
- Führen Sie ein *Update* durch.

1.1.5 Die wichtigsten Befehle in Subversive Team

Hier folgen noch einmal die wichtigsten SVN-Befehle. Alle TortoiseSVN Befehle finden Sie im Kontextmenü bei Rechtsklick->TortoiseSVN auf ein Directory oder auf ein File im Windows Explorer. Darüber hinaus sind im folgenden die Befehle von Subversion Team genannt, falls Sie dieses Plug-In verwenden möchten, einfacher zu nutzen ist allerdings TortoiseSVN.

Add to version control... Kopieren Sie als Beispiel Ihre Files CForwardCounter.cpp und CForwardCounter.h in das Verzeichnis src des Praktikumsprojekts. Diese sind nun im Project Explorer mit einem Fragezeichen markiert. Nun müssen die Files dem SVN bekannt gemacht werden, sonst werden sie dort nicht gesichert. Das Bekanntmachen geschieht im Kontextmenü über Team → Add to version control... . Das Fragezeichen ist als Hinweis für den Erfolg nun einem gelben Rechteck gewichen.

Add to svn:ignore... Wichtig, es sollen in der Regel nur *Quellfiles* zur Versionskontrolle hinzugefügt werden, also keine temporären Dateien der Übersetzung und auch keine Binärfiles wie exe-Files oder Bibliotheken. Grund ist, dass ein SVN Repository nie etwas vergisst (nicht einmal bei *SVN Delete*) und mit jedem Commit Befehl wächst. Daher macht es keinen Sinn, den Speicherplatz mit temporären Versionen zu verschwenden, die durch einen Build Vorgang ohnehin jederzeit neu erstellt werden. Also fügen Sie alle cpp-Dateien und hpp-Dateien zum Repository hinzu, mehr nicht. Temporäre Dateien fügen Sie über Team → Add to svn:ignore... der Liste der zu ignorierenden Files hinzu. Die Regel kann entweder für die Dateiendung oder für das individuelle File erstellt werden. Die Files besitzen dann im Project Explorer kein SVN-Symbol mehr und werden bei Commit auch nicht einbezogen. Offenbar ist es aber nicht möglich, ein File der ignore-Liste hinzuzufügen, sobald es einmal in die Versionsverwaltung einbezogen war, svn:ignore ist dann grau dargestellt und kann nicht gewählt werden. Hier hilft nur SVN Delete, was unproblematisch ist weil es sich ja ohnehin um ein temporäres File handelt. Dann wird das temporäre File neu erzeugt und kann svn:ignore hinzugefügt werden.

Alternativ kann man Ausnahmen hinzufügen unter *Window* → *Preferences* → *Team* → *Ignored Resources* mittels *Add Pattern...*, also z.B. die Zeichenkette *.*project*.

Sie sollten allerdings nicht das Verzeichnis .settings mit svn:ignore versehen. Dieses Verzeichnis existiert nicht immer, kann dann aber auch Dateien zur Verwaltung von SVN enthalten und dies führt daher für SVN zu Konflikten.

.profile und .cprofile enthalten XML-Metadaten zu den Projekteinstellungen und sollten durchaus der Versionsverwaltung unterstellt werden, damit Sie im Problemfall nicht alle Einstellungen verlieren. Während .profile immer existiert, existiert .cprofile nur für Projekte die mit der C++ Entwicklungsumgebung CDT angelegt sind. Falls Sie allerdings absolute Pfade in den Properties verwenden, dann können diese auf einem anderen Rechner falsch sein, so dass eine Versionsverwaltung der beiden Files dann weniger Sinn ergibt.

Update, Edit conflict, Merge Bevor wir eine neue Version zum SVN senden führen wir grundsätzlich die Update-Operation aus, Rechtsklick auf Verzeichnis src → Team → Update. Sollte im SVN-Repository bereits ein Teamkollege eine Änderung eines Files gespeichert haben, dann ist deren Versionsnummer natürlich höher als von der Versionsnummer Ihrer lokalen Kopie. Daher wird bei Update die neuere Version aus dem SVN in Ihre lokale Kopie geladen und automatisch in Ihr File eingearbeitet. Sollten Sie beide dieselbe Stelle modifiziert haben, dann gibt es einen Konflikt, den Sie mit Rechtsklick auf das File → Team → *Edit Conflicts* selber auflösen müssen, alternativ mit Merge (oder einem externen Programm wie WinMerge).

Commit... Erst nach einem Update und gegebenenfalls dem Auflösen von Konflikten sollten Sie Ihre neue Version in Ihrer lokalen Kopie mit *Commit* im SVN speichern mit Rechtsklick auf das Verzeichnis src → Team → Commit... . Es öffnet sich ein Fenster, in das Sie unbedingt eine Erläuterung der Änderungen seit dem letzten Commit schreiben sollten. Diese Erläuterungen helfen später, die Versionen mit bestimmten Änderungen wiederzufinden und werden von uns auch kontrolliert. Durch Commit erhalten die ins SVN geladenen Dateien dort eine neue Versionsnummer. *Wir erwarten von Ihnen, dass Sie ca. zehnmal Ihre Zwischenversionen einschließlich aussagekräftiger Kommentare mit Commit... ins SVN Repository laden, beispielsweise abends nach Abschluss der Arbeiten, ansonsten wird Ihr Praktikum nicht anerkannt.*

Cleanup Im Kontextmenü unter Team → Cleanup können Sie die Situation retten, falls die Kommunikation mit dem SVN während einer Transaktion unterbrochen wurde und das SVN nun keine Befehle mehr akzeptiert. Es werden bei Cleanup die noch nicht fertig bearbeiteten SVN-Befehle zu Ende geführt, so dass neue Befehle wieder möglich sind.

SVN Delete Löscht das File aus der lokalen Kopie und bei Commit aus der nächsten Revision. Lädt man danach eine ältere Revision, so ist das File wieder da, denn

das SVN vergisst schließlich nichts. Man kann über *Update to Revision...* sogar einzelne Files oder Verzeichnisse gezielt restaurieren.

Im Gegensatz dazu löscht ein Delete im *Windows Explorer* das File aus der lokalen Kopie und es wird beim nächsten Update aus dem SVN restauriert.

Show History Zeigt die bisher mit Commit erzeugten Versionen einschließlich der Textkommentare.

1.2 Bewertung Ihrer Lösung zur Praktikumsaufgabe

Um das Software-Praktikum zu bestehen müssen folgende Prüfungskriterien erfüllt werden:

- Es treten keine Compilerfehler, Linkerfehler oder Laufzeitfehler auf.
- Die **Unit-Tests** liefern korrekte Ergebnisse bei Codierung und Decodierung. Um dieses Kriterium zu überprüfen, werden die einzelnen Klassen Ihres Projekts und die Gesamtfunktion des LZW-Algorithmus durch Unit Tests von uns überprüft. Siehe hierzu Abschnitt 1.3. In mainPraktikum.cpp werden wir später noch eine Ihnen unbekannte Zeichenfolge testen. Alle Sonderfälle (lange Folgen gleicher Zeichen, Sonderzeichen, leerer String) sind aber in der mainPraktikum.cpp Originalversion bereits enthalten. *Alle* Files des Originalzustands werden vor unserem Test durch deren Originalversionen überschrieben, insofern sollten Sie am Ende sicherstellen, dass mit den Originalversionen die Tests fehlerfrei durchlaufen. Die Originalversionen dieser Files finden Sie dazu unter der ältesten Revisionsnummer mit *Update to Revision...* .
- **Ein- und Ausgaben abschalten:** Für die Bewertung ist es zwingend erforderlich, dass das Programm von alleine terminiert und nicht auf eine Tastatureingabe wartet. Weiterhin sollten während des Programmlaufs keine Kontrollausgaben auf dem Bildschirm gemacht werden. Bitte denken Sie also daran, nach Fertigstellung entsprechende Programmzeilen aus Ihren Quellfiles auszukommentieren. Die Bemerkung bezieht sich natürlich nicht auf mainPraktikum.cpp, denn dieses File ersetzen wir für den Test durch eine Version, die nicht stoppt.
- Im **Subversion SVN** Repository haben Sie während der Bearbeitungszeit des Praktikums mindestens 10 Zwischenzustände Ihrer Lösung inklusive Code und Kommentar mittels SVN Commit gespeichert.
- **Plagiatcheck**: Sollte unsere Plagiatprüfung fündig werden, dann werden stets beide bzw. alle nahezu identischen Versionen disqualifiziert. Bitte implementieren Sie die Lösung daher vollständig selbst und kommentieren Sie ausführlich selbst

Ihren Code. Natürlich dürfen Sie den LZW-Algorithmus und das Konzept der Implementierung auch in Gruppenarbeit erarbeiten. Die Implementierung und Kommentierung sollen Sie dann aber vollständig selbständig erarbeiten, nur so erwerben Sie die angestrebte Erfahrung und Übung.

- **Welche Files werden im SVN benötigt?** Alle von Ihnen erstellten Quellfiles *.cpp und Headerfiles *.hpp werden im SVN benötigt. Alle Quellfiles und Headerfiles müssen im Verzeichnis src liegen. Alternativ können diese Files auch vollständig direkt im Projektverzeichnis liegen, bitte aber nicht beide Varianten gleichzeitig.

  Für Programmierung und Test dürfen Sie das File mainPraktikum.cpp gerne modifizieren. Sie werden vermutlich während Ihrer Programmentwicklung in mainPraktikum.cpp die Tests temporär auskommentieren. Am Ende müssen Sie aber unbedingt mit der Originalversion von mainPraktikum.cpp testen, so dass wirklich alle Tests fehlerfrei durchlaufen. Vergessen Sie auch nicht, ganz am Ende von Teil 2 auch den Test von Teil 1 noch einmal zu überprüfen. Alle Files, die Sie zu Beginn vorgefunden haben, werden von uns vor dem Test mit deren Originalversion überschrieben.

  Falls Sie andere Files oder weitere Directories ins SVN eingecheckt haben (was nicht erforderlich ist), so brauchen Sie diese am Ende nicht explizit zu entfernen, da der Test diese entweder überschreibt oder ignoriert.

1.3 Unit-Tests

Wie schon in der Übung testen wir Ihren Code mit Hilfe von Unit-Tests des *Google C++ Testing Framework*. Die Unit-Tests prüfen Ihre einzelnen Klassen sowie das Gesamtprojekt auf korrekte Funktionalität mit Hilfe von Beispieleingaben. Die Unit-Tests stehen Ihnen zur Verfügung. Dennoch sollten Sie insbesondere den LZW-Trie-Algorithmus mit einem kurzen Eingabestring einmal selbst von Hand durchführen.

Die folgende Tabelle beschreibt alle Unit-Tests, mit denen Ihr Programm getestet wird. Für ein Bestehen des Praktikums ist es notwendig, dass alle Tests fehlerfrei ausgeführt werden. Das Ausführen aller Tests kann eine gewisse Zeit dauern. Bitte warten Sie, bis am Ende das Ergebnis der Tests steht.

Testcase	**Test**	**Beschreibung**
CForwardCounterTest	InitialZero	Der Wert eines Vorwärtszählers muss nach der Initialisierung 0 sein.
	Counting	Beim Aufruf der Methode `count()` auf einer Instanz eines Vorwärtszählers muss der Wert des Zählers um 1 erhöht werden.
CEntryTest	InitialEmpty	Das Attribut `m_symbol` der Klasse `CEntry` muss nach der Initialisierung ein leerer String sein.
	SetSymbol	Das Attribut `m_symbol` der Klasse `CEntry` muss sich über die Methode `setSymbol(string symbol)` setzen lassen.
	CountInstances	Die Klassenmethode `getNumber()` der Klasse `CEntry` muss die genaue Anzahl an Instanzen der Klasse zurückliefern.
CArrayTest	Array	Die Klasse `CArray` soll sich wie ein Array verhalten. Es soll also über den Indizierungsoperator auf alle Elemente zugegriffen werden können.
	Exception	Der Indizierungsoperator der Klasse `CArray` soll eine Exception vom Typ `XOutOfBounds` werfen, wenn ein ungültiger Index angegeben wird.
CKnotTest	Symbol	Das Attribut `m_symbol` der Klasse `CKnot` muss initial ein leerer String sein und sich über die Methode `setSymbol (string symbol)` setzen lassen.
	Parent	Das Attribut `m_parent` der Klasse `CKnot` muss initial -2 sein und sich über die Methode `setParent(int parent)` setzen lassen.
CDoubleHashingTest	SimpleHashing	Die erste Hashfunktion des Hashing-Algorithmus wird getestet. Es findet also noch kein doppeltes Hashing statt.
	DoubleHashing	Die zweite Hashfunktion des Hashing-Algorithmus wird getestet. Hier findet ausschließlich mehrfaches Hashing statt.

Testcase	Test	Beschreibung
CLZWListTest	Encoder	Es werden mehrere Zeichenketten mit der Array-Implementierung des LZW-Algorithmus kodiert und die Ergebnisse mit den richtigen Werten verglichen.
	Decoder	Dann wird wieder Decodiert und das Resultat mit den Originaldaten verglichen.
CLZWTrieTest	TestPhrases	Dieser Testcase wird erst im zweiten Teil ausgeführt. Es finden die gleichen Encoder- und Decoder-Tests wie bei der Array-Implementierung statt, nur diesmal mit der Trie-Implementierung des LZW.

1.4 Dokumentation mit Doxygen

Wesentlicher Bestandteil einer Software-Entwicklung ist Dokumentation. Eine andere Person sollte anhand der Dokumentation den Programmfluss verfolgen können.

Insbesondere für alle Schnittstellen (Deklarationen von Klassen und deren Memberfunktionen und Membervariablen) sollen alle Kommentare im Kommentarformat von *Doxygen* erstellt werden. Dies betrifft also die Header-Files *.hpp.

Kompilieren Sie mit *Doxygen* ein Handbuch, so wie Sie es bereits in den Übungen durchgeführt haben. Verwenden Sie einzeilige Kurzkommentare und (nach Bedarf) ausführliche Blockkommentare.

Wir erwarten, dass Sie *alle* Teile des Praktikums angemessen kommentieren (wie es hoffentlich für die Übungen bereits erfolgt ist). Im Praktikum müssen Sie unter anderem Headerdateien selbst erstellen. Diese versehen Sie bitte mit derselben Kommentarstruktur, die Sie in der bereitgestellten Headerdatei vorfinden:

- *Blockkommentar* ganz oben, der die Aufgabe der Klasse ausführlich erläutert (Doxygen zwingend)
- Blockkommentar über jeder *Funktion*, der die Funktionsweise, die Parameter und die Rückgabewerte erläutert (Doxygen zwingend)
- Kommentar über (oder hinter der Deklaration) jeder *Membervariablen*, der ihre Verwendung in Stichworten erläutert (Doxygen zwingend)
- Darüber hinaus Kommentare zu allen Programmzeilen, soweit diese nicht zweifellos (z.B. anhand der Variablenbezeichnungen) selbsterklärend sind. Da diese

Kommentare sich nicht auf Anwenderschnittstellen beziehen ist hier Doxygen nicht zwingend erforderlich, es genügen reine C++ Kommentare.

Kommentieren Sie stets so, dass Sie nach 6 Monaten Ihr Programm auf Anhieb wieder verstehen. Meist ist dies ohne Kommentare bereits nach Tagen nicht mehr so einfach. Nicht zuletzt an den Kommentaren lassen sich Plagiate in der Regel leicht erkennen, bitte geben Sie sich damit Mühe.

1.5 Tipps

Debugger Verwenden Sie den Debugger um sich Werte anzusehen und den Ablauf zu kontrollieren, Beschreibung in Übung 1 (neue Fassung).

Zeilennummern Schalten Sie Zeilennummern ein, um die Fehlermeldungen besser zuordnen zu können (Rechtsklick auf linken Rand des Editors → Show Line Numbers)

Projekte nur im Project Explorer kopieren So werden auch die Pfade korrekt angepasst.

Flussdiagramme Erstellen Sie einen Ablaufplan oder ein Flussdiagramm Ihres Programms bevor Sie dieses implementieren und spielen Sie dann auf dem Papier ein konkretes Beispiel durch. Dies kann Ihnen insbesondere beim Decoder erheblich Zeit sparen. Erst wenn Sie den *Decoder* z.B. aus dem Skript verstanden haben sollten Sie mit der Implementierung beginnen.

Skizzen zur Datenstruktur Machen Sie sich auch Skizzen zu der Datenstruktur z.B. von `m_symbolTable`, die auf ein CArray aus `CEntry` Objekten zeigt.

Array in Trie-Aufgabe Lassen Sie sich nicht davon verwirren, dass sowohl die Array-Aufgabe als auch die Trie-Aufgabe ein Array verwenden. Im ersten Fall werden die Tabellenzeilen linear (also direkt hintereinander weg) abgelegt und jedes Feld kann Strings aus mehreren Zeichen enthalten. Dies ist einfacher zu verstehen, führt wegen der Mehrfachspeicherung von Zeichen aber zu ineffizienter Speichernutzung.

Im zweiten Fall ist der Trie-Baum in einem Array mittels Hashing abgelegt und jedes Arrayfeld enthält *nur ein einzelnes* Zeichen eines Strings, es wird also viel weniger Speicherplatz benötigt.

Eingabe von leerem String Es wird auch der leere String getestet. Dieser muss also abgefangen werden, weil es sonst die Ursache von Programmabstürzen sein kann, es ist halt beim leeren String kein Zugriff auf Inhalte möglich.

2 LZW Algorithmus

Dieser Abschnitt wiederholt kurz den LZW Algorithmus. Die Implementierungsaufgabe folgt in Abschnitt 3.

Der der Aufgabe zugrundeliegende LZW-Algorithmus ist anhand der Beschreibung im Skriptum und den Erläuterungen in der Vorlesung zu implementieren. Bevor Sie mit der Implementierung beginnen erstellen Sie bitte mit Papier und Bleistift den Encodierungs- und Decodierungsvorgang für die Zeichenfolge `"abababababab"` wie im Skript, allerdings nun mit dem größeren Alphabet aus 256 Zeichen. Während der *Encodierung* erzeugen Sie dabei den Dictionary des Encoders. Während der *Decodierung* erzeugen Sie, völlig unabhängig vom Dictionary des Encoders, den Dictionary des Decoders. Danach erstellen Sie ein Flussdiagramm für Ihr Programm. Dann wird das Flussdiagramm in Programmcode umgesetzt. In der Implementierung überprüfen Sie dann mittels des Debuggers Schritt für Schritt, ob Ihre Implementierung sich genauso verhält, wie Ihre Lösung mit Papier und Bleistift und Ihr Flussdiagramm es fordern.

Die eigentliche Implementierungsaufgabe ist in Abschnitt 3 beschrieben. Im folgenden werden die Grundzüge des Algorithmus noch einmal sehr kurz und ohne Beispiel erklärt, eine ausführlichere Beschreibung mit Beispielen befindet sich im Skript.

Intention Aufgabe des LZW-Algorithmus ist es, lange Zeichenketten zu komprimieren um Speicherplatz und Übertragungsbandbreite zu sparen. Ziel ist es dabei, möglichst lange Zeichenfolgen durch einen Indexwert zu repräsentieren. Dies gelingt umso besser, je öfter identische Zeichenfolgen im Eingangsdatenstrom auftreten (Redundanz).

Dictionary Grundlage des Algorithmus ist ein Dictionary, welcher z. B. mittels Hashing als Array oder als Baum effizient implementiert werden kann. Im Praktikum werden Sie zwei Varianten implementieren. Zunächst werden Sie ein Array ohne Hashing verwenden, wobei jedes Arrayelement eine Zeichenfolge repräsentiert. Dies ist eine zwar ineffiziente, aber dafür schlanke Variante, anhand derer man leichter das Grundprinzip des LZW Algorithmus versteht. Dann werden Sie zum Ersatz der sequentiellen Speicherung einen Trie erzeugen und mittels Hashing in einem Array speichern.

Der Dictionary wird stets individuell für jeden Datensatz (jedes File, Dokument, etc.) im Sender und identisch dazu auch im Empfänger neu erzeugt und ist nicht Bestandteil der fertig kodierten Zeichenkette. Vor Beginn der Codierung wird das gesamte verwendete Alphabet in den Dictionaries von Sender und Empfänger gespeichert. Jeder Eintrag des Dictionaries wird durch seinen Index identifiziert. Als Resultat der Codierung werden Indexwerte ausgegeben, also gespeichert oder gesendet. Die Datenkompression beruht darauf, dass möglichst lange Zeichenketten durch jeweils einen Indexwert repräsentiert werden.

Array-Implementierung Im ersten Teil des Praktikums werden Sie den Dictionary als Array implementieren. Beide Dictionaries werden als Array realisiert und sind initial in identischer Weise mit dem Alphabet gefüllt, in unserem Fall also in den ersten 256 Einträgen mit den Zeichen des ASCII-Alphabets[1] als Stringobjekten gefüllt. An Stelle 65 des Dictionaries finden Sie also ein String-Objekt mit Inhalt `"A"`, der zugehörige Indexwert ist 65.

Nun wiederholt sich der folgende Vorgang bis alle Zeichen codiert sind.

Sie lesen Zeichen für Zeichen (char by char) aus dem zu codierenden Datensatz ein und suchen nach jedem Zeichen die gesamte bisher eingelesene Zeichenfolge im Dictionary. Ist die Zeichenfolge schon vorhanden, so lesen Sie ein weiteres Zeichen ein, verlängern die Zeichenfolge um das neue Zeichen und suchen nun diese verlängerte Zeichenfolge im Dictionary. Dies führen Sie exakt so lange fort, bis Sie eine Zeichenfolge erreicht haben, die im Gegensatz zu allen vorherigen Versuchen nicht mehr im Dictionary vorhanden ist. Nun folgen zwei Schritte. Erstens wird die erreichte Zeichenkette dem Dictionary als neuer Eintrag hinzugefügt. Zweitens wird der zugehörige Index für die um das letzte Zeichen verminderte Zeichenkette, die also noch im Dictionary gefunden wurde, gesendet. Das letzte Zeichen der neu gespeicherten Zeichenfolge, also das erste nicht mehr übertragene Zeichen, wird darüber hinaus zum ersten und zunächst einzigen Zeichen der nächsten Zeichenfolge und der Vorgang beginnt von vorn.

Der *Decoder* erzeugt aus dem empfangenen Datenstrom ebenfalls denselben Dictionary. Der Decoder erhält den Dictionary *nicht* vom Encoder, vielmehr erzeugt der Decoder den Dictionary eigenständig aus den empfangenen Symbolen des Encoders. Der Dictionary ist dabei stets identisch zum Dictionary des Encoders, hinkt allerdings stets um einen Eintrag hinterher. Der Decoder hinkt hinterher, weil ja jeweils das letzte Zeichen des neuesten Encoder-Eintrags dem Decoder erst einen Schritt später bekannt wird. Der gesendete Index gehört nämlich zu dem String des neuesten Encoder-Eintrags, allerdings vermindert um das letzte Zeichen. Dieses letzte Zeichen ist aber stets auch das erste Zeichen des nächsten encodierten Strings. Somit wird dieses Zeichen dem Decoder im nächsten Schritt bekannt und der Decoder kann daher im nächsten Schritt den Eintrag in seine Tabelle vornehmen, hinkt also um einen Schritt hinterher.

Das Array ist initial wiederum mit dem Alphabet, in unserem Fall also in den ersten 256 Einträgen mit den numerischen Zeichencodes des ASCII-Alphabets gefüllt, Indexwerte 0 bis 255.

Nach Initialisierung der Decodertabelle mit dem Alphabet beginnt die Decodierung. Zunächst prüft der Decoder, ob der Vector mit den empfangenen Indizes

[1] siehe ASCII-Tabelle im Anhang

überhaupt Werte enthält. Ist der Vector leer, so wird die Decodierung mit leerem Ausgabestring beendet.

Ansonsten wird eine Schleife gestartet, die einen Indexwert nach dem anderen einliest, decodiert und den Dictionary erweitert. Für jeden empfangenen Indexwert erfolgen zwei Schritte. Erstens wird stets ein **neuer Eintrag** des Dictionaries gebildet mit der Zeichenfolge zum vorletzten empfangenen Index, verlängert um das erste Zeichen der Zeichenfolge zum letzten bisher empfangenen Index. Der neue Eintrag wird in den Speicherplatz mit dem kleinsten noch freien Indexwert gespeichert. Diese Speicherung hat der *Encoder* bereits einen Schritt vorher vorgenommen, der *Decoder* hinkt also um einen Schritt dem Encoder hinterher. Zweitens wird die zum letzten empfangenen Index zugehörige Zeichenfolge aus dem Dictionary ausgelesen, in den **Ausgabedatenstrom** des Decoders gesendet und zusätzlich noch für den nächsten Decodierungsschritt gespeichert.

Nun zum Sonderfall. Der Decoder hinkt beim Aufbau seiner Tabelle um einen Schritt hinterher. Somit kann der Sonderfall eintreten, dass der empfangene Indexwert dem aktuell erst noch zu speichernden Indexwert des Decoders entspricht. Dieser Index ist im Encoder bereits unmittelbar vorher Bestandteil des Dictionaries geworden und kann daher verwendet werden. Der Decoder findet aber diesen Indexwert noch nicht in seinem Dictionary. Weder kann der empfangene Indexwert direkt decodiert noch das fehlende erste Zeichen des neuen Strings zur Ergänzung der letzten Stelle des Strings für den neuen Dictionary Eintrags ermittelt werden. Zum Glück gibt es in diesem Spezialfall Abhilfe, denn das fehlende letzte Zeichen ist in diesem Fall identisch zum ersten Zeichen desselben Strings. Die beiden im Encoder zuletzt neu abgespeicherten Strings sind in diesem Fall nämlich fast identisch, lediglich der neueste Eintrag ist um ein Zeichen länger. Der erste dieser beiden Strings soll im Decoder gerade abgespeichert werden und sein letztes Zeichen fehlt noch. Der zweite dieser beiden Strings wird ohnehin erst später gespeichert.

Lange Rede, kurzer Sinn. Entspricht der zu decodierende Index dem nächsten zu speichernden Index, steht also noch nicht für die Decodierung im Dictionary, so besteht der zu speichernde String aus dem zuletzt decodierten String, verlängert um sein *eigenes* erstes Zeichen. Damit ergeben sich sowohl die Ausgabe als auch die aktuell erst noch zu speichernde Zeichenfolge aus dem (an der Position des letzten Zeichens noch unvollständigen) vorletzten Eintrag zuzüglich Kopie seines erstes Zeichens an seine letzte Position.

Trie-Implementierung Der zweite Teil des Praktikums erfordert die Implementierung des Dictionaries als Baum. Eine spezielle Baumstruktur, die besonders für längere Texte geeignet ist, wird als *Trie* (von Re*trie*val) bezeichnet. Die ersten 256 Knoten werden wiederum mit den Werten der ASCII-Tabelle gefüllt und die Wer-

te der Elternknoten auf -1 gesetzt. An Stelle 65 des Dictionaries findet sich also ein Knoten mit `parent = -1; index = 65; symbol = 'A'`. Das erste Zeichen wird ohne Aktion eingelesen. Das zweite Zeichen wird eingelesen und sein numerischer ASCII-Wert zusammen mit der Adresse (hier identisch zum numerischen ASCII-Wert) des ersten eingelesenen Zeichens der Hash-Funktion übergeben.

- Dieser Hashwert markiert die Position, an der ein neuer Knoten angelegt werden soll, der erstens als Symbol den Wert des zuletzt eingelesen Zeichens enthält und zweitens als Parent die Position des vorherigen eingelesenen Zeichens speziell für diese Zeichenfolge.
- Ist an der Stelle schon ein Knoten vorhanden, der aber nicht die erwarteten Daten aufweist (anderer Parent oder anderes Symbol), so ist beim Hashen eine Kollision passiert und es muss neu sondiert werden (Rehashing). Dies geschieht hier mit *doppeltem Hashing*. Die Rehashing-Funktion gibt eine andere mögliche Position für den Knoten zurück. Das Rehashen wird so lange wiederholt, bis man auf einen leeren Knoten oder auf einen Knoten mit den korrekten Einträge trifft.
- Ist an der erreichten Position der Dictionary frei, so wird ein Knoten mit den gewünschten Daten neu angelegt und die Position des Elternknotens im Ausgabedatenstrom ausgegeben.
- Ist das Rehashing beendet und der erreichte Knoten nicht frei, so sind hier bereits die korrekten Daten vorhanden, es hat also diese Zeichenfolge schon vorher gegeben und es wird daher nun versucht, diese Folge zu verlängern. Dazu wird ein weiteres Zeichen eingelesen und sein ASCII-Wert mit dem Positionswert des erreichten Knotens gehashed, der Vorgang beginnt also (ggf. mit Rehashing) von vorn.
- Wird beim Versuch, ein neues Zeichen einzulesen, das Ende des Datenstroms erreicht, so wird zum Abschluss die Adresse des letzten erreichten, korrekte Daten enthaltenden Knotens dem Ausgabedatenstrom übergeben.

3 Aufgabe

- Sie haben in Abschnitt 1.1 dieser Praktikumsbeschreibung bereits ein Projekt mit Namen *Praktikum* (als Kopie von Blatt2) angelegt und das Verzeichnis *src* mit den *.cpp und *.hpp Files aus Ihrem persönlichen SVN Repository als lokale Kopie (Working Copy) gefüllt.

- Sie haben in Abschnitt 2 die Lösung für den String `abababababab` und ein Flussdiagramm für Ihre Implementierung bereits in Papierversion erstellt.
- Verwenden Sie das SVN Repository, um nach jedem gelösten Teilproblem der Implementierung Ihre Ergebnisse mit *Commit* zu sichern. Vergessen Sie nicht die Kommentare beim Commit. Sichern Sie möglichst solche Zwischenschritte, die keine Fehlermeldungen beim Build oder beim Ausführen erzeugen. Im Team wären sonst Ihre Kollegen mit fehlerhaftem Code konfrontiert.
- Wenn Sie einmal nicht weiterkommen, nutzen Sie die CIP-Übungen und stellen Sie Ihre Fragen im Stud.IP.

3.1 Projekt für das Praktikum vorbereiten

Im File mainPraktikum.cpp können Sie nacheinander alle Gruppen von Tests bis auf einen Test Ihrer Wahl auskommentieren um effizienter und effektiver mit dem Debugger arbeiten zu können. Sie können auch die gesamte Methode `int main()` im File mainPraktikum.cpp zunächst vollständig auskommentieren und durch eine eigene Version ohne google Test für Ihr Debugging ersetzen. Für unsere Ergebnisüberprüfung werden wir allerdings Ihr mainPraktikum.cpp gegen die Originalversion austauschen, also muss Ihr Programm am Ende mit der Originalversion von mainPraktikum.cpp fehlerfrei in Teil 1 und in Teil 2 durchlaufen.

Sehen Sie sich im Editor das File mainPraktikum.cpp an. Im unteren Bereich finden Sie alle für den Test verwendeten Strings als Argument von `testPhrasesList` und dazu jeweils die erwarteten, korrekten Indexwerte des Encoders als Argumente der Befehle `resultList.push_back()` und `resultTrie.push_back()`. Damit können Sie im Debugger die korrekte Funktionsweise leicht überprüfen (und natürlich auch Ihre Bleistift/Papier Version). Beachten Sie, dass auch der leere String getestet wird.

Nun *kopieren* Sie alle *.cpp und *.hpp Files der folgenden Liste aus Ihren Lösungen der Übungsaufgaben 1 bis 7 in das Unterverzeichnis Praktikum/src des Projekts (Project Explorer oder Windows Explorer). Wählen Sie bei CCounter nicht die Version aus Aufgabe 7 mit dem Beobachtermuster, sondern die davor erstellte Version aus Übung 5. Natürlich könnten Sie auch die Version aus Aufgabe 7 verwenden, Sie müssten dann aber zusätzlich die Files mit den Definitionen zu CCounterObserver einbinden.

Headerfiles	**Quellfiles**
`CArray.hpp`	
`CCounter.hpp (Übg 5)`	`CCounter.cpp (Übg 5)`
`CForwardCounter.hpp`	`CForwardCounter.cpp`
`CDoubleHashing.hpp`	`CDoubleHashing.cpp`
`CEntry.hpp`	`CEntry.cpp`
`CKnot.hpp`	`CKnot.cpp`
`XOutOfBounds.hpp`	`XOutOfBounds.cpp`
Schon vorhanden:	Schon vorhanden:
	`mainPraktikum.cpp`
`CLZW.hpp`	`CLZW.cpp`
`CEnc.hpp`	`CEnc.cpp`
`CDec.hpp`	`CDec.cpp`

3.2 Implementierung als Array

Nun geht es an die eigentliche Programmierarbeit. Im ersten Teil des Praktikums sollen Sie die Datenstruktur Dictionary des LZW-Algorithmus als Array implementieren, denn wir benötigen z.B. die Operation Delete eines Dictionary nicht. Dazu erstellen Sie zwei neue Klassen (beide mit hpp- und Cpp-Datei) mit den Namen `CArrayEnc` für den Encoder und `CArrayDec` für den Decoder. `CArrayEnc` erbt von `CEnc` und `CArrayDec` erbt von `CDec`. Beide Basisklassen `CEnc` und `CDec` sind abstrakt und erben ihrerseits von `CLZW`, wobei diese drei Klassen bereits vorgegeben sind. Die Header für Ihre Klassen müssen Sie komplett selbst erstellen. Im Folgenden werden Ihnen dazu Hinweise gegeben. Denken Sie unbedingt an die Kommentare (siehe Section 1.4 auf Seite 11).

- `m_symbolTable`
  Sie benötigen ein Array als private Membervariable. Wie Sie dieses benennen ist letzten Endes egal, wir haben es `m_symbolTable` genannt. Als Datentyp des Arrays verwenden wir das Klassentemplate `CArray`, das Sie schon in einer Übung erstellt haben. Dieses Template erfordert Datentyp und Größe. Wir verwenden als Datentyp `CEntry` und als Größe `LZW_DICT_SIZE`. Diese Konstante ist im CLZW-Header festgelegt. Eine Festlegung dieser Konstanten als statische Variable oder als globale Variable scheitert daran, dass deren Initialisierung für die Templateklasse CArray zu spät erfolgen könnte und daher nicht ohne Tricks funktioniert.

- `CArrayEnc()` bzw. `CArrayDec()`
  Natürlich benötigen Sie diese Konstruktoren. Im Konstruktor werden die ersten 256 Einträge des Dictionaries mit den Strings zu den Positionen 0-255 der ASCII-Tabelle belegt. Um einen **`int`**-Wert einem `string` zuzuweisen, können Sie in Klassen, die von `CLZW` abgeleitet sind, die statische Methode `string intToString(int i)` verwenden. Zum Beispiel wird der Wert 3 folgendermaßen der Position 3 des Dictionaries zugewiesen:
  `m_symbolTable[3].setSymbol(intToString(3));`
  Das Gegenstück ist die Funktion **`unsigned int`** `charToInt(char);`, ebenfalls eine statische Funktion der Klasse `CLZW`. Diese sollten Sie verwenden, damit auch Umlaute und andere Zeichen mit ASCII-Werten oberhalb 127 korrekt in den Zahlenbereich 128 bis 255 gewandelt werden.

- **`int`** `searchInTable(const string &)`
  Diese Funktion sucht im Array nach dem String, der ihr als Parameter übergeben wird, und gibt bei erfolgreicher Suche die Position des Strings zurück. Wurde der String nicht gefunden, so gibt die Funktion -1 zurück. Tipp: Machen Sie sich die Sache einfach, und suchen Sie, unabhängig vom Füllungsgrad des Arrays, immer das *ganze* Array ab.

- `vector<unsigned int> encode(const string &)`
  Hier sind Sie am Kernstück der Array-Implementierung angelangt. Diese Methode wird in der Klasse `CArrayEnc` definiert. Natürlich haben Sie bei der Programmierung alle Freiheiten, dennoch sind hier einige Hinweise gegeben, wie wir diese Aufgabe gelöst haben.

  Wenn die Eingangszeichenfolge leer ist, dann beenden Sie die Encodierung. Ansonsten durchlaufen Sie iterativ die folgenden Schritte.

  Sie benötigen zwei Strings, einen String S für die alte, bisher eingelesene Zeichenkette, sowie einen String Sx für die alte Zeichenkette nach Erweiterung um ein neu eingelesenes Zeichen x. Außerdem benötigen Sie einen Ausgabevektor `vector<unsigned int>`, in den Sie die Indexwerte des Codierungsresultats schreiben können. Nun befolgen Sie die Anweisungen des Algorithmus.

  Sie müssen Zeichen für Zeichen einlesen und jeweils die unten beschriebenen Schritte durchführen. Das zeichenweise Einlesen eines Strings sollten Sie mit dem `string::const_iterator` realisieren. Machen Sie sich damit vertraut, wie man den Iterator benutzt.

  1. Bestehende Zeichenkette S (zu Beginn leer) mit neu eingelesenem Zeichen x in Zeichenkette Sx zusammenfügen. Prüfen, ob die so entstandene neue Zeichenkette schon im Dictionary vorhanden ist.

2. Wenn ja: Sx an S zuweisen, neues Zeichen x einlesen. Falls nicht End-Of-File, dann weiter bei Schritt 1, sonst weiter bei 4.
3. Wenn nein: Ausgabe des Indexwerts der alten Zeichenkette S im Dictionary, also Hinzufügen der Positionsnummer zum Ausgabevektor. Hinzufügen der neuen Zeichenkette Sx *an die nächste freie Stelle* ins Array. Variable S der alten Zeichenkette mit dem zuletzt eingelesenen Zeichen x überschreiben. (Last-First-Property: Das letzte Zeichen der vorherigen Zeichenfolge ist stets das erste Zeichen der neuen Zeichenfolge.) S enthält nun nur das Zeichen x. Dann das nächste neue Zeichen x einlesen. Falls nicht End-Of-File, dann weiter bei Schritt 1.
4. Der Algorithmus erfordert, dass Sie ganz zum Schluss, sobald Sie das Fileende erreicht haben, noch die Position der letzten alten Zeichenkette S ausgeben (also den Index in den Ausgabevektor schreiben).

- `string decode(const vector<unsigned int>&)`
  Diese Methode wird in der Klasse `CArrayDec` definiert. Sie lesen Index für Index aus dem Vektor ein, geben den zum Index zugehörigen String des Arrays aus und wenn dies nicht ein einzelnes Zeichen des Alphabets ist, dann tragen Sie eine neue Zeile in den LZW-Dictionary, d.h. ein neues Element in das Array ein.

  Spätestens an dieser Stelle zahlt sich aus, wenn Sie inzwischen das Verfahren im Skript einschließlich des Beispiels noch einmal gründlich nachvollzogen und das Verfahren mit Papier und Bleistift einschließlich Flussdiagramm aufgeschrieben haben, für den Decoder ist dies besonders zu empfehlen. Überprüfen Sie nun genau dieses geplante Verhalten mittels Debugger bei Ihrer Implementierung.

  Ganz wichtig, der Decoder besitzt seine eigene Tabelle, völlig unabhängig vom Encoder. Der Decoder baut seine Tabelle, abgesehen von der Initialisierung, nur anhand der empfangenen Indexwerte vom Typ unsigned int auf, die Tabelle des Encoders wird *nicht* explizit übertragen. Wie es dem Decoder gelingt, seine Tabelle aufzubauen, ist im Skriptum der Vorlesung beschrieben. Zu Beginn wird die Tabelle mit dem Alphabet initialisiert, exakt identisch zum Encoder. Der Aufbau der Tabelle des Decoders hinkt der Tabelle des Encoders immer um einen Schritt hinterher, denn es fehlt dem Decoder stets das letzte Symbol des vom Encoder zuletzt gespeicherten Strings. Dieses Symbol wird im nächsten Schritt ermittelt, denn es ist identisch zum ersten Symbol des Strings der folgenden Übertragung.

  Der Ablauf umfasst grob die folgenden Schritte.

  - Initialisierung des Dictionaries mit dem Alphabet.
  - Ausgabe leerer String `""` falls der Eingabevektor leer ist.
  - Ersten Index einlesen und zugehörigen String ausgeben und für den nächsten Schritt zusätzlich speichern.

  - Nun folgt eine Schleife über alle weiteren Indexwerte der Eingangsfolge.
    * Falls der Indexwert noch nicht im Dictionary eingetragen ist, (nur) dann den zuvor decodierten String um sein erstes Zeichen verlängern, ansonsten zum Indexwert den String aus dem Dictionary auslesen.
    * Resultat an den Ausgabestring anhängen
    * Vorherigen String um erstes Zeichen des aktuellen Strings (d.h des Resultats) verlängern und an nächste freie Position im Dictionary speichern. Übrigens muss, falls der empfangene Index noch nicht vorhangen war, der empfangene Index gleich dem Index dieser verwendeten nächsten freien Position sein.
    * Resultat für den nächsten Schritt speichern, dies ist im nächsten Schritt der "vorherige String".
    * Falls keine freien Stellen im Dictionary mehr vorhanden sind, so sollte der Algorithmus eine Fehlermeldung ausgeben und terminieren.

Herzlichen Glückwunsch, die erste große Hürde ist gemeistert. Testen Sie Ihren Code mit den Unit-Tests, die von uns bereitgestellt wurden. Verwenden Sie den Debugger, um eventuelle Fehler zu finden und die berechneten Datenwerte zu kontrollieren.

3.3 Implementierung des Encoders und Decoders als Trie

Wieder erstellen Sie eine neue Klasse, diesmal mit Namen `CTrieEnc` für den Encoder und `CTrieDec` für den Decoder. Testen Sie Ihren Code mit den Unit-Tests. Ändern Sie dafür zuerst in der Datei *mainPraktikum.cpp* die Zeile
`#define` `TEIL_1` in **`#define`** `TEIL_2` um.

Den Header müssen Sie neu schreiben, er weist aber Ähnlichkeit mit dem Header der vorherigen Aufgabe auf. Die Funktion `searchInTable` entfällt. Es werden im Folgenden die Funktionen und ihre Aufgaben erläutert, speziell die Dinge, die sich gegenüber der Array-Implementierung geändert haben:

- `m_symbolTable`
  Der Datentyp der Dictionary-Elemente des Tries ist `CKnot`.

- `CTrie()`
  Im Konstruktor werden erneut die ersten 256 Elemente initialisiert. Da die ersten 256 Elemente Wurzelelemente sind und daher keine Elternknoten haben, setzen Sie dieses Feld auf -1. Zur Erkennung der noch unbelegten Knoten soll bei diesen das Attribut `m_parent` mit dem Wert -2 initialisiert werden. Die Unit-Tests des Praktikums prüfen auch, ob Sie das umgestellt haben.

- `vector<unsigned int> encode(const string &in)`
  Hier ist der LZW-Algorithmus implementiert. Für das Hashing und auch für Rehashing benutzen Sie bitte die Funktion **`unsigned int`** `hash(`**`unsigned int , unsigned int , unsigned int, unsigned int`**`)`
  aus Ihrer Klasse `CDoubleHashing` aus der Übungsaufgabe 7.

Sie benötigen die beiden Integer `I` und `J`, für die Elternposition `I` und für den numerischen ASCII-Wert `J` des neu eingelesenen Zeichens. Das erste Argument wird für die Parent-Adresse, das zweite Argument für den numerischen ASCII-Wert verwendet.

Außerdem benötigen Sie einen `CForwardCounter attemptCounter` für die Anzahl der Rehashingversuche. Zudem erstellen Sie wieder einen geeigneten Ausgabevektor. Das erste Zeichen lesen Sie blind ein, außer der Eingabestring ist leer. Danach bedienen Sie sich wieder des Iteratorkonzeptes, um nach und nach die zu codierende Zeichenkette einzulesen. In jedem Iterationsschritt müssen folgende Aufgaben erledigt werden:

- Neues Zeichen in Variable J einlesen. Hashwert von Position alter Zeichenkette I mit numerischem ASCII-Wert des neuen Zeichens J ermitteln. Der Hashwert markiert die Position, an der der Knoten abgelegt werden soll.

- Ist die ermittelte Position mit anderen Werten als I und J belegt, dann ist eine Kollision aufgetreten und Sie müssen so oft rehashen, bis Sie

  – auf einen leeren Knoten treffen, oder

  – auf einen Knoten treffen, der die Werte von I und J bereits enthält, oder

  – beim ursprünglichen Hashwert wieder ankommen, also keine neuen Adressen mehr zu erwarten sind.

  Zum Rehashing $h_i(x)$ rufen Sie nochmals die Hashfunktion mit den gleichen Werten von I und J sowie inkrementiertem Wert von `attemptCounter` auf. *Achtung*: Der ursprüngliche Hashwert $h_0(x) = h(x)$ bleibt bei Rehashing unverändert, Sie benötigen ihn um eine Endlosschleife zu vermeiden. Überlegen Sie sich ein geeignetes Schleifenkonstrukt, das diese Schritte so lange ausführt, bis eine geeignete Position im Dictionary gefunden wurde.

- Wenn die Zeichenkette noch nicht vorhanden ist, prüfen Sie, ob Sie auf einen freien Knoten gestoßen sind. Dies erkennen Sie daran, dass der Index des Knotens -2 ist oder dass der gespeicherte String leer ist. Ist dies der Fall, dann tragen Sie die Elternposition I und den String mit dem Zeichen zum Zahlenwert J in den Knoten ein. Der Wert von I wird darüber hinaus in den Ausgabevektor geschrieben. Zum Schluss überschreiben Sie I mit dem Wert von J und lesen ein neues Zeichen in J ein.

- Wenn der Knoten an der errechneten Position die korrekten Werte von I und J bereits enthält, dann ist die geprüfte Zeichenkette bis zum aktuellen Zeichen J schon im Dictionary gespeichert. Es gilt im Dictionary an der ermittelten Position also: Index == J; Parent == I. Die gerade ermittelte Position wird dann in I gespeichert und ein neues Zeichen J wird eingelesen.

- Sobald beim Lesen von J das Ende des Eingabestrings erreicht wird, ist als Abschluss die Position I des letzten Teilstrings auszugeben.

- Nun implementieren und dokumentieren Sie noch den passenden Decoder, dann ist es geschafft.

3.4 Lösung einreichen und finaler Test

Wichtig: Zum Abschluss führen Sie nun folgende Testschritte durch.

- Überprüfen Sie, dass alle Ihre Quellfiles mit Commit im SVN gespeichert sind, also alle Quellfiles mit Add dem SVN hinzugefügt wurden.

- Ganz zu Beginn haben Sie einige Files im SVN vorgefunden (mainPraktikum.cpp, CLZW.*, CEnc.* CDec.*). Speichern Sie nun diese Originalversionen in Ihrem Verzeichnis src der lokalen Kopie, ersetzen Sie also Ihre Versionen durch die Originalversionen. Diese Versionen haben Sie sich entweder als Originalfiles zurückgelegt oder finden Sie mit Tortoise unter Update to Revision... , wählen Sie für die Files einfach die älteste Version.

- Testen Sie beide Teile (`#define TEIL_1` und `#define TEIL_2`) auf fehlerfreie Funktion.

- Und nun testen Sie, wie wir es hinterher ebenfalls machen. Legen Sie in genau der verteilten portableDevEnv Version unter Windows entlang der Beschreibung in Abschnitt 1 ein Projekt PraktikumCheck an (als Kopie von Blatt2) und legen Sie dort mit Checkout eine frische lokale Kopie Ihrer SVN Head-Revision im Verzeichnis PraktikumCheck/src an. Testen Sie beide Teile (`#define TEIL_1` und `#define TEIL_2`) auf fehlerfreie Funktion. Nun sind Sie sicher, dass keine überraschenden Seiteneffekte existieren und auch unser Test positiv verlaufen wird.

Warum dieser abschließende Test? Wir akzeptieren keine individuellen Beschwerden, falls unser Test fehlschlägt. Dies sind die Klassiker für Probleme:

- Es waren in mainPraktikum.cpp einzelne Tests auskommentiert und daher gar nicht geprüft worden.

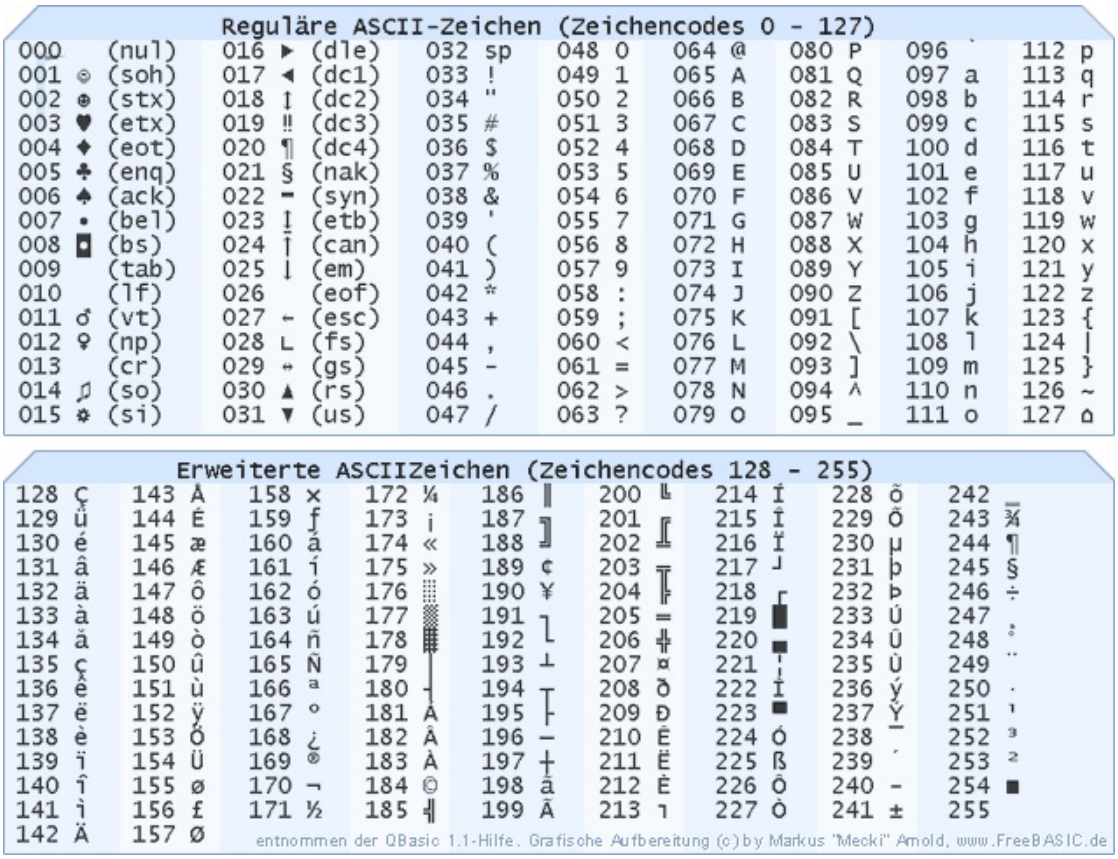
- Das Projektfile enthielt Pfade in zusätzliche Directories mit Quellfiles, die bei uns dann fehlten und zu Linkerfehlern führten.
- Es wurde nicht die allerletzte Version hochgeladen.
- Einzelne zusätzliche Zeichen (z.B. öffnende Klammer) führen zu Compilerfehlern, obwohl das Programm im Prinzip korrekt funktioniert.
- Teil 2 funktionierte korrekt, aber Teil 1 nicht mehr. Es war also der Test mit Teil 1 am Ende nicht wiederholt worden.

4 Anhang

4.1 C++ Literaturhinweise

- Thinking in C++ 2nd Edition by Bruce Eckel (z. B. Vol. 2, Kap. 2: *Iostreams*)
  `http://mindview.net/Books/TICPP/ThinkingInCPP2e.html`
- The C++ Resources Network (z. B. Tutorial *Input/Output with files*)
  `http://www.cplusplus.com/`
- `http://www.learncpp.com/`

4.2 ASCII Tabelle



Reguläre ASCII-Zeichen (Zeichencodes 0 - 127)

000	(nul)	016 ▶	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001 ☺	(soh)	017 ◀	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002 ☻	(stx)	018 ↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003 ♥	(etx)	019 ‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004 ♦	(eot)	020 ¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005 ♣	(enq)	021 §	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006 ♠	(ack)	022 ▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007 •	(bel)	023 ↨	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008 ◘	(bs)	024 ↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025 ↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011 ♂	(vt)	027 ←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012 ♀	(np)	028 ∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013	(cr)	029 ↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014 ♫	(so)	030 ▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015 ☼	(si)	031 ▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Erweiterte ASCIIZeichen (Zeichencodes 128 - 255)

128	Ç	143	Å	158	×	172	¼	186	‖	200	╚	214	Í	228	õ	242	‗
129	ü	144	É	159	ƒ	173	¡	187	╗	201	╔	215	Î	229	Õ	243	¾
130	é	145	æ	160	á	174	«	188	╝	202	╩	216	Ï	230	µ	244	¶
131	â	146	Æ	161	í	175	»	189	¢	203	╦	217	┘	231	þ	245	§
132	ä	147	ô	162	ó	176	░	190	¥	204	╠	218	┌	232	Þ	246	÷
133	à	148	ö	163	ú	177	▒	191	┐	205	═	219	█	233	Ú	247	‚
134	å	149	ò	164	ñ	178	▓	192	└	206	╬	220	▄	234	Û	248	°
135	ç	150	û	165	Ñ	179	│	193	┴	207	¤	221	¦	235	Ù	249	¨
136	ê	151	ù	166	ª	180	┤	194	┬	208	ð	222	Ì	236	ý	250	·
137	ë	152	ÿ	167	º	181	Á	195	├	209	Ð	223	▀	237	Ý	251	¹
138	è	153	Ö	168	¿	182	Â	196	─	210	Ê	224	Ó	238	¯	252	³
139	ï	154	Ü	169	®	183	À	197	┼	211	Ë	225	ß	239	´	253	²
140	î	155	ø	170	¬	184	©	198	ã	212	È	226	Ô	240	-	254	■
141	ì	156	£	171	½	185	╣	199	Ã	213	ı	227	Ò	241	±	255	
142	Ä	157	Ø														

entnommen der QBasic 1.1-Hilfe. Grafische Aufbereitung (c) by Markus "Mecki" Arnold, www.FreeBASIC.de

Abbildung 1: Erweiterte ASCII Tabelle