# Diplomarbeit

*„Experimentelle Untersuchung von String B-Trees bezüglich ihrer Anwendbarkeit in Genom-Datenbanken und im Information Retrieval"*

Manuel Scholz

Betreuer: Prof. Dr. H. Schweppe,

Prof. Dr. J. Stoye, Z. Kanaeva

Institut für Informatik

Freie Universität Berlin

mscholz@inf.fu-berlin.de

*15.04.2002*

# Inhaltsverzeichnis

1. Einleitung............................................................4

1.1 Motivation............................................................4

1.2 Zielsetzung...........................................................5

2. Die String B-Tree Datenstruktur............................7

2.1 Definitionen..........................................................7

2.2 Das Grundmodell des String B-Trees......................8

2.3 Patricia Tries ........................................................9

2.4 Das erweiterte Modell des String B-Trees.............13

2.5 Operationen auf String B-Trees ...........................15

2.6 Zusammenfassung................................................17

3. Implementierung..................................................18

3.1 Ziel der Implementierung......................................18

3.2 Verwendete Algorithmen......................................19

3.3 Der Prototyp: „sbtree“..........................................19

3.4 Wichtige Aspekte der Implementierung.................20

3.4.1 Erstellen eines eindeutigen Endmarkers.................20

3.4.2 Binärcodierung der Worte...................................21

3.4.3 Splitting-Operation ............................................22

3.4.4 Caching.............................................................24

3.4.5 Hintergrundspeicherzugriffe.................................25

4. Planung der Untersuchungen................................26

4.1 Einflussgrößen.....................................................27

4.2 Messgrößen.........................................................28

4.3 Operationen.........................................................31

4.4 Untersuchungen am B-Baum................................31

5. Experimentelle Untersuchungen...........................33

5.1 Vorbemerkungen..................................................33

5.1.1 Zeitaufwand der Experimente...............................33

5.1.2 Methodik der Untersuchungen..............................33

5.2 Überprüfung der theoretischen Aussagen .............34

5.3 Technische Untersuchungen.................................37
5.3.1 Auswirkungen der Caches....................................37
5.3.2 Verhältnisse der Messgrößen................................40
5.3.3 Optimaler Knotengrad..........................................41
5.4 Ein- und Mehrworttextuntersuchungen.................43
5.4.1 Auswirkungen der Suchmusterlänge......................47
5.5 Untersuchungen der Alphabetgröße ......................48
5.6 Untersuchungen verschiedener Texte ....................51
5.6.1 Vorbemerkungen .................................................52
5.6.2 Untersuchungsergebnisse und Analyse...................53
5.7 Vergleich mit der Suffix Tree Datenstruktur..........58
5.7.1 Vorbemerkungen..................................................58
5.7.2 Suffix Trees (wotd) .............................................59
5.7.3 Untersuchungsergebnisse und Analyse...................60
6. Zusammenfassung und Ausblick...........................67
6.1 Ausblick ..............................................................69
6.2 Abschlussdiskussion.............................................70
7. Anhang................................................................72
7.1 Literatur..............................................................72

# 1. Einleitung

## *1.1 Motivation*

Schon in den Anfängen des Computerzeitalters wurde die Eigenschaft von Computern, große Datenmengen effizient zu verwalten, erkannt und vielseitig genutzt. Im Laufe der Jahrzehnte und besonders in den letzten Jahren hat die computergestützte Datenverwaltung stetig und so stark zugenommen, dass sie heutzutage praktisch in allen Lebensbereichen Verwendung findet. Nicht nur viele der großen Firmen besitzen Datenbanken riesigen Ausmaßes, auch in der Forschung und der Informationsverwaltung im Internet gibt es eine unüberschaubare Menge an digitalen Informationen.

Zwei in diesem Zusammenhang besonders aktuelle Gebiete sind die Genforschung und das Information Retrieval. Durch die in den letzten Jahren stark vorangeschrittene Sequenzierung einer Vielzahl von unterschiedlichen Organismen existieren im Bereich der Gendatenbanken riesige Mengen an DNA-, Protein- und Beschreibungsdaten, welche ständig aktualisiert und durchsucht werden müssen. Auch im Information Retrieval gibt es viele Bereiche (Zeitungsarchive, Wörterbücher, Enzyklopädien, Telefonbücher, usw.), die gerade durch das Internet mit einer täglich wachsenden Zahl von Informationen konfrontiert werden.

Trotz des stetigen Technologiefortschrittes, welcher schnellere und größere Speicher umfassende Computer hervorbringt, ist eine effiziente Verwaltung der Daten aktueller als je zuvor. Aufgrund der immensen Größe der zu verwaltenden Datenmengen reichen selbst die Hauptspeichergrößen von großen Computern oft nicht aus, um die Daten komplett zu beinhalten. Da eine Änderung dieses Verhältnisses auch in Zukunft nicht abzusehen ist, gibt es besonders auf dem Gebiet der persistenten Indexstrukturen für große Textcorpora einen großen Bedarf an effizienten Algorithmen und Datenstrukturen.

Die bisher in diesem Bereich häufig diskutierten Datenstrukturen sind Suffix Trees [13,16] und Suffix Arrays [12], zwei typische Datenstrukturen, die für die Mustersuche in großen Textcorpora konzipiert sind und einige Variationen der klassischen B-Baum Struktur, wie z.B. der Präfix B-Baum [2], welcher zum Teil auch Verwendung in Datenbanksystemen findet.

In der Arbeit "*The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications*" von P. Ferragina und R. Grossi (1999) wird eine neuartige persistente Indexstruktur, der String B-Tree[1] vorgestellt. Aufgrund seiner Struktur, die eine Verbindung von B-Bäumen und Patricia-Tries [10] ist, besitzt der String B-Tree vielversprechende Eigenschaften. Im Gegensatz zum Suffix Tree kann der String B-Tree durch seine B-Baum Struktur nicht entarten. Er hat somit das gleiche Worst Case

1 Im Folgenden wird immer die englische Form des Namens verwendet, da eine Teilübersetzung nicht sinnvoll erscheint.

Verhalten wie ein B-Baum, kann dabei jedoch Strings mit beliebiger Länge verwalten, was mit einem B-Baum nicht möglich ist. Der String B-Tree bietet sich somit gut für die Mustersuche in großen Textcorpora an.

Trotz dieser vielversprechenden Eigenschaften der String B-Tree Datenstruktur gibt es bisher nur zwei unveröffentlichte Arbeiten [5,14], die experimentelle Untersuchungen an Implementierungen der String B-Tree Datenstruktur durchgeführt haben.

Die erste Arbeit ist eine experimentelle Untersuchung von den Autoren selbst [5]. Sie führt einen Vergleich zwischen den Implementierungen von String B-Trees, Suffix Arrays und Präfix B-Bäumen durch. Dabei wird nur die Anzahl der Festplattenzugriffe betrachtet, Ausführungszeiten werden dabei nicht berücksichtigt.

Die Arbeit von K. R. Rose [14] betrachtet den Bereich der Datenbanken und vergleicht die String B-Tree Implementierung mit dem Datenbanksystem *Sleepycat*.

Es ist jedoch bisher keine Studie einer String B-Tree Implementierung bezüglich ihrer Anwendung in Genom-Datenbanken und im Information Retrieval bekannt. Im Rahmen dieser Arbeit soll diese Studie anhand experimenteller Untersuchungen der String B-Tree Datenstruktur durchgeführt werden.

## 1.2 Zielsetzung

Ziel der Diplomarbeit ist die experimentelle Untersuchung der String B-Tree Datenstruktur speziell im Hinblick auf die Bereiche Genom-Datenbanken und Information Retrieval. Diese Untersuchungen sollen an einer selbsterstellten Implementierung des String B-Trees, die auf den in [6,5] erläuterten Algorithmen basiert, durchgeführt werden. Als Eingabedaten erhält die Implementierung beliebige Texte, aus denen sie einen String B-Tree konstruiert, der die Gesamtheit aller Suffixe der Worte des Textes enthält. Da die exakte Substringsuche in den zu untersuchenden Anwendungsbereichen eine große Rolle spielt, liegt der Schwerpunkt dieser Arbeit auf der Betrachtung dieser Suchoperation. Die Ausgabe der gefundenen Elemente wird dabei nicht berücksichtigt, sondern nur überprüft, ob der gesuchte Substring im Text enthaltenen ist oder nicht.

Anhand dieser Suchoperation sollen die Eigenschaften der String B-Tree Datenstruktur experimentell untersucht werden. Dabei werden die Auswirkungen der Einflussgrößen der Datenstruktur selbst (Baumgröße, Knotengröße) als auch die Einflussgrößen der verwendeten Textdaten (Alphabetgröße, Struktur des Textes) untersucht. Weiterhin wird ein erster Vergleich mit der Suffix Tree Datenstruktur durchgeführt, die bereits bei der Mustersuche in großen Textcorpora in der Praxis Anwendung findet.

In erster Linie werden in den Untersuchungen die Ergebnisse der Gesamtzeiten betrachtet. Die Messungen der Festplattenzugriffe und der CPU-Zeit werden dabei nur als Hilfsgrößen für die Analyse der Ergebnisse verwendet.

Basierend auf den Ergebnissen der experimentellen Untersuchungen soll eine Abschätzung über die Eigenschaften des String B-Trees in der praktischen Anwendung und dessen Eignung für die Anwendung in den Bereichen Genom-Datenbanken und Information Retrieval getroffen werden.

# 2. Die String B-Tree Datenstruktur

Der String B-Tree ist eine sehr aktuelle Datenstruktur, die speziell für den Umgang mit großen, homogenen Datenmengen, die sich im Hintergrundspeicher befinden, optimiert wurde. Insbesondere für das Speichern und Durchsuchen von großen Textdatenbanken (z.B. Zeitungsarchive, Gendatenbanken) sind die String B-Trees gut geeignet.

Die zugrunde liegende Idee dieser Struktur ist, einen $B^+$-Baum ähnlichen Aufbau mit einer sehr kompakten und für die Stringsuche optimierten Baumstruktur, dem Patricia Trie [10], zu verknüpfen. Dabei dient ersterer für die Anordnung der Knoten im Hintergrundspeicher und letztere für die Suche innerhalb der Knoten.

Der folgende Text soll eine Einführung in die Datenstruktur geben. Dabei werden zunächst die Randbedingungen und die Definition festgelegt, danach der grundlegende Aufbau und die Anordnung der Daten in den Knoten erklärt und abschließend die Funktion und das Laufzeitverhalten der Suchoperation analysiert.

## 2.1 Definitionen

Um die Notationen innerhalb der gesamten Arbeit konsistent zu halten, sollen an dieser Stelle die Rahmenbedingungen und die Definitionen der häufig verwendeten Größen gegeben werden:

- Als Speichermodell wird die klassische 2-Stufen-Hierarchie (wie in [4] vorgeschlagen) verwendet, die von einem begrenzten und schnellen Hauptspeicher und einem langsamen unbegrenzten Hintergrundspeicher ausgeht. Der Hintergrundspeicher ist in festgelegte Speicherbereiche (Seiten) unterteilt. Die Seitengröße wird im Folgenden mit $s$ bezeichnet.
- Sei $\Sigma$ eine endliche und geordnete Menge, das Alphabet. Die Größe von $\Sigma$ sei $m = |\Sigma|$ und $\Sigma^*$ sei die Menge aller Zeichenketten über $\Sigma$.
- Sei $Z$ eine endliche Zeichenkette (Eingabedaten), so dass gilt: $Z \in \Sigma^*$.
  Die Länge von $Z$ sei $z = |Z|$.
- Sei $Z[q,r]$ ein Substring (Teilzeichenkette) aus $Z$, von der Stelle $q$ bis zur Stelle $r$ mit $1 \leqslant q \leqslant r \leqslant z$ und $Z[q]$ ein Zeichen aus $Z$ an der Stelle $q$ mit $1 \leqslant q \leqslant z$.
- Sei $Z[q,u]$, mit $q \leqslant u \leqslant r$ ein Präfix,
  $Z[t,r]$, mit $q \leqslant t \leqslant r$ ein Suffix und
  $Z[t,u]$, mit $q \leqslant t \leqslant u \leqslant r$ ein Substring von $Z[q,r]$.
- Ein Suchmuster $P$ der Länge $p = |P|$ ist in $Z$ enthalten, wenn gilt:
  $\exists\, i \in \mathbb{N}_{\setminus\{0\}}\colon Z[i, i+|P|-1] = P$.
- Die Zeichenkette Z enthält an $n$ Stellen, mit $n \in \mathbb{N}_{\setminus\{0\}}$ ein bestimmtes

Endmarkersymbol, welches die Zeichenkette in $n$ einzelne Worte unterteilt, so dass für jedes Wort $w = Z[x,y]$ gilt: $(Z[x-1]=\$ \vee x=1) \wedge (Z[y+1]=\$) \wedge (\$ \notin Z[x,y])$

- $g$ sei der Knotengrad der String B-Tree Knoten und $k_v$ der Füllungsgrad (Anzahl der Elemente im Knoten) eines Knotens $v$.

## 2.2 Das Grundmodell des String B-Trees

Wie im Namen der Datenstruktur bereits zu erkennen ist, hat der String B-Tree von seiner Grundstruktur eine große Ähnlichkeit mit den B-Bäumen, genauer gesagt mit $B^+$-Bäumen. Wie der $B^+$-Baum speichert er mehrere Elemente[2] in einem Knoten, welchen er bei einem Zugriff jedesmal aus dem Hintergrundspeicher lädt. Der Grad $g$ eines String B-Trees gibt an, wie viele Elemente in einem Knoten enthalten sein können. Dabei gilt im gesamten Baum folgende Invarianz: $g-1 \leqslant e \leqslant (2 \cdot g)-1$ (*mit* $e$ *= Anzahl der Elemente im Knoten*). Die Elemente im Knoten sind der Größe nach geordnet und entsprechend dieser Ordnung gibt es mehrere (abhängig vom Füllungsgrad des Knotens) Verweise auf Kinderknoten. Die Elemente in den inneren Knoten dienen (wie beim $B^+$-Baum) nur zur korrekten Anordnung der Blätter, welche dann die eigentlichen Elemente (bzw. Referenzen auf jene) enthalten. Ordnet man alle Elemente der Blätter entsprechend derer (lexikographischen) Ordnung an, so erhält man eine geordnete Liste aller Worte des Textes. Dies ist ein enormer Vorteil bei der Präfixsuche, auf den später noch genauer eingegangen wird.

Ein Unterschied des vereinfachten Modells des String B-Trees zum $B^+$-Baum besteht nun darin, dass die Elemente im Knoten nicht die Worte selbst, sondern nur Referenzen (Pointer) auf den Anfang der Worte sind. Die Anordnung der Pointer richtet sich entsprechend nach der Ordnung der zugrunde liegenden Worte. Um alle diese Eigenschaften besser zu verdeutlichen folgt eine Beispielabbildung (Abbildung 1) und ihre Erklärung.

Der untere Bildrand zeigt die Daten, die in einer Datei liegen und in diesem Fall mit den Zahlen 1 bis 70 indiziert sind. Das dunkle Kästchen nach jedem Wort ist der bereits erwähnte Endmarker. Im oberen Bereich sieht man eine vereinfachte Form des String B-Trees. Die Zahlen in den Knoten geben die Verweise auf die Datei an. Bei einem Verweis muss nur auf den Anfang des Wortes gezeigt werden, da das Ende durch den Endmarker angegeben ist. Die Elemente in den Knoten sind entsprechend ihren Verweisen geordnet, so dass z.B. der Inhalt des mittleren rechten Knotens (38, 23) die Ordnung der Worte *klein* und *pyramide* darstellt.

2 In der gesamten Arbeit sind damit ausschließlich Strings gemeint.

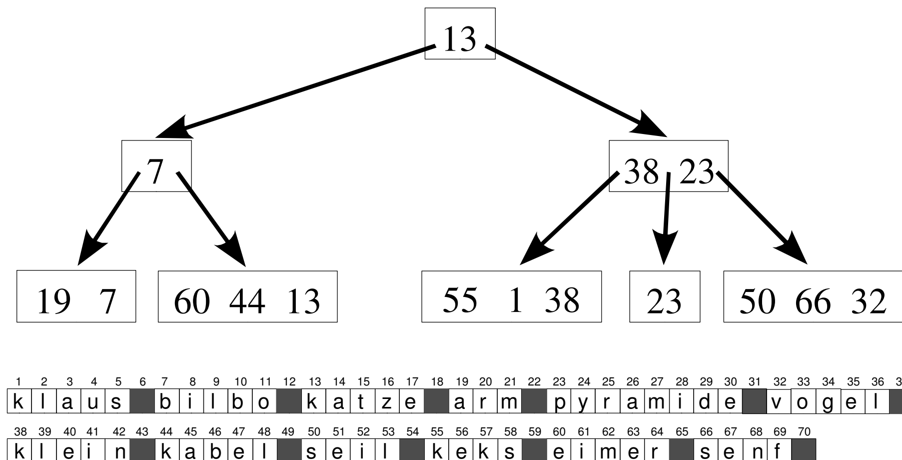


*Abbildung 1: Vereinfachte Darstellung eines String B-Trees*

Die Verweise auf die Kinderknoten unterliegen der gleichen Ordnung, so dass das Element rechts von einem Verweis jeweils eine Obergrenze für alle Elemente des darunter liegenden Teilbaumes darstellt. Geht man im Beispiel von der linken Kante des obersten Knotens aus, so weiß man, dass alle untergeordneten Elemente auf Worte verweisen, welche lexikographisch kleiner als *katze* sind.

Wie weiter oben schon erwähnt befindet sich die Gesamtheit aller Worte eines String B-Trees geordnet in seinen Blättern, so dass bei diesem Beispiel die fünf Blätter von links nach rechts traversiert, die Verweise auf alle Worte in geordneter Reihenfolge (von *arm* bis *vogel*) enthalten.

Wie gesagt ist das momentan verwendete Modell nur eine vereinfachte Version des String B-Trees und in dieser Form natürlich nicht brauchbar, da es durch die Verweise in den Knoten zu viel Hintergrundspeicherzugriffe benötigen würde.

### *Bemerkung*

Das hier beschriebene (und in dieser Form allgemein bekannte) $B^+$-Baum Modell weicht von dem in [6] vorgeschlagenen Modell leicht ab. Dieses besitzt für jeden Verweis auf einen Kinderknoten nicht nur ein zugehöriges Element (als Obergrenze), sondern jeweils ein Paar von Elementen, das die Ober- und die Untergrenze der Elemente des darunter liegenden Teilbaumes repräsentiert. Der dort beschriebene Vorteil der geringeren Komplexität war nicht nachvollziehbar, was zu dem Entschluss führte, das allgemein übliche Modell zu verwenden.

## 2.3 Patricia Tries

Die Hauptidee, welche die String B-Trees mit ihren Eigenschaften so interessant werden lässt, ist die Einbindung von Patricia Tries in deren Knoten. Warum das so vorteilhaft ist,

und wodurch dies erreicht wird, soll in diesem Abschnitt erklärt werden.
Patricia Tries sind eine bezüglich Speicherplatzeffizienz optimierte Art der sogenannten Tries (trie von Re**trie**val). Aufgrund des verwandten Aufbaus werden zuerst zwei ähnliche (aber einfachere) Trie-Strukturen erläutert und darauf basierend der Patricia Trie erklärt.

### *Tries*

Tries wurden als Datenstruktur entwickelt, um in ihnen grosse Textmengen (bestehend aus vielen einzelnen Strings) zu speichern, nach denen dann im Trie möglichst optimal gesucht werden soll. Um dies zu erreichen, machen sich die Tries die gemeinsamen Präfixe der Strings zunutze. Sucht man also nach einem Wort "abc", so betrachtet man dabei anfangs auch alle Worte, welche einen gleichen Präfix (wie z.B. "ab") besitzen. Damit solche unnötigen Betrachtungen vermieden werden, speichert der Trie diese Informationen nur einmal.

Zusammengefasst lautet die Definition dieser Baumstruktur wie folgt: Ein Trie ist eine m-äre Baumstruktur, welche die in ihr gespeicherten Strings von der Wurzel an buchstabenweise in den Kanten speichert. Somit ergibt die Konkatenation der Buchstaben aller Kanten von jedem Weg der Wurzel zu einem Blatt die im Trie enthaltene Wortmenge. Am besten lassen sich all die genannten Eigenschaften an einem Beispiel beschreiben, das in Abbildung 2 gegeben ist.

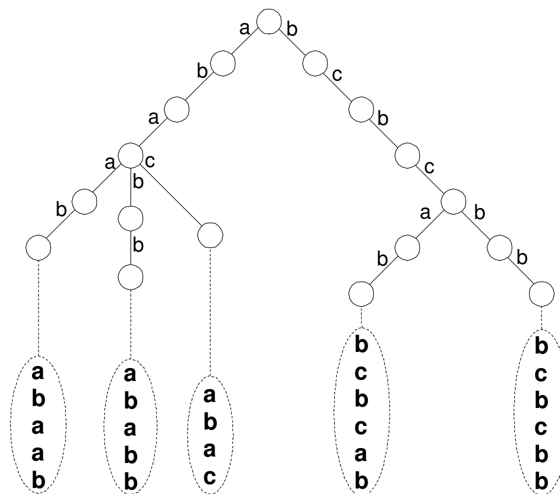

*Abbildung 2: Einfacher Trie*

Der in diesem Beispiel gezeigte Trie beinhaltet folgende Strings: "abaab, ababb, abac, bcbcbb, bcbcab". Startet man nun an der Wurzel und folgt den Kanten bis zu einem Blatt, so läuft man eines der fünf enthaltenen Worte ab. Wie oben bereits erklärt, sieht man hier, dass die gemeinsamen Präfixe der Strings nur einmal im Baum vorkommen, was Platz und somit (bei der Suche) auch Zeit spart. Erst bei voneinander abweichenden Strings

verzweigt der Trie.

### *Compacted Tries*

Bei dem bisherigen Modell wurde pro Kante immer nur ein Buchstabe (Symbol) gespeichert, was für grosse Textmengen ein erheblicher Platzverbrauch ist. Der Compacted Trie verringert diesen deutlich, in dem er die im Baum enthaltenen Ketten „zusammenzieht". Durch die Tatsache, dass bei einer Kette von Kanten immer nur eine Möglichkeit der Traversierung besteht, ist es möglich die Informationen aller Kanten dieses Weges auf einer Kante zusammenzufassen. Daraus folgt, dass die Kanten auch mehrere Buchstaben beinhalten können. Das Resultat ist der sogenannte Compacted Trie; es folgt ein Beispielbaum in Abbildung 3, welcher die gleichen Strings wie der bereits gezeigte Trie enthält.

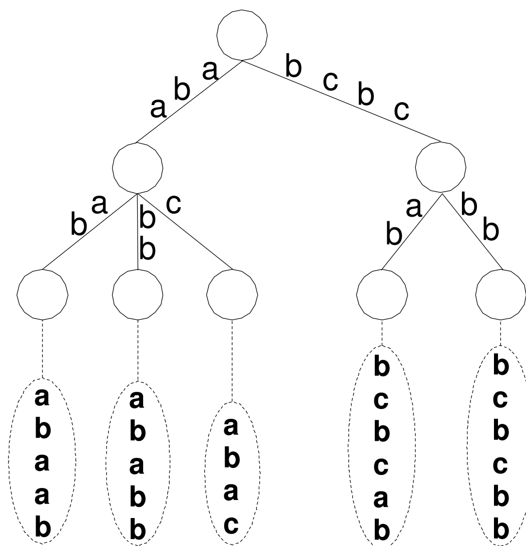


*Abbildung 3: Beispiel für einen Compacted Trie*

### *Patricia Tries*

Der letzte Schritt der Komprimierung (der auf dem Compacted Trie aufbaut) wird in den Patricia Tries durchgeführt und ist in diesem Fall verlustbehaftet, was bedeutet, dass ein Teil der ursprünglichen Informationen verloren geht. Später wird jedoch gezeigt, dass trotz der geringen Informationsmenge weiterhin eine eindeutige Suche in diesem Trie möglich ist.

Der Unterschied zwischen Compacted und Patricia Tries besteht darin, dass beim letzteren nur noch der erste Buchstabe jeder Kante erhalten bleibt, nämlich genau der Buchstabe, der für die Verzweigung der Kanten verantwortlich ist. Somit besitzt ein Patricia Trie zwar genauso viele Kanten wie ein Compacted Trie, er enthält aber pro Kante nur einen Buchstaben (den sogenannten *Branching Character*).

Im Gegensatz zu den bisher vorgestellten Tries muss der Patricia Trie jedoch zusätzlich die aktuelle Länge des Teilstrings in jedem Knoten speichern, wodurch der Patricia Trie insgesamt trotzdem platzeffizienter als der Compacted Trie ist. Speziell beim Patricia Trie im String B-Tree gibt es den Unterschied, dass in jedem Blatt nicht das Wort selbst, sondern nur eine Referenz darauf enthalten ist. Es folgt Abbildung 4, die noch einmal die Unterschiede zwischen den beiden Tries deutlich macht.

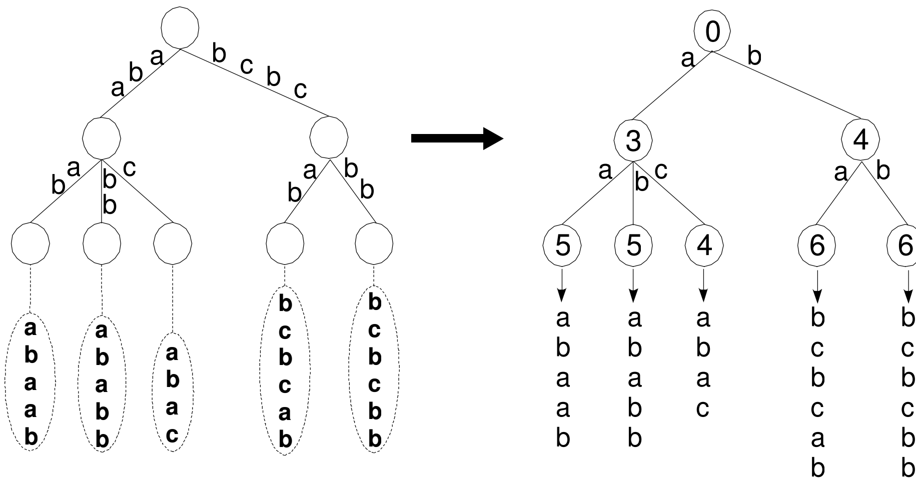


*Abbildung 4: Unterschied zwischen einem Compacted und einem Patricia Trie*

### *Suche in Patricia Tries*

Aufgrund der sehr kompakten und verlustbehafteten Speicherung der Strings im Patricia Trie stellt sich die Frage, wie man in dieser Datenstruktur möglichst effizient suchen kann. Dabei geht es um die Positionssuche des Suchmusters im Patricia Trie, da dies für die Suche im String B-Tree notwendig ist. Darauf soll aber erst in den Folgeabschnitten genauer eingegangen werden. Diese etwas kompliziertere Suche auf einem Patricia Trie lässt sich grundsätzlich in zwei Phasen unterteilen:

Die erste Phase der Suche (*Blind Search*) führt nur einen ungenauen Vergleich durch, dessen Ziel es ist, ein Wort mit dem *longest common prefix* (*lcp-Wort*) zu finden. Dazu wird das Suchmuster nur mit den Branching Characters im Patricia Trie verglichen. Je nach Vergleich wird dann die entsprechende Kante gewählt und der Baum so lange traversiert, bis ein Blatt bzw. das Ende des Suchmusters erreicht ist. Endet die Suche in einem Blatt, so muss das Wort im Blatt den *lcp* besitzen. Dabei ist zu beachten, dass benachbarte Worte (welche den gleichen Vaterknoten besitzen) ebenfalls den gleichen *lcp* besitzen können, wenn das Suchmuster z.B. schon eine Ebene höher von allen Folgeelementen verschieden ist.

Diese Tatsache macht man sich zunutze, wenn die Suche mitten im Baum endet (wenn das Suchmuster zu kurz ist). Man weiß nun, dass alle darunter liegenden Kinder einen

gemeinsamen *lcp* besitzen. Folglich wird aus den darunter liegenden Worten ein beliebiges ausgewählt. Ein Beispiel für eine im Blatt endende Suche ist in Abbildung 5 auf der linken Seite gegeben.

In der zweiten Phase (Verifikation) wird das gefundene Wort nun benutzt, um die korrekte Position des Suchmusters zu bestimmen. Dazu werden beide Worte verglichen und die erste Stelle bestimmt, an der die beiden Worte nicht mehr übereinstimmen (Fehlstelle). Mit Hilfe des Buchstabens an der Fehlstelle bestimmt man den entsprechenden Nachfolgeknoten. Von diesem aus kann nun die korrekte Position des Suchstrings (ebenfalls durch Vergleich mit dem abweichenden Buchstaben) bestimmt werden. In der folgenden Beispielabbildung (Abbildung 5, rechts) gibt es nur einen Nachfolgeknoten. Die korrekte Position des Suchmusters ist vor dem kleinsten Nachfolgeblatt, da der Buchstabe an der Fehlstelle kleiner ist als alle anderen.

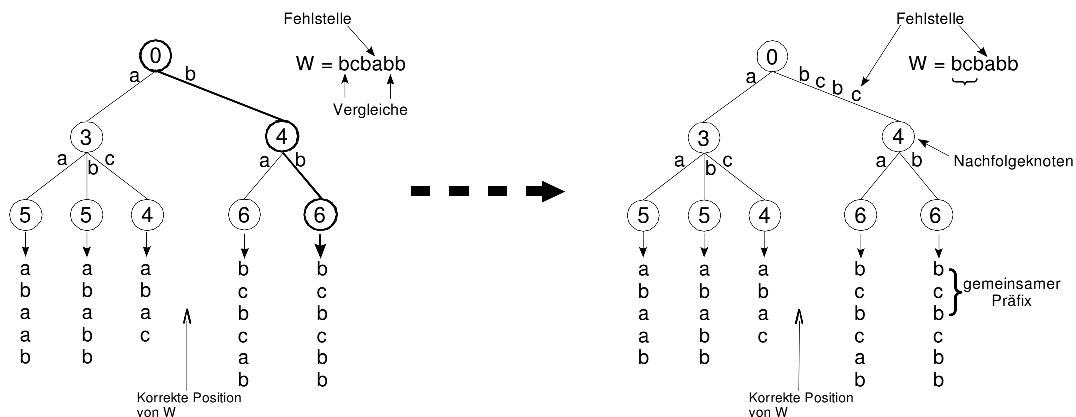


*Abbildung 5: Suche in einem Patricia Trie (1. und 2. Phase)*

## 2.4 Das erweiterte Modell des String B-Trees

Nachdem in den bisherigen Abschnitten auf die einzelnen Teile des String B-Trees eingegangen wurde, soll nun deren Zusammenwirken, also der String B-Tree als ganzes betrachtet werden. Dabei wird nicht nur dessen Funktion, sondern auch dessen Komplexität analysiert.

Betrachtet man die in Abschnitt 2.2 eingeführte $B^+$-Baum Struktur, so sieht man schnell, dass die Effizienz (wie beim B-Baum) erheblich von der Größe der Knoten bzw. deren Grad abhängt. Je mehr Elemente nun in einen Knoten passen, desto größer ist der Verzweigungsgrad des Baumes und desto geringer seine Höhe und damit auch die Zahl der Zugriffe auf den Hintergrundspeicher. Durch die große Zahl an Elementen innerhalb des Knotens werden jedoch im Knoten mehr Vergleiche und somit auch mehr Hintergrundspeicherzugriffe benötigt. Das liegt daran, dass im Knoten nicht die eigentlichen Elemente, sondern nur deren Verweise stehen, so muss bei jedem Vergleich im Knoten auf den Hintergrundspeicher zugegriffen werden. Selbst wenn man im Knoten

nicht sequentiell ($O(k)$ Zugriffe[3]), sondern mit Hilfe eines binären Baumes sucht, so beträgt die Zahl der Hintergrundspeicherzugriffe immer noch $O(\log k)$. Um in der B-Baum Struktur von der Knotengröße zu profitieren, ohne dabei zu viele Zugriffe im Knoten selbst zu haben, braucht man eine von der Größe unabhängige Verwaltung der Elemente im Knoten.

Diese Anforderung erfüllt der Patricia Trie, da er sehr kompakt aufgebaut ist und bei einer Suche nur ***einen*** Zugriff auf den Hintergrundspeicher (Textdatei) benötigt.

Dies ist leicht am Suchalgorithmus zu erkennen: Das Ergebnis der ersten Suchphase ist das *lcp-Wort*, welches in der zweiten Phase mit dem Suchmuster verglichen wird. Dieser Zugriff ist der einzige in der Suche, der benötigt wird, um die korrekte Position des Suchmusters zu bestimmen. Die Vergleichsbuchstaben (Branching Characters) auf den Kanten sind alle im Patricia Trie selbst enthalten.

Durch die Verknüpfung von $B^+$-Baum Struktur und Patricia Tries (für die Verwaltung der Eemente in den Knoten) ergeben sich für den String B-Tree pro Suchanfrage $O(p/s+\log_g n)+1$ Zugriffe auf den Hintergrundspeicher. Dabei ist zu beachten, dass durch die kompakte Datenhaltung in den Patricia Tries sehr viele Elemente in einem String B-Tree Knoten gespeichert werden können und somit auch *g* im allgemeinen relativ groß ist. Abschließend folgt die Abbildung (Nr. 6) eines String B-Tree Modells, welches noch einmal die Verknüpfung von $B^+$-Baum Struktur und Patricia Trie verdeutlicht. Im Gegensatz zum ersten Modell sind nicht nur die Worte selbst, sondern auch deren Suffixe in den Baum eingefügt worden. Dies ist notwendig für eine exakte Substringsuche, die im nächsten Abschnitt im Detail erklärt wird. Die Patricia Tries in den Knoten der Abbildung sind nur symbolhaft und stimmen nicht mit der tatsächlichen Struktur überein.

Nachdem gezeigt wurde, wie der String B-Tree aufgebaut ist, und welche Eigenschaften sich daraus ergeben, soll abschließend auf die Operationen des String B-Trees eingegangen werden.

3 Wenn es nicht anders angegeben wird, so ist im Folgenden die O-Notation im Text immer auf die Zahl der Hintergrundspeicherzugriffe und nicht auf die der Rechenoperationen bezogen.

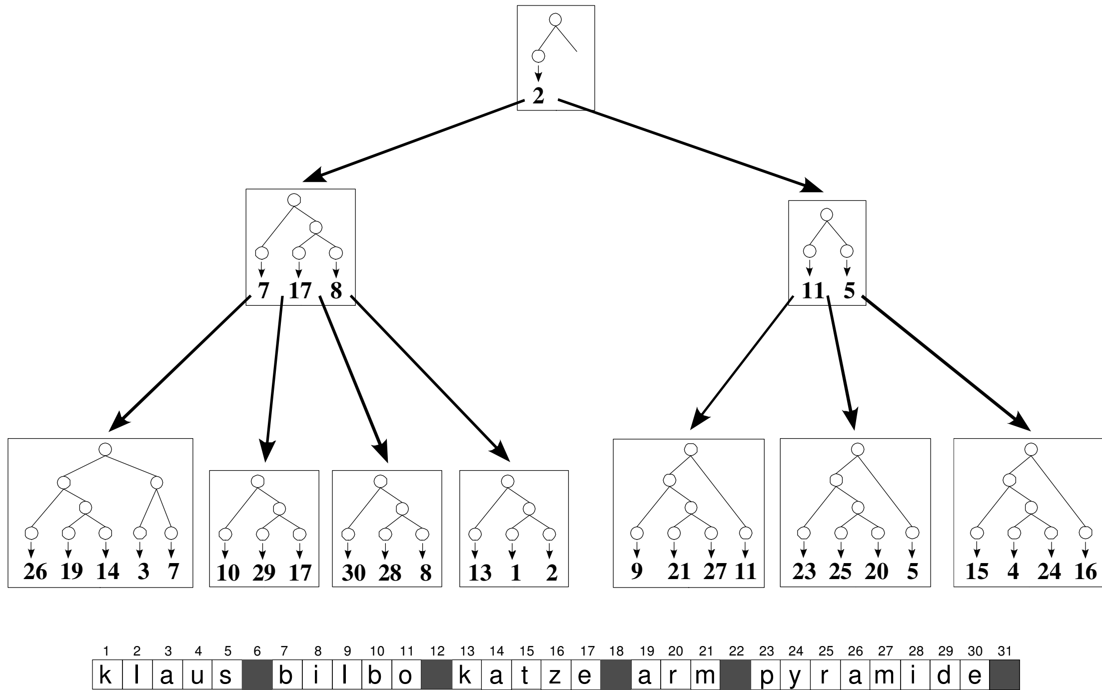


*Abbildung 6: String B-Tree (detaillierte Darstellung)*

## 2.5 Operationen auf String B-Trees

In diesem Abschnitt werden nur die für diese Arbeit relevanten grundlegenden Operationen (Suchen und Einfügen) auf String B-Trees behandelt. Andere und komplexere Operationen wie z.B. das Löschen oder die Ähnlichkeitssuche wurden weggelassen, da sie für die experimentellen Untersuchungen nicht relevant sind (vgl. Abschnitt 4.3).

### *Suche*

Die Suchoperation ähnelt wiederum stark der Suche in herkömmlichen B-Bäumen. Es wird beim Wurzelknoten begonnen und dann Schritt für Schritt die Baumstruktur traversiert. In jedem Knoten wird die Position des Suchelementes bestimmt und dementsprechend der nächste Kinderknoten gewählt bzw. die Suche beendet, wenn das Element bereits gefunden wurde. Als Rückgabe liefert die Suchoperation entweder den Fundknoten und die Fundposition in dem Knoten oder einen Wert, der den Misserfolg signalisiert.

### *Präfix- und Substringsuche*

Gerade in der Anwendung in Gendatenbanken und im Information Retrieval ist es oft

nicht von so großer Bedeutung, eine komplette Wortsuche durchzuführen, sondern vielmehr den Präfix eines Wortes bzw. einen Substring in einem Wort oder Satz wiederzufinden. Im Folgenden soll die Vorgehensweise dieser beiden Suchanfragen auf String B-Trees erklärt werden.

Ziel der Präfixsuche ist es, alle Worte eines Textes zu finden, die den gesuchten Präfix enthalten. Dabei ist es nicht notwendig, jedes einzelne Wort mit dem Präfix zu vergleichen, da man sich die (lexikographische) Ordnung der Worte zunutze machen kann. Ein echter Präfix (|Präfix| < |Wort|) eines Wortes ist entsprechend dieser Ordnung immer kleiner als das Wort selbst. Betrachtet man die lexikographisch aufsteigend geordnete Menge aller Worte des Suchtextes, so stellt man fest, dass sich ein eingeordneter Präfix von einem bzw. mehreren Wort(en) genau an der Position vor diesen befinden würde. So z.B. der Präfix "man" bei der geordneten Wortfolge: " ..., man, manchmal, Mandel, Mangel, ...".

Aufgrund dieser Tatsache ist für eine Präfixsuche nur die Positionsbestimmung innerhalb der geordneten Baumstruktur notwendig. Diese Suche entspricht genau der Verknüpfung der beiden Suchen auf $B^+$-Baum und Patricia Trie, die bereits beschrieben wurden. Hat man diese Position gefunden, so muss man den String B-Tree nur noch ab dieser Position in der Ordnungsreihenfolge so lange traversieren (und die Worte ausgeben), bis man auf ein Wort trifft, das nicht mehr den gesuchten Präfix besitzt. Wie schon in Abschnitt 2.2 beschrieben ist dies in der $B^+$-Baum Struktur sehr einfach und in linearer Zeit möglich.

Bei der exakten Substringsuche in einem String B-Tree greift genau der gleiche Suchalgorithmus wie bei der gerade beschriebenen Präfixsuche. Der eigentliche Unterschied zur Präfixsuche besteht in der zugrunde liegenden Menge der Strings. Für die Substringsuche wird die Menge der zu durchsuchenden Strings aus den Suffixen aller Worte des Textes gebildet, das heißt für jedes Wort im Text werden seine l (*mit* $l$ *= Wortlänge*) Suffixe der Menge der zu durchsuchenden Strings hinzugefügt.

Durch eine Präfixsuche auf diesen Suffixen ergibt sich leicht ersichtlich eine Substringsuche auf der Menge der ursprünglichen Worte. Dies hat zur Folge, dass in den String B-Trees häufig die Menge aller Suffixe der gegebenen Worte in den Baum eingefügt wird, da bei vielen Anwendungen nur eine Substringsuche relevant ist. Die in dieser Arbeit betrachtete Substringsuche gibt keine Ergebnisse zurück, sondern überprüft nur, ob der Substring enthalten ist oder nicht.

Im folgenden Teil der Arbeit soll mit der Suche immer die soeben beschriebene exakte Substringsuche, also die Präfixsuche auf der Menge aller Suffixe gemeint sein. Folglich sind mit den Elementen des String B-Trees fortan immer die Suffixe der Worte (bzw. des Wortes) der Eingabedaten (Texte) gemeint.

### *Einfügen*

Auch die Einfügeoperation der String B-Trees ähnelt, wie alle anderen bisher genannten Operationen und Eigenschaften, der entsprechenden Operation im B-Baum. Wenn die

Knoten, die bei der Suche auf dem Weg zur Einfügestelle traversiert werden, alle einen ausreichend kleinen Füllungsgrad haben, so wird das betreffende Element einfach in die Liste der Elemente des entsprechenden Knotens eingefügt. Wird auf dem Weg zur Einfügestelle ein Knoten traversiert, der bereits den maximalen Füllungsgrad besitzt, so wird dieser Knoten gesplittet. Durch dieses Splitten, das ebenfalls beim Wurzelknoten durchgeführt wird, ergibt sich eine ausbalancierte Baumstruktur. Da dies für B-Bäume bereits lange bekannt und ausführlich erklärt wurde (vgl. [3]), soll an dieser Stelle nicht genauer auf das Einfügen und das damit verbundene Splitten eingegangen werden.

## 2.6 Zusammenfassung

Abschließend sollen in diesem Abschnitt alle wichtigen Eigenschaften des String B-Trees und die daraus resultierenden Folgerungen angeführt werden. Dies ist eine Zusammenfassung der wichtigsten Aussagen aus [6], welche zum Verständnis angepasst wurden:

Der String B-Tree ist eine externe Datenstruktur, welche die Struktur eines $B^+$-Baumes mit Patricia Tries verknüpft, in dem es diese für die Verwaltung der Daten innerhalb der Knoten verwendet. Um die korrekte Position des Suchmusters im Knoten festzustellen, benötigt der Patricia Trie dabei nur einen Zugriff auf die Textdatei. Im gesamten String B-Tree werden nicht die eingefügten Worte selbst, sondern nur deren Verweise verwendet. Aus diesen Eigenschaften lassen sich folgende Aussagen ableiten:

- Durch die Verwendung von Verweisen auf die Worte und der kompakten Datenhaltung in den Patricia Trie ist es dem String B-Tree möglich, Worte beliebiger Länge zu verwalten.
- Durch die Verwendung der $B^+$-Baum Struktur sind String B-Trees eine ausgeglichene Baumstruktur und haben somit das gleiche Worst Case Verhalten wie B-Bäume.
- Die exakte Substringsuche benötigt im Worst Case $O(p/s + \log_g n)$ Hintergrundspeicherzugriffe.
- Das Einfügen eines Suffixes benötigt im Worst Case $O(q/s + \log_g n)$ Hintergrundspeicherzugriffe (*mit q = Suffixlänge*).

# 3. Implementierung

Nachdem im letzten Abschnitt die Struktur und die Operationen des String B-Trees erklärt wurden, soll in diesem Abschnitt auf die praktische Umsetzung (Implementierung) des Modells eingegangen werden. Dabei sollen verstärkt die Abweichungen und Besonderheiten der Implementierung im Vergleich zu dem in [6] vorgestellten Modell berücksichtigt werden.

### *Bereits vorhandene Implementierungen*

Zum jetzigen Zeitpunkt ist keine vollständige bzw. zugängliche Implementierung, sondern nur die Existenz dreier Prototypen bekannt. Diese Aussage beruht auf eigener Recherche und der Aussage von R. Grossi. Aus welchen Gründen keine Informationen zu diesen Prototypen veröffentlicht wurden und warum anscheinend nicht an ihnen weitergearbeitet wurde, ist mir nicht bekannt.

Es stellte sich also nicht die Frage, ob für die im Rahmen dieser Arbeit geplanten Untersuchungen auf eine bereits vorhandene Implementierung zurückgegriffen werden soll, sondern es ist von vornherein vorgesehen, eine eigene Implementierung der String B-Tree Struktur zu realisieren.

## *3.1 Ziel der Implementierung*

Wie bereits in der Zielsetzung der Diplomarbeit zu erkennen, ist das Hauptziel der Implementierung eine adäquate Umsetzung der String B-Trees für eine experimentelle Untersuchung deren Eigenschaften. Aufgrund der Tatsache, dass dies die erste zugängliche Implementierung ist, soll weiterhin auf eine erweiterbare und gut dokumentierte Programmierung geachtet werden. Die Möglichkeit, die vorhandene Implementierung für praktische Anwendungen zu erweitern, soll ebenfalls gegeben sein.

Durch diese Zielsetzungen gibt es jedoch auch eine klare Abgrenzung der Implementierung und einige Aspekte, die bewusst zurückgestellt wurden. Da bei den experimentellen Untersuchungen die Auswirkungen verschiedenster Einflussgrößen auf das Laufzeitverhalten des String B-Trees im Vordergrund stehen und der praktische Vergleich mit anderen Datenstrukturen nur exemplarisch betrachtet wird, ist in erster Linie das relative Verhältnis der Ergebnisse (und nicht deren absolute Werte) wichtig. Aus diesem Grund wurden aufwendige Optimierungen weitestgehend weggelassen. Ein weiteres Gegenargument ist eine durch die Optimierungen wesentlich schwerer verständliche Implementierung, was im Gegensatz zu den oben genannten Punkten (Erweiterbarkeit, Dokumentation) steht.

Als letzter Punkt ist festzuhalten, dass in der Implementierung nicht alle denkbaren Operationen der String B-Trees umgesetzt worden sind, sondern nur die für die Untersuchung notwendigen Punkte implementiert wurden. Es wurde jedoch darauf

geachtet, dass bei einer weiterführenden Arbeit diese Operationen ohne weiteres eingearbeitet werden können. Aufgrund dieser Eigenschaften ist diese Implementierung ebenfalls als ein Prototyp zu betrachten.

## 3.2 Verwendete Algorithmen

In diesem Abschnitt soll kurz auf die in der Implementierung verwendeten Algorithmen und die sich daraus ergebenen Probleme eingegangen werden.

Da die beiden Arbeiten von P. Ferragina und R. Grossi [6,5] die einzigen bekannten Quellen sind, welche den String B-Tree ausreichend erklären, basiert die Mehrheit der verwendeten Algorithmen darauf. Durch das hohe Abstraktionsniveau des Pseudocodes der Algorithmen gab es in vielen Bereichen eine Vielzahl von Implementierungsmöglichkeiten, welche in den meisten Fällen gründlich analysiert und ausgewertet werden mussten ehe eine Auswahl für den Prototyp getroffen werden konnte. Da die Algorithmen für die B-Baum Struktur nicht angegeben sind, wurden die B-Baum Algorithmen aus [4] verwendet.

Die Splittingoperation bei den Patricia Tries wurde ebenfalls nicht beschrieben. Es konnte auch keine Quelle gefunden werden, die einen Algorithmus für diese beschreibt, so dass ein eigener Algorithmus für das Splitting der Patricia Tries entwickelt wurde, welcher die in [6] beschriebenen Anforderungen erfüllt und später detaillierter erklärt wird.

Weitere Angaben zum Prototyp finden sich im Anhang oder unter *http://www.inf.fu-berlin.de/~mscholz/sbtree/*, wo auch der Quellcode des Prototyps zu finden ist.

## 3.3 Der Prototyp: „sbtree“

### Programmiersprache

Aufgrund der oben genannten Ziele der Implementierung wurde diese in der Programmiersprache C realisiert, die aufgrund ihrer Eigenschaften am besten dafür geeignet ist. Betrachtet man Implementierungen von anderen Datenstrukturen, die in der Praxis Anwendung finden, so sieht man, dass die große Mehrheit in C umgesetzt wurde. Das ist in erster Linie auf die hohe Ausführungsgeschwindigkeit von C zurückzuführen. Weiterhin ist C eine sehr systemnahe Sprache, was ebenfalls für die Implementierung von Datenstrukturen praktische Vorteile hat, da viele Zugriffe (Haupt- und Hintergrundspeicher) sehr systemnah programmiert werden sollten, um eine bestmögliche Nutzung der Systemressourcen zu erreichen.

Ein weiterer wichtiger Vorteil von C ist dessen Portierbarkeit, die gerade im Hinblick auf die Weiterverwendung meiner String B-Tree Implementierung ein wichtige Rolle spielt. Der Prototyp *„sbtree“* ist in ANSI C geschrieben und somit sowohl unter Solaris und Linux (gcc) als auch unter Windows (lcc) lauffähig.

### *Programmstruktur des Prototyps*

Das Programm ist in drei Module aufgeteilt:

- *sbtree*
  Das Hauptprogramm, das die Operationen (Einfügen und Suchen) des String B-Trees und eine Testumgebung enthält. Weiterhin beinhaltet es untergeordnete Operationen, wie z.B. Split-Node, welche zum Einfügen in den String B-Tree benötigt werden.
- *sbnode*
  Dieses Programm enthält den Datentyp und die Operationen für einen String B-Tree Knoten. Solch ein Knoten besteht aus einem Patricia Trie, welcher mit einem Feld, das die Nachfolgerliste enthält, verknüpft ist.
- *fileprocs, encwordprocs, stackprocs*
  Die Hilfs- und Hintergrundroutinen sind in drei Bereiche aufgeteilt. Ein Programmteil (*fileprocs*) enthält sämtliche Dateizugriffsroutinen, ein weiterer (*encwordprocs*) alle Wortoperationen (Wortlänge, Binärcodierung, usw.) und einer (*stackprocs*) die Implementierung eines Stacks.

Weitere Informationen zur genauen Struktur und zum Aufbau des Prototyps sind in der Programmdokumentation enthalten.

## *3.4 Wichtige Aspekte der Implementierung*

Auf eine Gesamtbeschreibung des Prototyps wird an dieser Stelle verzichtet. Jedoch sollen einzelne interessante Aspekte der Implementierung erwähnt werden, da sie zum Teil sehr hilfreich für das spätere Verständnis einiger Untersuchungsergebnisse sind.

### *3.4.1 Erstellen eines eindeutigen Endmarkers*

Wie P. Ferragina und R. Grossi in [5] vorschlagen, wird im Prototyp "*sbtree"* ein eindeutiger Endmarker hinter jedes Wort des Textes geschrieben, damit für jedes eingefügte Wort die Eindeutigkeit gewährleistet sein kann. Der Endmarker wird als Teil des Wortes behandelt. Um die Eindeutigkeit des Endmarkers zu gewährleisten, werden diese zusätzlich fortlaufend nummeriert. Dies geschieht durch eine Zahl, die sich hinter dem Endmarkersymbol befindet. Es folgt eine Beispielabbildung.

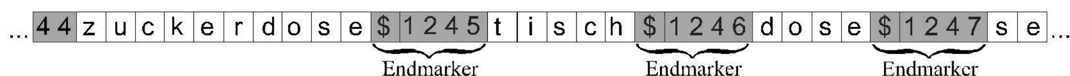


*Abbildung 7: Detaildarstellung des Endmarkers (mit '$' als Startsymbol des Endmarkes)*

Wie man in der Abbildung erkennen kann, würde es ohne den eindeutigen Endmarker beim einfügen dieser Daten zwei gleiche Strings, nämlich den String „dose“ geben, da er einmal als ganzes Wort und einmal als Suffix des Wortes „zuckerdose“ in den Baum eingefügt werden würde. Durch den Endmarker jedoch ergeben sich die beiden (ungleichen) Strings „dose$1245“ und „dose$1247“.

Das eigentliche Problem gleicher Worte und Präfixe von anderen Worten liegt in der Patricia Trie Struktur, welche dadurch komplexer (und somit auch weniger effizient) werden würde. Es kann dadurch nicht mehr gewährleistet werden, dass in jedem Blatt des Patricia Tries genau ein Element enthalten ist. Grundsätzlich ist die Implementierung einer solchen Variante möglich, diese wird jedoch aus den bereits genannten Gründen der schlechten Effizienz nicht in Betracht gezogen.

Betrachtet man die praktische Anwendung im Hinblick auf die Endmarker, so müssten die Eingabedaten noch einmal vorverarbeitet werden. Dies ist aber nur notwendig bei Texten mit mehreren Worten (z.B. Wörterbüchern), da Eingabedaten, die nur ein Wort enthalten, durch ihr Dateiende bereits mit einem eindeutigen Endmarker versehen sind.

### 3.4.2 Binärcodierung der Worte

Ein wichtiger Teil der String B-Tree Implementierung ist die Umsetzung der Patricia Tries. Da das bereits beschriebene theoretische Modell in jedem Knoten 0 bis $m$ ($m$ = *Alphabetgröße*) Kanten zu den Kinderknoten zulässt, müssen alle Kantenlisten entweder als weniger effiziente dynamische Struktur umgesetzt werden oder sie verbrauchen als statisches Feld wesentlich mehr Speicherplatz. Aus der unregelmäßigen Zahl der Kinderknoten ergeben sich zusätzlich Probleme bei der Abbildung auf den Hintergrundspeicher.

Um diese Probleme zu vermeiden, schlagen P. Ferragina und R. Grossi in [5] eine Binärcodierung der Eingabeworte vor. Die praktischen Vorteile, die sich daraus ergeben, sind sofort ersichtlich, wenn man das Beispiel in Abbildung 8 betrachtet.

Durch die Codierung hat der Trie keine m-äre, sondern eine binäre Struktur, die wesentlich einfacher und somit effizienter zu implementieren ist. Da jeder Knoten (mit Ausnahme der Wurzel) genau zwei Kinder besitzt, können diese einfach in einem zweielementigen Feld gespeichert werden, das einen wesentlich schnelleren Zugriff ermöglicht. Zusätzlich ist eine kompakte Abbildung auf den Hintergrundspeicher durch einen einfachen Preorder Walk zu realisieren.[4]

4 Die Abbildung auf den Hintergrundspeicher ist in [5] ausführlich beschrieben.

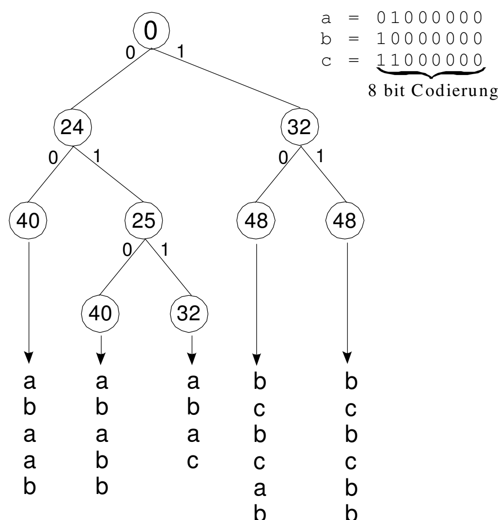


*Abbildung 8: Beispielabbildung eines binären Patricia Tries*

Ein weiterer in [5] angesprochener Vorteil ist die Unabhängigkeit vom Eingabealphabet. Im "*sbtree*" Prototyp sind durch eine 8 Bit Codierung Alphabetgrößen von 2-255 (Ein Zeichen ist für den Anfang des Endmarkers reserviert) möglich. Es wurde eine statische Codierungsgröße von 8 Bit verwendet, weil der Datenzugriff in Byterastern effizienter ist als der Zugriff auf einzelne Bits. Außerdem entsteht durch die längeren Worte im Trie kein Nachteil, da diese dann viele Stellen mit gleichen konstanten Werten besitzen, welche im Patricia Trie komprimiert werden. Der Patricia Trie wird somit auch nicht größer. Die Codierung der Bits entspricht der einfachen binären Umsetzung der Zahlenwerte des ASCII Codes der Zeichen. Dadurch ergibt sich für die Ordnung im Baum keine lexikographische, sonderen eine dem ASCII Wert zugrunde liegende Ordnung. Die Ordnung ist jedoch unerheblich, so lange sie in der gesamten Implementierung konsistent bleibt.

Die eigentliche Codierung der Eingabedaten geschieht nicht dauerhaft, sondern ist in den Operationen des Patricia Tries gekapselt. Diese Operationen finden Verwendung, wenn der Baum traversiert wird oder zwei Worte miteinander verglichen werden. Die Daten im Hintergrundspeicher werden dabei nicht verändert.

### 3.4.3 Splitting-Operation

Da der Splitting Algorithmus für binäre Patricia Tries selbst entwickelt wurde und somit keine Quelle dafür existiert, soll er an dieser Stelle genauer erläutert werden.

Ist ein Knoten in einem String B-Tree voll (d.h. er enthält $2 \cdot Grad - 1$ Elemente), so muss er gesplittet werden. Im Gegensatz zu einem normalen B-Baum ist im Knoten kein Feld, sondern eine Baumstruktur, der Patricia Trie, enthalten. Diese Baumstruktur muss nun

beim Splitten durch einen senkrechten "Schnitt" in zwei Teile geteilt werden. Dabei ist zu beachten, dass es sich in dem Prototypen nicht um m-äre, sondern wie oben beschrieben um binäre Patricia Tries handelt.

Die Angabe einer Splitstelle ist eine reine Positionsangabe und bezieht sich nur auf die Positionen der Blätter, da der restliche Teil des Baumes nach außen hin gekapselt (nicht sichtbar) ist. Das Problem dabei ist, dass die "Naht" der Splitstelle durch die gesamte Baumstruktur verläuft und deswegen überall berücksichtigt werden muss. Um also überall im Baum ausreichend Informationen über die Splitnaht zu besitzen, muss zumindest der Teilbaum, in dem die Naht verläuft, komplett durchlaufen werden. Praktischerweise wird beim Laden eines Knotens bei der Umwandlung der persistenten Darstellung in die Hauptspeicherpräsentation einmal der komplette Patricia Trie durchlaufen. Bei diesem Vorgang werden im Prototypen die zusätzlich notwendigen Informationen in den Patricia Trie Knoten gespeichert. Diese Information besteht im Grunde nur aus der aktuellen Position des Knotens bezüglich der Blätterliste: Die korrekte Position in den Blättern entspricht einfach deren horizontale Anordnung und in den inneren Knoten entspricht diese der größten Position im linken Teilbaum. In Abbildung 9 sind die Positionsangaben die Zahlenwerte, die in den Knoten enthalten sind[5].

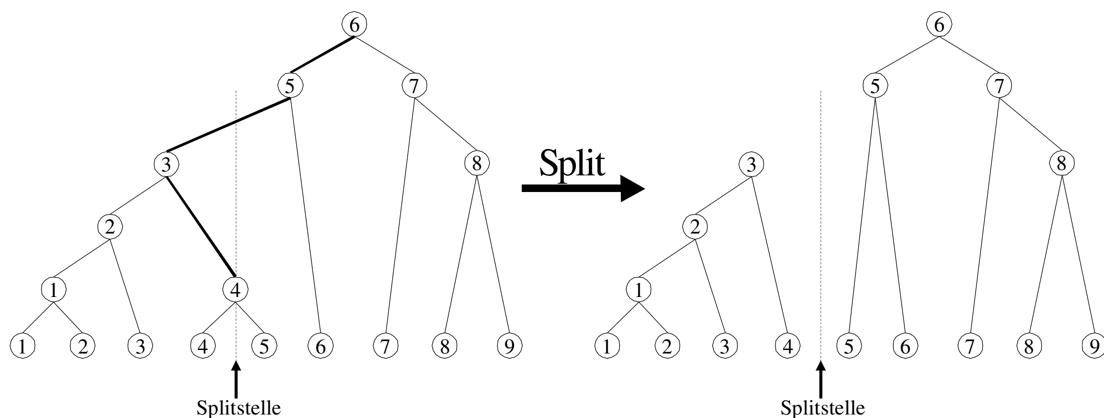


*Abbildung 9: Splitten eines binären Patricia Tries (Die Zahlen in den Knoten repräsentieren die Position bezüglich der Blätterliste)*

Verfügt man über diese Angaben, so ist es möglich in jedem Knoten festzustellen, ob dieser sich rechts oder links von der Splitstelle befindet. Mit Hilfe dieser Information kann man nun den Baum von der Wurzel aus traversieren. Dabei wird immer die Kante gewählt, die in die Richtung der Splitnaht geht. In der Abbildung ist dieser Weg mit einer dickeren Linie hervorgehoben. Jedesmal, wenn bei der Traversierung eine Richtungsänderung auftritt, also die Splitnaht gekreuzt wurde, muss die eben traversierte

5 Um die Abbildung so klar wie möglich zu gestalten, wurden alle weiteren Inhalte des Patricia Tries weggelassen. (Die Zahlen in den Knoten repräsentieren in diesem Schaubild nicht die Länge des gespeicherten Strings !)

Kante neu ausgerichtet werden, so dass sie nicht mehr die Splitnaht kreuzt und dadurch alle Invarianten im Baum erhalten bleiben. Das ist gewährleistet, wenn die Kante auf den Knoten verweist, an dem die nächste Richtungsänderung stattfindet. In der Abbildung ist der Fall gezeigt, in dem zwar keine Richtungsänderung stattfindet, jedoch genau die Splitstelle (Innerer Knoten mit Position 4) gefunden ist. Ist die gerade genannte Abbruchbedingung (Splitstelle gefunden) erfüllt, so wird die unterste Kante neu ausgerichtet und somit der Knoten an der Splitstelle gelöscht.

Ist das eigentliche Splitten der Bäume getan, so muss (je nach Splitstelle) für einen der beiden Bäume eine neue Wurzel gefunden werden, damit die Eigenschaften des Patricia Tries erhalten bleiben.

### 3.4.4 Caching

Ein sehr wichtiger Aspekt für die gesamte Speicherhierarchie eines Rechners und somit auch für den Zugriff auf den Hintergrundspeicher stellt das Caching (Puffern) dar. Ziel des Cachings ist es, so viele Zugriffe wie möglich auf den in der Hierarchie tieferen (und somit langsameren) Speicher zu vermeiden.

Grundsätzlich ist die Verwendung eines programminternen Puffers bei String B-Trees von großem praktischen Nutzen, da ein intensiver Datenaustausch zwischen Haupt- und Hintergrundspeicher stattfindet. Im Hinblick auf die Ziele dieser Arbeit ergeben sich aus der Implementierung eines Puffers wenig nutzen, sondern eher Nachteile bei den Untersuchungen. Da in den Untersuchungen die Eigenschaften einer String B-Tree Implementierung betrachtet werden, ist eine möglichst klare Umsetzung der Struktur nötig, damit deren Eigenschaften deutlicher erkennbar sind. Durch Hinzufügen eines Puffers wird die Implementierung zwar verbessert und dadurch schneller und effizienter (was in der praktischen Anwendung auch notwendig ist), bei den Messergebnissen ergibt sich jedoch eine unerwünschte Verzerrung und somit ein unklares Bild der zugrunde liegenden Struktur.

Ein weiterer Nachteil ergibt sich aus rein praktischen Überlegungen. Da der String B-Tree für große Textcorpora konzipiert wurde, die um ein vielfaches größer als der verwendete Hauptspeicher sind, zeigen sich die zu untersuchenden Eigenschaften des String B-Trees natürlich auch erst bei diesen sehr großen Datenmengen.

Die verwendete Implementierung benutzt nicht den gesamten Hauptspeicher, sondern nur so viel, wie für einen String B-Tree Knoten und die beiden Datenpuffer benötigt wird; es wird kein weiterer Knoten gepuffert. Dadurch zeigen sich die Eigenschaften des String B-Trees schon bei verhältnismäßig kleinen Baumgrößen. Es kann also unnötiger Aufwand an Speicherplatz und Zeit vermieden werden, ohne dabei die relativen Ergebnisse zu verfälschen.

Aus eben genannten Gründen wurde in der Implementierung des Prototyps auf die Verwendung eines selbst implementierten Puffers verzichtet. Speziell für den Vergleich mit anderen Datenstrukturen und für die Anwendung ist ein Puffer natürlich unverzichtbar

und einer der ersten Aspekte, der bei einer Weiterführung des Prototyps umgesetzt werden müsste.

### *3.4.5 Hintergrundspeicherzugriffe*

Durch den intensiven Gebrauch des Hintergrundspeichers ergibt sich der Hauptanteil an der Geschwindigkeit der Implementierung aus der Effizienz der verwendeten Zugriffsmethoden. Da für die experimentellen Untersuchungen nur ein auf den String B-Tree bezogener Vergleich notwendig ist, wurden für die Implementierung des Prototyps nur die Standard-Ein- und Ausgaberoutinen (fread, fwrite) verwendet.

Bei einer auf dem Prototyp aufbauenden Implementierung, welche für die praktische Anwendung oder stärker für den Vergleich mit anderen Datenstrukturen genutzt werden soll, ist die Verwendung von effektiveren Methoden (wie z.B. *Memory Mapping*[6]) eine wichtige Voraussetzung. Auf solche Verbesserungen soll hier jedoch nicht im Detail eingegangen werden, da diese mit der eigentlichen Implementierung nichts zu tun haben.

6 Damit ist das in UNIX C verwendete Memory Mapping (*mmap*) gemeint.

# 4. Planung der Untersuchungen

Nachdem das Modell des String B-Trees und im Anschluss die wichtigsten Eigenschaften des implementierten Prototyps erklärt wurden, soll nun der zentrale Aspekt dieser Arbeit, die experimentellen Untersuchungen, behandelt werden. In diesem Kapitel wird die Planung und Vorbereitung der Experimente erklärt. In dem danach folgenden Kapitel werden dann die einzelnen Untersuchungsergebnisse vorgestellt.

Wie bereits in der Einführung beschrieben, handelt es sich bei den String B-Trees um eine sehr aktuelle Datenstruktur, über die es bisher nur drei bekannte Arbeiten [6,5,14] gibt. Trotz dieser Tatsache sind bereits zwei experimentelle Untersuchungen der String B-Tree Datenstruktur vorhanden. Die eine Arbeit [5] ist ebenfalls von P. Ferragina und R. Grossi. In ihr wird ein Vergleich zwischen den Implementierungen des String B-Trees, des Suffix Arrays und des Präfix B-Baumes durchgeführt. Der Vergleich beschränkt sich jedoch nur auf die Anzahl der Festplattenzugriffe, die in der jeweiligen Implementierung gezählt wurden. Dabei wurden die verschiedenen Caches von Festplatte und Betriebsysstem bewusst nicht berücksichtigt. Bis auf eine kurze Erwähnung beim Vergleich mit Präfix B-Bäumen wurden keine Laufzeiten genannt. Aufgrund dieser eingeschränkten Betrachtung lassen sich diese Ergebnisse nicht für diese Arbeit verwenden und werden im Folgenden nicht weiter berücksichtigt.

Die Ergebnisse der anderen experimentellen Arbeit von K. R. Rose [14], welche eine String B-Tree Implementierung mit dem Datenbanksystem *Sleepycat* vergleicht, ist ebenfalls für diese Arbeit unerheblich. Durch den Vergleich mit einem Datenbanksystem sind die untersuchten Anwendungsfälle speziell auf diesen Kontext bezogen und für die in dieser Arbeit durchgeführten Untersuchungen bezüglich Genom-Datenbanken und Information Retrieval praktisch kaum von Bedeutung.

Als erster Schritt in der Vorbereitung stellt sich die Frage, von welcher Art und welchem Umfang die durchzuführenden Experimente sein sollen. Dabei ergeben sich drei Bereiche:

- **Überprüfung der theoretischen Aussagen**

  Diese Untersuchung soll die in [6] gegeben theoretischen Aussagen für die benötigten Hintergrundspeicherzugriffe anhand der Implementierung überprüfen. Konkret werden dabei die exakte Substringsuche, welche $O(p/s + \log_g n)$ und das Einfügen, das $O(q/s + \log_g n)$ Zugriffe benötigt, untersucht. Dabei soll betrachtet werden, inwieweit sich diese Aussagen in die praktischen Anwendungsfälle übertragen lassen.

- **Untersuchung der Eigenschaften des String B-Trees**

  Dieser Bereich bildet den Schwerpunkt der gesamten Untersuchungen. In ihm sollen sowohl die Eigenschaften des String B-Trees selbst (Baum- und Knotengröße) als auch sein Verhalten bezüglich verschiedener Datenmengen mit unterschiedlicher Struktur (Alphabetgröße, Zeichenverteilung, Repetitivität) anhand von Experimenten untersucht werden. Aufgrund dieser Ergebnisse soll festgestellt werden, für welche Anwendungsfälle der String B-Tree gut bzw. weniger gut geeignet ist. Bei dieser Bewertung sollen besonders die Suchoperation und die Bereiche Genom-Datenbanken

und Information Retrieval berücksichtigt werden.

- **Vergleich mit einer anderen Datenstruktur**
  Der Vergleich mit einer anderen Datenstruktur ist nur ein untergeordneter Teil, der eine erste Bewertung der Praxisrelevanz ermöglichen soll. Die zu vergleichende Datenstruktur sollte deshalb eine für Genom-Datenbanken und Information Retrieval konzipierte Datenstruktur sein. Auf einen ausführlichen Vergleich mit einer bzw. mehreren anderen Datenstrukturen wird verzichtet, da dies den Rahmen dieser Arbeit sprengen würde und somit eher ein sinnvoller Ansatz für weiterführende Arbeiten wäre.

Aufgrund der Festlegung dieser Bereiche muss nun eine konkrete Betrachtung der für die Untersuchung notwendigen Einfluss- und Messgrößen durchgeführt werden.

## 4.1 Einflussgrößen

Im Folgenden sollen sämtliche Einflussparameter angegeben und im einzelnen beschrieben werden.

- **Parameter der Datenstruktur**
  Diese Parameter umfassen die veränderlichen Größen der Datenstruktur und der Implementierung.
  - *Größe des String B-Trees*
    Die Größe des String B-Trees, die durch die Anzahl seiner Elemente gegeben ist, stellt eine entscheidende Einflussgröße dar, welche in sämtlichen Untersuchungen berücksichtigt werden muss. Sie ist sehr variabel und wird durch die zugrunde liegende Textgröße $z$ bestimmt.
  - *Grad der Knoten des String B-Trees*
    Der Knotengrad und die daraus resultierenden Knotengröße wirkt sich direkt auf die Höhe des Baumes aus und ist besonders in Hinblick auf die Überprüfung der theoretischen Aussagen eine relevante Größe. In den technischen Untersuchungen wird der optimale Wert experimentell ermittelt und für alle Folgeuntersuchungen festgesetzt.
  - *Der Datenpuffer*
    Die Größe des Datenpuffers gibt an, wie viele Daten bei einem Festplattenzugriff[7] in den Hauptspeicher geladen werden.
- **Datenparameter**
  Mit der Bezeichnung Datenparameter sind alle veränderbaren Einflussgrößen bezüglich der zugrunde liegenden Eingabedaten (Textcorpora, Genomdaten, etc.) gemeint. Alle aufgeführten Datenparameter werden in den experimentellen Untersuchungen berücksichtigt und zum Teil isoliert betrachtet.

7 Damit ist ein Festplattenzugriff im Datenmodell gemeint, welches im Folgeabschnitt *Messgrößen* beschrieben wird.

- *Ein- / Mehrworttexte*
  Dem String B-Tree ist es möglich, sowohl Texte, die nur aus einem Wort bestehen (z.B. Genomdaten) als auch Texte, die mehrere Worte beinhalten (z.B. Wörterbücher) zu verarbeiten.
- *Größe des Alphabetes*
  Je nach Anwendung können die Alphabetgrößen der Texte stark schwanken. Häufig vorkommende Größen sind z.B. 4 (DNA-Daten) und ca. 80 (für normale ASCII Textdateien).
- *Verteilung der Buchstaben im Alphabet*
  Auch die Verteilung der Buchstaben im Alphabet ist eine veränderbare Größe, die sich auf die Laufzeit der Implementierung auswirken kann. In der Praxis treten in den Datenmengen (natürlichsprachliche Texte, Genomdaten, Programmiertexte) viele verschiedene Verteilungen auf. Zusätzlich ist auch die Wiederholung von Mustern im Text ein wichtiger Aspekt, der in den Untersuchungen der realen Texte betrachtet wird.
- *Art des Textes*
  Die Art des Textes ist im Prinzip die Zusammenfassung der drei bisher genannten Einflussgrößen. Jedoch sollen diese nicht nur einzeln untersucht, sondern auch gerade in Wechselwirkung mit den anderen Größen betrachtet werden. Da dies für praktische Anwendung von besonderem Interesse ist, werden hierbei auch nur reale (keine künstlichen) Datenmengen benutzt.

- **Umgebungsparameter**
  Hier werden all die Größen aufgeführt, die die bisherigen „umgeben“ und für die Untersuchungen als fest angenommen werden.
  - *Betriebssystem*
    Hiermit werden alle Einflussgrößen, die durch das Betriebssystem des Rechners gegeben sind, zusammengefasst. Dieses umfasst eine Vielzahl an Parametern, wie z.B. die Seitengröße des Betriebssystems oder die Größe und Art des Caches. Änderungen dieser Parameter haben für die Untersuchungen nur eine geringe Relevanz, weshalb diese als gegebene Größen vorausgesetzt werden (vgl. Abschnitt 5.1.2).
  - *Zugrunde liegende Hardware*
    Mit der Hardware wird die Gesamtheit aller Bauteile zusammengefasst, die eine Auswirkung auf die Experimente haben. Auch sie werden als gegeben angesehen (vgl. Abschnitt 5.1.2).

## 4.2 Messgrößen

Nachdem sämtliche Einflussparameter betrachtet wurden, sollen nun die möglichen Messgrößen und deren Aussagekraft im Hinblick auf die beiden Hauptaspekte der

Untersuchungen, nämlich die Überprüfung der theoretischen Aussagen und der praktischen Anwendung der String B-Trees analysiert werden.

### *Festplattenzugriffe (FPZ)*

Da, wie schon mehrfach erwähnt, die technischen Detailaspekte aufgrund fehlender Notwendigkeit nicht genauer betrachtet werden sollen, wird auch bei der Messung der Festplattenzugriffe auf eine reale Messung der tatsächlichen Zugriffe verzichtet. Außerdem wäre eine reale Zeitmessung nur bedingt mit den theoretischen Aussagen aus [6] vergleichbar gewesen, da sich die O-Notation nicht auf die konkrete Laufzeit bezieht und in ihr keine Konstanten berücksichtigt werden.

Stattdessen wird das Modell aus [6] verwendet, d. h. in der Implementierung umgesetzt, damit auch ein Vergleich mit theoretischen Aussagen durchgeführt werden kann. Das Modell geht davon aus, dass für eine geladene Seite ein Festplattenzugriff erfolgt. Grundsätzlich sind dabei zwei verschiedene Arten von Festplattenzugriffen zu unterscheiden. Zum einen ein Zugriff auf die Datei, in der sich der Baum befindet, und zum anderen ein Zugriff auf die Datei, in der sich die Eingabedaten befinden. Dabei wird davon ausgegangen, dass sich jeder Knoten des String B-Trees in genau einer Seite befindet und somit für das Laden eines solchen Knotens genau ein Zugriff nötigt ist.

### *Rechenzeit (CPU-Zeit)*

Die zweite gemessene Größe ist die Rechenzeit des Programmes, also die Zeit, die der Prozessor für die Ausführung des Programmes benötigt hat. Diese Zeit ist unabhängig von anderen parallel laufenden Prozessen und somit eine sehr genaue Angabe für die Laufzeit eines Programmes.

Bei Laufzeitmessungen von Programmen ist sie deshalb die am häufigsten verwendete Messgröße. Bei Untersuchungen dieser Arbeit hat sie als Messgröße jedoch in den meisten Fällen nur eine unterstützende Funktion, da der Hauptaspekt, nämlich die Dauer der Festplattenzugriffe, durch sie nicht gemessen werden kann. In vielen Fällen kann sie jedoch bei der Bewertung der Gesamtzeiten erste Anhaltspunkte über deren Ursache liefern. Letztendlich lässt sich an ihr auch in etwa das Verhältnis zwischen Rechenzeit und Zugriffszeit[8] ermitteln, was wiederum Auskunft über die Relevanz der Codeoptimierungen im Hinblick auf die Hintergrundspeicherzugriffe gibt. Darauf soll jedoch in den Untersuchungsergebnissen noch genauer eingegangen werden.

Die konkrete Messung der Rechenzeit in der Implementierung erfolgte in den laufzeitrelevanten Abschnitten mit Hilfe der C-Funktion *clock()*.

8 Da in dieser Arbeit häufig die Begriffe "Festplatten- und Hintergrundspeicherzugriffe" verwendet werden und damit ein kompletter Zugriff (Suchen + Laden der Datenseite) gemeint ist, soll mit dem Wort "Zugriffszeit" ebenfalls die Zeit für einen kompletten Zugriff (Suchen + Laden) gemeint sein.

### *Gesamtlaufzeit*

Die wichtigste und zugleich am schwierigsten zu interpretierende Messgröße ist die Gesamtlaufzeit. Wie der Name schon sagt, gibt die Gesamtlaufzeit die Zeit an, die vom Start eines Programmes (oder auch Abschnittes) bis zur Beendigung des selbigen vergeht. Dabei werden sämtliche parallel laufenden Prozesse mitberücksichtigt, was in Extremfällen zuverlässige Aussagen so gut wie unmöglich macht. Deshalb versucht man so weit wie möglich sämtliche Quellen für die Verursachung von Messstörungen auszuschalten. Trotzdem ist bei der Messung und der Bewertung der Gesamtlaufzeit immer zu berücksichtigen, dass die Varianz der Messwerte grundsätzlich relativ hoch ausfällt.

In den Untersuchungen dieser Arbeit dient die Gesamtlaufzeit in erster Linie dazu, die benötigte Zeit der Festplattenzugriffe zu messen. Doch selbst wenn man davon ausgehen kann, dass ein Großteil dieser Zeit wirklich den Zeitverbrauch der Festplattenzugriffe darstellt, ergibt sich das Problem, wie genau diese Zugriffszeit zu bewerten ist.

Um eine genauere Bewertung der Zugriffszeiten vorzunehmen, wird kurz noch einmal die Funktionsweise einer Festplatte verdeutlicht. Dabei sind zwei Größen zu beachten: Die Suchzeit und die Transferrate. Mit dem Begriff "Suchzeit" ist der Zeitabschnitt bezeichnet, welcher für die Suche (inkl. Latenzzeit) des entsprechenden Speicherblocks auf der Festplatte vergeht. Technisch gesehen ist dies die Zeit, welche der Lesekopf benötigt, um sich richtig zu positionieren. Die zweite Größe ist die Transferrate, welche den Datendurchsatz[9] der Festplatte bezeichnet. Aus dieser Größe läßt sich die Zeit berechnen, welche für das Laden einer bestimmten Anzahl von Blöcken[10] (Seiten) benötigt wird.

Die Zugriffszeit setzt sich nun aus der Summe dieser beiden Zeiten zusammen, wobei zu beachten ist, dass die Suchzeit gerade bei kleinen Datenmengen (z.B. einen Block) den erheblich größeren Anteil darstellt. Bei den String B-Trees ist dies der Fall, da die Knoten im Idealfall der Größe eines Blocks entsprechen. Ein weiteres Problem besteht nun darin, dass die Suchzeit im Gegensatz zur Durchsatzrate sehr variabel ist, da sie von der Lokalität der Datensätze auf der Festplatte abhängt, was ganz einfach daran liegt, dass der Lesekopf längere Wege zurücklegen muss und deswegen auch mehr Zeit benötigt. Somit spielen also auch die Lage der Dateien und deren Fragmentierung ein Rolle.

Als weitere Einflussgrößen seien noch kurz das Dateisystem des Betriebsystems und der Festplattencache im Betriebsystem und auf der Festplatte selbst erwähnt. An dieser Stelle soll jedoch nicht genauer darauf eingegangen werden, da speziell die Auswirkungen der Caches in den Untersuchungen genauer besprochen werden.

Abschließend ist zu sagen, dass bei der Durchführung der Experimente alle genannten Faktoren natürlich so weit wie möglich vermieden bzw. berücksichtigt wurden. Trotzdem sind diese Einflüsse bei allen experimentellen Untersuchungen, die eine Betrachtung der Gesamtzeitmessung vornehmen, immer miteinzubeziehen.

---

9 Meist in Megabyte pro Sekunde gemessen.

10 Ein Block ist die kleinste Einheit, die von der Festplatte geladen werden kann.

Die konkrete Messung der Gesamtlaufzeit in der Implementierung erfolgte in den laufzeitrelevanten Abschnitten mit Hilfe der C-Funktion *time()*.

## 4.3 Operationen

Als letztes sollen die Operationen auf String B-Trees aufgeführt und ihre Relevanz im Hinblick auf die durchzuführenden Untersuchungen beurteilt werden.

Im Grunde genommen besitzt der String B-Tree genau wie der B-Baum drei Grundoperationen. Gerade im Information Retrieval und in der Genforschung gibt es viele komplexere und weiterführende Operationen wie z.B. die häufig verwendete Ähnlichkeits- oder Teilsequenzsuche. All diese Operationen lassen sich jedoch immer auf die drei elementaren (Suche, Einfügen, Löschen) zurückführen. Aus diesem Grund sollen nur diese Operationen betrachtet werden:

- **Suche**
  Die Suche ist die wichtigste und in der Anwendung am häufigsten verwendete Operation. Weiterhin wird die Suche ebenfalls in den beiden anderen Operationen (Einfügen, Löschen) verwendet, wodurch sie sozusagen den Kern aller Operationen des String B-Trees darstellt. Aufgrund dieser repräsentativen Eigenschaft der Suchoperation ist eine diesbezügliche Untersuchung besonders wichtig.
  Bis auf die Untersuchung der theoretischen Aussagen wird in allen folgenden Experimenten ausschließlich die exakte Substringsuche in den String B-Trees, welche überprüft, ob die Substrings enthalten sind, untersucht.
- **Einfügen**
  Das Einfügen, dessen Laufzeitverhalten zu einem Großteil von der Suchoperation bestimmt wird, ist sowohl notwendig, weitere Elemente in einen String B-Tree einzufügen, als auch dafür da, den String B-Tree zu erstellen. Aufgrund des sehr ähnlichen Laufzeitverhaltens der Einfüge- und der Suchoperation wurde diese Operation nur bei der Überprüfung der theoretischen Aussagen genauer betrachtet.
- **Löschen**
  Da die Löschoperation für die Untersuchungen nur wenig Relevanz besitzt, wurde das Löschen von Elementen nicht implementiert und wird somit auch nicht untersucht.

## 4.4 Untersuchungen am B-Baum

In der Vorbereitungsphase der Untersuchungen ist eine Implementierung der B-Baum Datenstruktur durchgeführt worden, anhand derer bereits eine kleine Untersuchungsreihe durchgeführt wurde. Dies erschien als Vorbereitung sinnvoll, da die String B-Tree Struktur mit der B-Baum Struktur viele Gemeinsamkeiten aufweist. Aufgrund des Verlaufes der Untersuchungsergebnisse sollten bereits im Vorfeld Probleme erkannt und eine Basis für die Beurteilung der Ergebnisse der String B-Tree Untersuchungen

geschaffen werden.

In den folgenden Abschnitten wird teilweise auf diese Ergebnisse verwiesen. Aufgrund der Ähnlichkeit zu den String B-Tree Ergebnissen sollen diese jedoch an dieser Stelle nicht weiter aufgeführt werden.

# 5. Experimentelle Untersuchungen

In diesem Abschnitt werden einleitend der Aufwand und die Methodik der Untersuchungen beschrieben, welche die allgemeinen Probleme und Bedingungen der Experimente erklären. Danach werden alle Untersuchungen und deren Ergebnisse beschrieben. Es erfolgt zusätzlich eine Interpretation und erste Auswertung der Untersuchungsergebnisse. Im nächsten Kapitel folgt die Gesamtauswertung der Ergebnisse.

## *5.1 Vorbemerkungen*

### *5.1.1 Zeitaufwand der Experimente*

Wie schon in der Erklärung des Modells und in anderen Kapiteln oft erwähnt, ist der String B-Tree von seiner äußeren Struktur dem B-Baum sehr ähnlich. Er ist eine Datenstruktur, die von vornherein auf die Nutzung des Hintergrundspeichers ausgelegt ist; ganz im Gegensatz zu vielen anderen persistenten Datenstrukturen, die zuerst im Hauptspeicher aufgebaut werden und dann mit verschiedenen Methoden auf den Hintergrundspeicher abgebildet werden. Wie schon im Kapitel 3 (*Implementierung)* beschrieben, ist es notwendig, diese Eigenschaft so klar wie möglich, das heißt ohne Optimierungen, umzusetzen. Daraus resultiert jedoch ein sehr hoher Zeitaufwand, der sich sowohl beim Test der Implementierung als auch bei den experimentellen Untersuchungen ergibt. Der Umfang und die Testmengen der Untersuchungen mussten also sorgfältig gewählt und zum Teil leicht eingeschränkt werden, da sie sonst den Zeitrahmen der Arbeit gesprengt hätten.

Um ein konkretes Beispiel zu nennen: Die Erstellung eines String B-Trees mit 1 Million Elementen (Suffixen) dauert bei optimalem Knotengrad auf dem verwendeten Testrechner knapp 10 Stunden. Zusammenfassend ist zu bemerken, dass der String B-Tree durch seine Eigenschaften als Datenstruktur, die speziell für große Textcorpora konzipiert wurde, die die Größe des Hauptspeichers übersteigen, nur mit relativ großem Zeitaufwand zu untersuchen ist.

### *5.1.2 Methodik der Untersuchungen*

Alle folgenden Untersuchungen wurden lokal auf einer Sun Sparc Station (300 Mhz, 128 MB RAM) unter Solaris durchgeführt. Ein Großteil vom Festplattencache des Betriebssystem wurde dabei ausgeschaltet. Parallel zum Testbetrieb liefen nur sehr niedrig prioritäre Systemprozesse. Der Prototyp *sbtree* wurde mir dem ecgs Compiler, Version 1.1.2 und der Optimierungsoption "-O3" compiliert.

Bei der Einfügeoperation wurde der komplette Vorgang, bestehend aus dem Laden des Einfügewortes aus einer Datei, dem Suchen der Einfügeposition und dem Einfügen und

Zurückschreiben des Einfügeknotens gemessen.

Bei der Suchoperation wird für die Messung nur die Suche nach der Einfügeposition, also die Überprüfung, ob das Muster enthalten ist, untersucht. Die Ausgabe der Suchergebnisse, die in linearer Zeit bzgl. der Anzahl der gefundenen Worte läuft, wird dabei nicht berücksichtigt, da sie nichts mit der eigentlichen Suche in der String B-Tree Struktur zu tun hat.

Die in der Suche verwendeten Suchstrings werden aus der gleichen Stringdatei genommen, die für den Aufbau des Baumes verwendet wurde. Das heißt, dass die Suchmuster im Baum enthalten sind und somit bei der Suche im String B-Tree mindestens in einer Patricia Trie Suche (beim Vergleich mit dem *lcp-Wort*[11]) komplett durchlaufen werden. Das hat zwar meist eine längere Laufzeit zur Folge als bei zufällig ausgewählten Worten; diese ist im Gegensatz dazu jedoch konstant und damit für die Messungen besser geeignet. Die Länge des Suchmusters entspricht eigentlich der Länge des Wortes aus der Stringdatei. In den meisten Fällen wird sie jedoch sinnvollerweise verkürzt, was in den folgenden Untersuchungen von Fall zu Fall erklärt wird.

Alle für die Untersuchungen künstlich erstellten Texte wurden so generiert, dass alle enthaltenen Zeichen gemäß dem Bernoulli Modell unabhängig und identisch verteilt sind. Für die Zufallsbestimmung eines Buchstabens wurde die Gleichverteilung verwendet. Die Alphabetgrößen und Wortlängen sind unterschiedlich und werden an den entsprechenden Stellen angegeben.

Jeder einzelne Test wurde zehn mal wiederholt und als Ergebnis das arithmetische Mittel aller Testläufe genommen. Dabei ist zu bemerken, dass bei allen Testläufen die Standartabweichung bei den Messungen der CPU- und der Gesamtzeit weniger als 3% beträgt (Maximalabweichung weniger als 5%). Die Messergebnisse der Festplattenzugriffe waren natürlich konstant.

In allen folgenden Untersuchungen wird die Version 1.0 des *sbtree* Prototyps verwendet. Parallel wurden bereits zum Teil Änderungen an der Implementierung vorgenommen, die in den Untersuchungen jedoch nicht verwendet wurden, um eine Vergleichbarkeit zwischen den Experimenten zu gewährleisten. Details zur aktuellen Version der Implementierung finden sich unter *http://www.inf.fu-berlin.de/~mscholz/sbtree/.*

## *5.2 Überprüfung der theoretischen Aussagen*

Wie in Abschnitt 2.6 bereits erwähnt, wurden in [6] die Aussagen getroffen, dass die exakte Substringsuche $O(p/s+\log_g n)$ und das Einfügen $O(q/s+\log_g n)$ Festplattenzugriffe benötigt. Die erste Untersuchung soll experimentell überprüfen, inwieweit sich diese Aussagen in die praktische Anwendung übertragen lassen und wie getreu das theoretische Modell in der Implementierung umgesetzt wurde. Um die Ergebnisse der Untersuchungen mit den Thesen vergleichbar zu machen, müssen folgende

11 vgl. Abschnitt 2.3 *Suche im Patricia Trie*

Parameter berücksichtigt werden:

- **Wortlänge**
  Je nach Wortlänge und der Vergleichslänge zweier Worte kann es vorkommen, dass man mehrere Festplattenzugriffe benötigt, um zwei Worte zu vergleichen. Um dies zu vermeiden, wurde ein Mehrworttext benutzt, der Worte mit gleicher Länge (25 Buchstaben) enthielt.
- **Datenpuffergröße**
  Um möglichst klare Ergebnisse zu erhalten, ist es am besten, dass bei jedem Lade- bzw. Schreibvorgang eines Wortes genau ein Zugriff notiert wird. Um dies zu erreichen, wird die Puffergröße mit der Wortlänge gleichgesetzt.
- **Caching Effekte**
  Da die Worte beim Einfügen und Suchen sequentiell aus der Stringdatei gelesen werden, kann es zu Caching-Effekten kommen, wenn der Datenpuffer die gesuchten Daten bereits enthält. Um diese Effekte weitestgehend zu vermeiden, wurde das nächste einzufügende (bzw. zu suchende) Wort so gewählt, dass es genau hinter dem Datenpufferbereich lag und dieser somit erneut gefüllt werden musste.
- **Knotengrad**
  Wie beim B-Baum bestimmt der Knotengrad den Höhenzuwachs des String B-Trees. Damit der Baum möglichst wenig wächst, ist für eine normale Anwendung ein hoher Knotengrad von Vorteil. In dieser Untersuchung ist für die Komplexitätsbetrachtung der Hintergrundspeicherzugriffe ein Baum mit kleinerem Knotengrad jedoch viel aussagekräftiger, da an ihm das Wachstumsverhalten viel deutlicher zu erkennen ist.

Die Berechnung der Zugriffe auf die String B-Tree Datei muss dabei nicht angepasst werden, da diese unabhängig von der Knotengröße bei jedem Schreib- bzw. Ladevorgang genau einen Zugriff notiert.

Nachdem die Implementierung (so weit es geht) an das Modell angepasst ist, soll erörtert werden, welche Ergebnisse zu erwarten sind. Betrachtet man zunächst einen einzelnen Knoten, so wird für das Laden des Knotens und das Laden des Vergleichwortes (für die Suche im Patricia Trie) je ein Zugriff benötigt. Wird ein Element in einen Knoten eingefügt, so ergibt sich ein zusätzlicher Zugriff. Da es sich um einen $B^+$-Baum handelt, wird dieser immer von der Wurzel bis zu einem Blatt, also in seiner gesamten Höhe traversiert. Somit ergeben sich $h \cdot 2$ (*mit h = Höhe des String B-Trees*) Festplattenzugriffe für eine Suche plus einem Zugriff für das Speichern des Knotens, in dem das Element eingefügt wurde. Weil die einzufügenden (zu suchenden) Worte auch aus der Stringdatei geladen werden müssen, muss dazu noch ein weiterer Zugriff addiert werden. Damit ergeben sich für eine Suche $(h \cdot 2)+1$ und für eine Einfügeoperation $(h \cdot 2)+2$ Festplattenzugriffe

Die Höhe $h$ läßt sich nun wie in einem B-Baum durch den Knotengrad berechnen und beträgt $O(\log_g n)$. Die O-Notation vernachlässigt jedoch den Füllungsgrad des Knotens, der verschiedene Werte annehmen kann. Genauer gesagt gilt

$g-1 \leqslant Füllungsgrad \leqslant 2 \cdot g-1$, das heißt, es müssten sich ingesamt mindestens $2 \cdot (\log_{(2 \cdot g-1)} n)+1$ und höchstens $2 \cdot (\log_{g-1} n)+1$ Zugriffe[12] für eine Suche und für eine Einfügeoperation ein Zugriff mehr ergeben. Dies deckt sich auch mit den Aussagen aus [6], welche in Abschnitt 2.6 beschrieben wurden.

In dem folgenden Diagramm sind nun die Ergebnisse von 1000 Einfüge- bzw. Suchoperationen dargestellt. Weiterhin sind die gerade erwähnten theoretischen Ober- und Untergrenzen eingefügt, um die Bewertung der Ergebnisse zu vereinfachen. Es ist zu beachten, dass es sich auf der X-Achse um eine logarithmische Skalierung handelt. Die angegebene Größe des String B-Trees entspricht der Größe des Baumes, nachdem die 1000 Elemente eingefügt wurden.

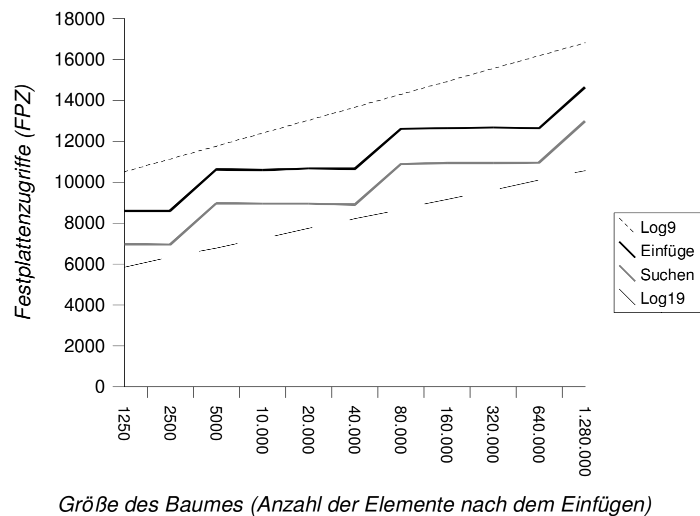


*Diagramm 1: FPZ bei 1000 Einfüge- bzw. Suchoperationen in einem String B-Tree mit Knotengrad 10*

Wie im Diagramm zu erkennen ist, verlaufen die beiden Ergebnisgraphen der experimentellen Untersuchungen genau in dem durch die Ober- und Untergrenzen vorgegebenen Bereich. An den Stufen der Ergebnisgraphen ist das Wachstum des String B-Trees zu erkennen. Weil der String B-Tree wie ein B-Baum schrittweise in der Wurzel wächst, ergibt sich insgesamt eine Art Treppe. Im Diagramm ergeben sich bei 1000 Durchläufen Stufenhöhen von 2000 Festplattenzugriffen und Höhenwerte zwischen 3 (1.250 Elemente) und 6 (1.280.000 Elemente).

Es gibt jedoch ein paar Abweichungen vom bereits vorgestellten theoretischen Modell, wie z.B. den erhöhten Gesamtwert der Zugriffe beim Einfügen. So ergeben sich für einen String B-Tree mit Höhe drei 8582 Zugriffe anstatt der berechneten 8000. Dieses Verhalten liegt ausschließlich an den Splittingoperationen (vgl. Abschnitt 3.4.3), die durch

12 Ohne Berücksichtigung der Zugriffe für die Splitoperationen.

die B-Baum-Struktur des String B-Trees hervorgerufen werden. Bei der Splittingoperation werden volle Knoten in der Mitte geteilt (gesplittet) und das mittlere Element im Vaterknoten gespeichert. Daraus ergeben sich 3 zusätzliche Festplattenzugriffe. Da die Häufigkeit der Splittingoperationen von der Beschaffenheit der einzufügenden Daten abhängt (Alphabetgröße, Buchstabenverteilung, etc.), ist sie theoretisch nicht so genau zu bestimmen und wurde deswegen in den Betrachtungen weggelassen. Die experimentelle Untersuchung der Anzahl der Splittingoperationen hat jedoch ergeben, dass die (um ca. 7%) erhöhten Messwerte genau durch diese hervorgerufen werden.

Eine weitere Auffälligkeit ergibt sich, wenn man die Ergebniswerte der Suche genauer betrachtet. Diese Ergebnisse liegen zwar wesentlich näher an den theoretisch berechneten Werten als die der Einfügeoperationen, sind aber um ca. 1-2 Prozent geringer als diese. Dieser etwas geringere Wert ist auf eine Wiederverwendung des Datenpuffers zurückzuführen, da sich dabei kein weiterer Festplattenzugriff ergibt. Trotz der vorhin beschriebenen Maßnahmen, die Verwendung des Datenpuffers so weit wie möglich dem Modell anzupassen, gibt es einige Ausnahmefälle, in denen sich ein Caching-Effekt nicht ausschließen läßt. Das kann z.B. der Fall sein, wenn beim Traversieren des Baumes ein Suchmuster zweimal hintereinander mit dem gleichen Wort im Baum verglichen wird, was in der $B^+$-Baum Struktur des String B-Trees möglich ist.

Berücksichtigt man die beiden angesprochenen Einflussfaktoren, so ergibt sich eine genaue Übereinstimmung mit den theoretischen Aussagen.

Dieses positive Resultat der ersten Untersuchung hat experimentell gezeigt, dass die Komplexität der Implementierung mit den in [6] angeführten Aussagen übereinstimmt. Dieses Ergebnis wird als Grundlage für alle folgenden Beobachtungen verwendet.

## 5.3 Technische Untersuchungen

Nachdem in der ersten Untersuchung die Implementierung mit den theoretischen Aussagen verglichen wurde, sollen nun die technischen Aspekte (Verhalten des Caches, optimaler Knotengrad) genauer untersucht werden.

### 5.3.1 Auswirkungen der Caches

Wie bereits in der Planung angenommen, wirken sich die verschiedenen Puffer und Caches, der sich zum Teil in der Hardware befinden (Festplatteninterner Cache) und zum Teil in der Software umgesetzt werden (Betriebssystem, Compiler), erheblich auf die Gesamtzeitmessung aus. Da der Einfluss eines Caches primär von der Dateigröße des Baumes abhängt, ist das Ziel dieser ersten Untersuchung herauszufinden, welche Baumgröße für die Experimente gegeben sein muss, damit ein Cachingverhalten so weit wie möglich ausgeschlossen werden kann. Dabei soll jedoch gleichzeitig berücksichtigt werden, dass die Größe aus praktischen Gründen ein gewisses Maß nicht übersteigen darf.

In der ersten Untersuchung wird jeweils eine ganze Reihe von String B-Trees verschiedener Größe untersucht, welche die gleiche Textdatei[13] verwenden. Die untersuchten Größen entsprechen der Formel $2^x \cdot 10.000$ (*mit x = 0,..,7*), reichen also von 10.000 bis zu 1.280.000 Elementen[14]. Beim Erstellen der Bäume wird die entsprechende Anzahl der Suffixe (je nach Baumgröße) eingefügt. Bei der Suche werden ganze Worte gesucht, die bereits im Baum enthalten sind. Die String B-Trees sind entsprechend ihres gleichen Grades zusammengefasst und mit einem Graphen repräsentiert. Der Knotengrad des Baumes beträgt 110. Die Größe der zwei Datenpuffer beträgt in diesem wie in allen folgenden Untersuchungen je 16.000 Zeichen. Beim Diagramm ist wieder zu beachten, dass die X-Achse eine logarithmische Skalierung besitzt.

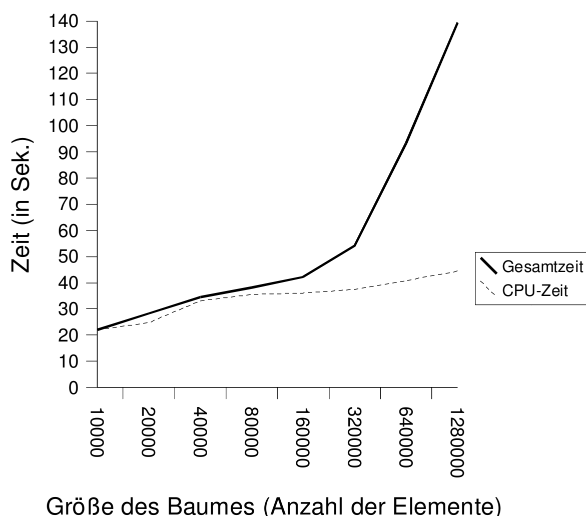


*Diagramm 2: Auswirkungen des Caches auf das Laufzeitverhalten anhand der Gesamtzeit und CPU-Zeit Kurve*

Wie man deutlich erkennen kann, ist der Verlauf der Gesamtzeitkurve bis zu einer Baumgröße von 160.000 Elementen (ca. 7 Megabyte) nahezu linear und sehr dicht an der CPU-Zeit Kurve. Das zeigt, dass ein Großteil der Festplattenzugriffe durch den Cache gepuffert werden und dadurch nur eine geringfügige Zeitverzögerung auftritt.

Betrachtet man jedoch den weiteren Verlauf der Kurve ab 160.000 Elementen, so stellt man fest, dass die CPU-Zeit weiterhin linear und mit der gleichen Steigung verläuft, die Gesamtzeitkurve jedoch eine wesentlich größere Steigung aufweist. Dabei ist zu beachten, dass die Kurve nicht exponentiell verläuft, sondern weiterhin linear bleibt, auch wenn es auf den ersten Blick nicht den Anschein hat. Hier kann man deutlich erkennen, dass die Größe des Caches nicht mehr ausreicht, um die Mehrzahl der Festplattenzugriffe

13 Die Textdatei besteht aus 2 Millionen Einzelworten der Länge 25 mit einer Alphabetgröße von 26.

14 Aus zeitlichen Gründen war diese Größe die maximal verwendbare (Einzelexperimente ausgenommen).

aufzufangen und somit zunehmend tatsächliche Zugriffe stattfinden, welche erheblich mehr Zeit verbrauchen.

Interessant ist auch, dass der Übergang zwischen einem vom Cache stark beeinflussten zu einem vom Cache nur sehr wenig beeinflussten Verhalten relativ abrupt (zwischen 160.000 und 320.000) stattfindet. Davor und danach ist die Kurve jeweils mit ungefähr gleichbleibender Steigung linear[15].

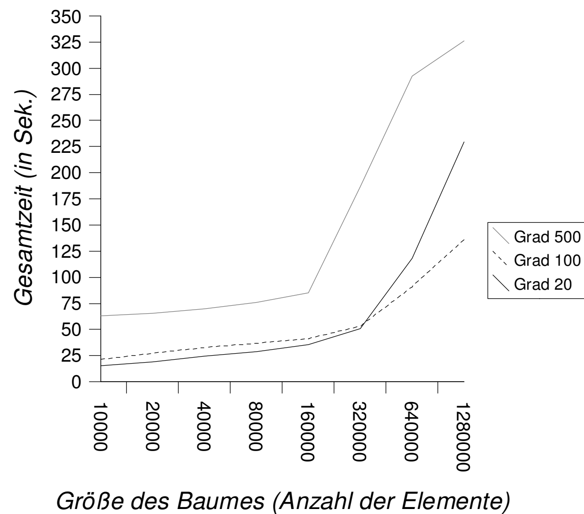


*Diagramm 3: Auswirkungen des Caches auf das Gesamtlaufzeitverhalten anhand dreier String B-Tree Reihen mit unterschiedlichen Graden*

Um den linearen Kurvenverlauf ein wenig genauer zu betrachten, sollen zwei weitere String B-Tree Reihen mit sehr extremen (großen und kleinen) Knotengraden betrachtet werden.

Auch bei den anderen Kurven (Grad 20 und Grad 500) stellt man das gleiche Verhalten fest, jedoch unterscheiden sich die Steigungen ein wenig voneinander, was aufgrund der verschiedenen Knotengrade auch zu erwarten ist. Weiterhin ist zu erkennen, dass bei einem sehr großen Knotengrad der Cache schon bei kleineren Baumgrößen an Wirkung verliert als bei einem kleineren Knotengrad. Das hängt sehr mit der Cacheverwaltung zusammen, welche wahrscheinlich mit kleineren Knoten effektiver arbeitet. Trotz dieser Beobachtung ist zu erkennen, dass die Auswirkungen des Caches nur eingeschränkt vom Knotengrad des Baumes abhängen.

Zusammenfassend ist festzuhalten, dass ab einer Baumgröße von 640.000 Elementen nur sehr geringe Verzerrungen durch den Cache zu erwarten sind. Bei den weiteren (nach den

15 Es ist zu beachten, dass eine lineare Kurve in diesem Diagramm ein logarithmisches Laufzeitverhalten darstellt.

technischen) Untersuchungen werden deshalb nur Baumgrößen von 640.000 und 1.280.000 Elementen betrachtet.

### 5.3.2 Verhältnisse der Messgrößen

Die Betrachtung der Verhältnisse der Messgrößen zueinander ist keine eigenständige Untersuchung, sondern soll vielmehr eine Basis für das Verständnis weiterer Untersuchungsergebnisse schaffen. Dazu wurde die gleiche Untersuchung verwendet,

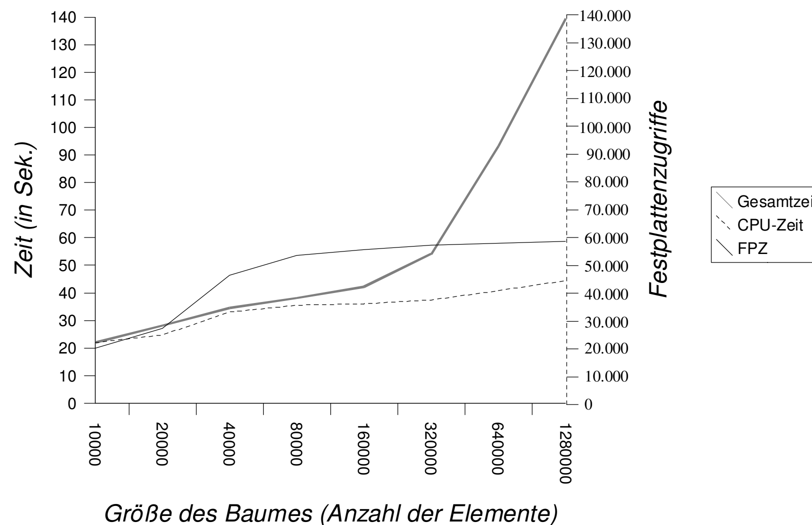


*Diagramm 4: Darstellung der verschiedene Messgrößen anhand eines String B-Trees mit Knotengrad 110*

welche Diagramm 2 zugrunde liegt. Das Diagramm wurde jedoch zusätzlich um eine Kurve erweitert, welche die Festplattenzugriffe darstellt. Aufgrund der unterschiedlichen Messeinheiten wurden auf der rechten Seite noch die Anzahl der Festplattenzugriffe hinzugefügt.

Wie man an dieser Kurve deutlich erkennen kann, wächst der String B-Tree zwischen 20.000 und 40.000 Elementen in der Höhe von 2 auf 3. Betrachtet man in diesem Abschnitt die Gesamtzeit und die CPU-Zeit, so merkt man, dass die Steigung der CPU-Zeit-Kurve in diesem Intervall nur gering und die Steigung der Gesamtzeitkurve praktisch kaum zunimmt. Weitere Untersuchungen, welche das Höhenwachstum von String B-Trees mit anderen Knotengraden betrachtet haben, ergaben ebenfalls nur eine leichte Zunahme der CPU-Zeit und der Gesamtzeitkurve an diesen Stellen.

Die "Treppenbildung", welche bei den Festplattenzugriffen deutlich zu erkennen ist, lässt sich also nur noch sehr schwach in den Gesamtlaufzeiten erkennen, was darauf hinweist, dass die Messung der Festplattenzugriffe (wie bereits vermutet) nur eingeschränkte Rückschlüsse auf die tatsächlichen Zugriffe und deren Zeitverbrauch zulässt.

### 5.3.3 Optimaler Knotengrad

Ziel dieser Untersuchung ist es, den optimalen Knotengrad und damit die optimale Knotengröße bezüglich der Implementierung und der verwendeten Plattform (Rechner und Betriebssystem) experimentell zu ermitteln, um somit die String B-Tree Struktur möglichst effizient zu nutzen.

Stellt man sich die Frage nach dem optimalen Knotengrad, so bemerkt man, dass diese nur mit der B-Baum Struktur zusammenhängt. Die Verknüpfung mit den Patricia Tries hat darauf keinen Einfluss und muss somit nicht berücksichtigt werden.

Da B-Bäume eine klassische und bereits ausführlich analysierte Datenstruktur sind, die heutzutage insbesondere bei Datenbanksystemen verwendet wird, existiert bereits eine Vielzahl an Literatur, welche sich mit B-Bäumen und deren Knotengrad auseinander gesetzt hat (vgl. [4,7]). Es wird grundsätzlich empfohlen, eine Knotengröße zu verwenden, die der Seitengröße des Hintergrundspeichers bzw. des Betriebssystems entspricht (wenn man nicht direkt auf den Hintergrundspeicher zugreift). Bezüglich des optimalen Knotengrades in B-Bäumen gibt es noch weitere theoretische Ansätze, welche mittels der Suchzeit und der Transferrate der Festplatte eine optimale Knotengröße herleiten können. Auf diese Ansätze soll jedoch nicht im weiteren eingegangen werden, da sie nur wenig praktische Aussagekraft besitzen.

Aufgrund der obigen Informationen wird bei den experimentellen Untersuchungen die ideale Knotengröße im Bereich der Seitengröße des Betriebssystems vermutet. Da ein String B-Tree Knoten im Prototyp einen Speicherplatz von *Knotengrad · 64 + 44 Byte* verbraucht, ergibt sich für die im Betriebssystem verwendete Seitengröße von 8192 Byte ein erwartetes Optimum bei einem Knotengrad von 127.

Bei der Untersuchung werden Knotengrade zwischen 20 und 500 betrachtet. Speziell im Bereich des erwarteten Optimums werden zur genaueren Betrachtung kleinere Schrittweiten verwendet. Das Optimum wird nur bis auf 10 Einheiten genau bestimmt, da der Umfang einer exakteren Untersuchung nicht gerechtfertig ist. Durch die unterschiedlichen Schrittweiten der Grade ist in Diagramm 5 zu beachten, dass die X-Achse nicht proportional ist.

Betrachtet man die Ergebnisse in Diagramm 5, so sieht man, dass die beiden Kurven zueinander leicht verschoben sind. Die leichte Linksverschiebung der Optima bei einer Baumgröße von 640.000 lässt sich auf den Cache zurückführen, weshalb im Folgenden nur die obere Kurve betrachtet wird.

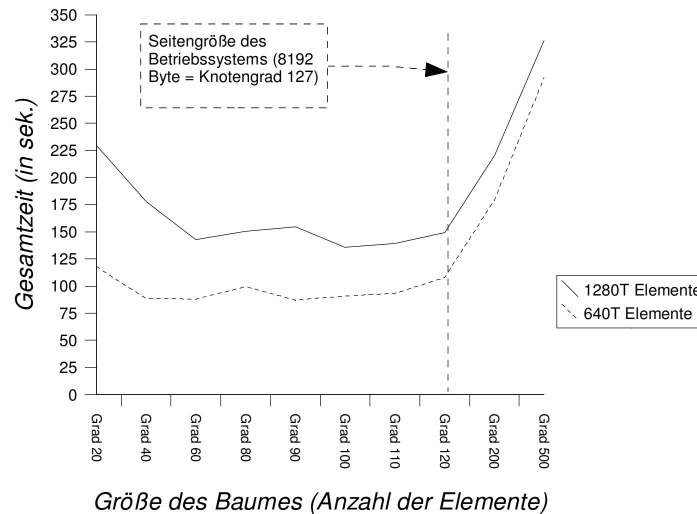


*Diagramm 5: Darstellung der Gesamtlaufzeiten von String B-Trees mit unterschiedlichen Graden (640.000 und 1.280.000 Elemente)*

Die obere Kurve besitzt zwei lokale Optima, welche sich bei den Knotengraden 100 (136 Sekunden) und 60 (142,6 Sekunden) befinden, wobei ersteres das Gesamtoptimum ist. Wider erwarten liegt das Optimum der Knotengröße (Grad 100 = 6444 Byte) leicht unterhalb der Seitengröße des Betriebssystems (8192 Byte). Der Grund dafür liegt wahrscheinlich in der Zuordnung der Speicherpakete des Betriebssystems. Es ist also nicht gewährleistet, dass ein Knoten immer genau einer Seite zugeordnet wird. Deshalb werden bei einer Knotengröße, welche genau der Seitengröße entspricht, bei einer leicht verschobenen Zuordnung gleich zwei Seiten für einen Knoten verwendet und dadurch mehr Zeit benötigt. Bei einer Knotengröße, die einen etwas kleineren Wert als die Seitengröße besitzt, ist die Wahrscheinlichkeit für eine solche Überlappung wesentlich geringer.

Dies ist wahrscheinlich auch der Grund, warum das zweite lokale Optimum ungefähr bei der Hälfte der Seitengröße (3884 Byte) liegt. In manchen Fällen können somit zwei Knoten in einer Seite untergebracht werden, wodurch die Ausführung beschleunigt wird.

Folglich müssten somit auch weitere lokale Optima bei einem Bruchteil bzw. einem Vielfachen der Seitengröße auftreten. Dies ist jedoch nicht zu beobachten. Das Fehlen dieser Kurveneigenschaft liegt wahrscheinlich einerseits an den zu großen Intervallen, in denen die Grade untersucht wurden, und andererseits an einer Vielzahl von betriebssysteminternen Einflussfaktoren, welche die Ergebnisse stark verändern.

Es bleibt festzuhalten, dass der optimale Knotengrad für den "*sbtree*" Prototyp für das verwendete Betriebssystem (Solaris) bei 100 (6444 Byte) liegt. Dieser Knotengrad wird für alle folgenden Experimente verwendet und deswegen vorausgesetzt. Wenn in

Ausnahmefällen ein anderer Knotengrad verwendet wird, ist dies entsprechend angegeben.

## 5.4 Ein- und Mehrworttextuntersuchungen

Im Gegensatz zu anderen Datenstrukturen (wie z.B. Suffix Trees) ist es dem String B-Tree möglich, sowohl Ein- als auch Mehrworttexte in einem Baum zu speichern. Bei einem Einworttext, also einer Textdatei, die nur ein Wort enthält, werden im String B-Tree alle Suffixe dieses (meistens sehr langen) Wortes gespeichert. In einem Mehrworttext werden alle Suffixe der einzelnen Worte im String B-Tree gespeichert. Typische Einworttexte sind z.B. Genomdaten oder natürlichsprachliche Texte, in denen gesucht werden soll. Praktische Beispiele für Mehrworttexte sind Wörterbücher, Zusammenfassungen von mehreren Texten (z.B. Gedichtbände) und Proteinsammlungen, welche eine Vielzahl von verschiedenen Proteinen beinhalten. Aufgrund dieser unterschiedlichen Textarten ergeben sich unterschiedliche Datenmengen. So ist in einem String B-Tree, der die Suffixe eines Mehrworttextes enthält, ein große Anzahl an gleich langen Suffixen enthalten. In einem String B-Tree, der die Suffixe eines Einworttextes enthält, kommt jede Suffixlänge nur einmal vor. Aufgrund dieser unterschiedlichen Eigenschaften der eingefügten Suffixe könnten innerhalb des String B-Trees und in den Patricia Tries unterschiedliche Strukturen entstehen, welche sich auf die Laufzeit auswirken.

Ziel der nun folgenden Untersuchung ist es herauszufinden, inwieweit sich dieser Unterschied zwischen Ein- und Mehrworttexten auf das Laufzeitverhalten der Substringsuche in den String B-Trees auswirkt.

Um das zu untersuchende Verhalten für möglichst klare Ergebnisse so weit wie möglich zu isolieren, müssen die Suchoperation auf beiden Texten so ähnlich wie möglich gestaltet werden. Dazu werden zwei künstlich erstellte Texte verwendet, die beide eine Alphabetgröße von 26 besitzen, deren Buchstaben unabhängig und identisch verteilt sind.

Weiterhin muss die Länge des Suchmusters im Einworttext beschränkt werden. Wie bereits erwähnt, werden die Suchmuster aus der Datei selbst genommen und sind somit im Baum enthalten. Dadurch ergibt sich mindestens bei dem Patricia Trie im Blatt des String B-Trees beim Vergleich ein komplettes Durchlaufen des gesamten Suchmusters, da dieses ja ebenfalls im Patricia Trie enthalten ist und somit zwei gleiche Worte verglichen werden. In den bisher verwendeten Mehrworttexten waren die Worte in ihrer Länge auf 25 begrenzt und somit auch deren Suchmuster. Da der Text beim Bestimmen des Suchmusters vom Anfang und dann linear durchlaufen wird, entsprechen die Längen der Suchstrings in Einworttexten meistens fast der gesamten Länge des Textes. Ein komplettes Durchlaufen dieser Strings in jeder Suchoperation würde das gesamte Ergebnis extrem verschlechtern und dadurch einen sinnvollen Vergleich mit dem Mehrworttext verhindern. Aus diesem Grund wird die Suchmusterlänge im Einworttext beschränkt. Eine genauere Betrachtung der Auswirkungen der Suchmusterlänge findet sich im zweiten Teil dieser Untersuchung.

Ein letzter zu berücksichtigender Aspekt ist die Gleichsetzung der Schrittlänge bei der Suchmusterbestimmung. Um eine gleiche Suchmusterlänge zu gewährleisten, muss im Mehrworttext immer wortweise weitergesprungen werden. Im Einworttext ist das nicht nötig, wird jedoch angepasst, da sonst die Nutzung des Datenpuffers für die Suchmuster unterschiedlich wäre.

Nachdem alle Möglichkeiten für die gleichen Voraussetzungen der Suchoperation erklärt wurden, soll nun die Betrachtung der Untersuchungsergebnisse folgen. Zuerst werden die Gesamtzeiten verglichen und danach die CPU-Zeiten und Festplattenzugriffe für eine genauere Beurteilung herangezogen. Es werden jeweils Bäume mit 640.000 bzw. 1.280.000 Elementen betrachtet. Durch eine andere Suchmusterlänge als bei den bisherigen Untersuchungen ist zu beachten, dass die Ergebnisse nicht mit den vorangegangenen verglichen werden können.

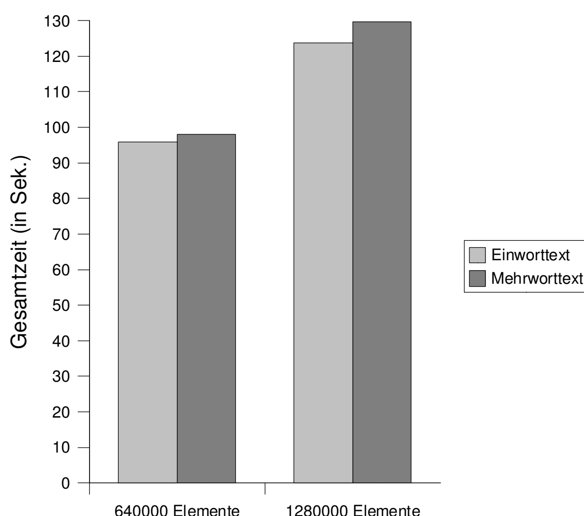


*Diagramm 6: Vergleich der Gesamtzeiten von 10.000 Suchoperationen auf Ein- und Mehrworttexten*

Betrachtet man das Diagramm 6, so stellt man auf den ersten Blick einen erkennbaren, wenn auch nicht gravierenden Unterschied zwischen der Suche in den verschiedenen Textarten fest. Nimmt man die genauen Zahlenwerte zu Hilfe, so ergibt sich bei einer Baumgröße von 640.000 Elementen eine um 2,3 Sekunden (2,4 %) und bei 1.280.000 Elementen eine um 5,8 Sekunden (4,7 %) schnellere Ausführung der Suche bei einem Einworttext.

Eine Differenz von 2,4% bzw. 4,7% kann als relativ gering angesehen werden. Betrachtet man zum Vergleich den Gesamtzeitunterschied von String B-Trees mit Knotengrad 90 und 100, so ergibt sich dabei bereits ein Unterschied von 13%. Trotzdem soll erörtert werden, wie dieser Unterschied zustande kommt. Dabei gibt es zwei Bereiche, welche untersucht werden müssen. Erstens die CPU-Zeit, die hauptsächlich Auskunft über die

Implementierung und deren Algorithmen gibt. Zweitens die Zahl der Festplattenzugriffe, die ein Indikator für Änderungen im Aufbau der Datenstruktur an sich ist. Da diese beiden Größen ebenfalls gemessen wurden, sollen deren Messergebnisse im Folgenden dazu benutzt werden, die Ursachen zu finden.

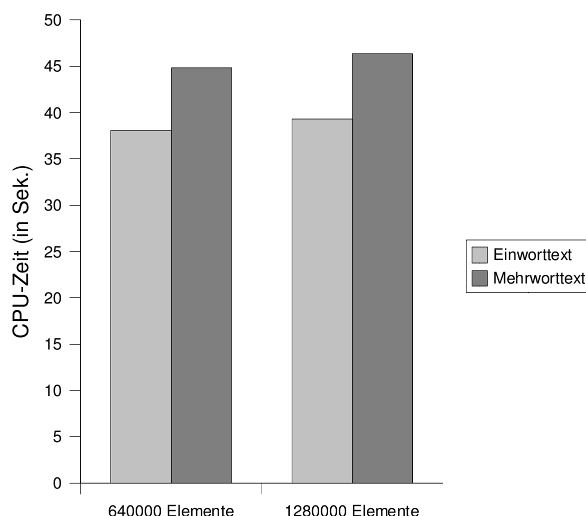


*Diagramm 7: Vergleich der CPU-Zeiten von 10.000 Suchoperationen auf Ein- und Mehrworttexten*

Bei der Betrachtung der Ergebnisse der CPU-Zeit-Messung in Diagramm 7 stellt man fest, dass das gleiche Verhalten wie bei der Gesamtzeitmessung auftritt; sogar noch ein wenig deutlicher. Diesmal beträgt der Unterschied sogar 6,8 Sekunden (640.000 Elemente) bzw. 7,1 Sekunden (1.280.000 Elemente). Auffällig dabei ist, dass die Differenz anscheinend nicht von der Größe der Bäume abhängt, sondern (fast) gleich bleibt.

Die Ursache für die unterschiedlichen Zeiten liegt also eindeutig in der Implementierung. Durch eine differenziertere Laufzeitmessung einzelner Funktionen der Implementierung konnte der dafür verantwortliche Programmteil genauer bestimmt werden.

Die eigentliche Verzögerung liegt in der Funktion *word_length,* welche die Wortlänge eines Wortes (bzw. Suffixes) bestimmt. Diese Größe ist in der Patricia Trie Suche notwendig, um zu überprüfen, ob ein Vergleich abgebrochen werden muss, wenn bereits das Wortende erreicht ist. Für jede einzelne Suchoperation auf dem String B-Tree wird die Funktion *word_length* einmal für das Suchmuster und je nach Höhe des Baumes meist mehrmals für das *lcp-Wort* verwendet. Bei den durchgeführten 10.000 Suchoperationen und einer Baumhöhe von 3 ergeben sich somit 40.000 Aufrufe der Funktion *word_length.* Der Unterschied der Funktion *word_length* für einen Ein- bzw. Mehrworttext besteht nun darin, dass in einem Mehrworttext jedes Wort einmal komplett durchlaufen werden muss, um seine Wortlänge zu bestimmen. In einem Einworttext ist dies nicht notwendig, da

durch die Gesamtlänge der Stringdatei für jedes Wort (bzw. Suffix) in der Datei die Länge sofort aus der Position des Wortanfangs berechnet werden kann.

Bevor eine abschließende Auswertung erfolgt, sollen zur Kontrolle noch einmal die Messergebnisse der Festplattenzugriffe betrachtet werden.

In Diagramm 8 bestätigt sich noch einmal die Vermutung, dass die Differenz der Gesamtlaufzeit ausschließlich von den unterschiedlichen CPU-Zeiten verursacht wird. Berücksichtigt man die durch die zufällige Verteilung der Buchstaben der Texte (vgl. Abschnitt 2.1) entstehenden Schwankungen, so kann man die Anzahl der Festplattenzugriffe im Prinzip als identisch betrachten.

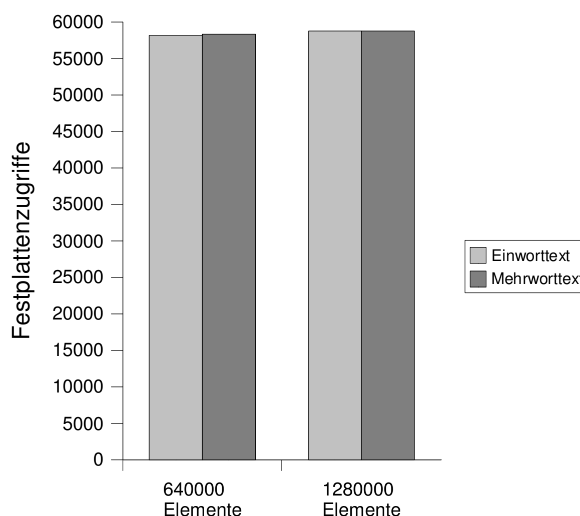


*Diagramm 8: Vergleich der Festplattenzugriffe von 10.000 Suchoperationen auf Ein- und Mehrworttexten*

Vergleicht man nun die Differenzen der CPU-Zeiten und der Gesamtzeit und berücksichtigt dabei deren hohe Varianz, welche das Bild bei den CPU-Zeiten ein wenig verzerrter wiedergibt, so ist klar, dass die unterschiedliche Gesamtlaufzeit bei der Suche in Ein- und Mehrworttexten nur auf eine implementierungsspezifische Ursache (die Funktion *word_length*) und nicht auf die Struktur des String B-Trees zurückzuführen ist. Die Struktur des String B-Trees ist also weitestgehend unabhängig von der Wortanzahl des Textes.

### *Bemerkung*

Natürlich stellt sich für die Implementierung die Frage, ob sich die unterschiedliche Laufzeit der Funktion *word_length* vermeiden lässt. Grundsätzlich gibt es die Möglichkeit, die Information der Wortlänge zusätzlich mit anzugeben. Für die Umsetzung wären verschiedene Lösungen denkbar, so z.B. die Codierung der Wortlänge im Text selbst,

oder in einer weiteren Datei, in der die einzelnen Wortlängen gespeichert wären. Jede dieser Varianten wirft jedoch wieder weitere Probleme (Vorbehandlung und Verlängerung des Textes, Zugriff und Suche in einer weiteren Datei) auf, die ausführlich betrachtet werden müssten, um genauere Aussagen darüber zu treffen.

### 5.4.1 Auswirkungen der Suchmusterlänge

Wie im ersten Teil dieser Untersuchung bereits erwähnt, sollen an dieser Stelle noch einmal die Auswirkungen der Suchmusterlänge bei einem im Text enthaltenen Suchmuster genauer untersucht werden. Dies dient zur genaueren Betrachtung der Eigenschaften der Implementierung, welche für die Analyse in späteren Untersuchungen eine wichtige Rolle spielen.

Um die Länge des Suchmusters besser bewerten zu können, soll an dieser Stelle kurz auf die möglichen Anwendungen eingegangen werden. Betrachtet man die beiden für diese Arbeit relevanten Bereiche, so ergeben sich Beispiele wie die Suche von DNA-Strängen und Proteinteilen oder die Suche von Textabschnitten in großen Textcorpora. Erfahrungswerte aus der Praxis besagen für die Suche in DNA-Strängen und Proteindatenbanken, dass die meisten Muster eine Länge von 100 Zeichen nicht überschreiten. Trotzdem kann in manchen Fällen nach wesentlich längeren Mustern gesucht werden, so z.B. wenn ein großes Protein (bis zu 10.000 Zeichen) als Muster benutzt wird.

Auch im Information Retrieval werden häufig kürzere Suchmuster verwendet, aber auch hier gibt es Anwendungsfälle (z.B. die Plagiatsuche längerer Textpassagen), in denen die Musterlänge 10.000 Zeichen erreichen kann.

Der Grund für die Auswirkungen der Suchmusterlänge auf die Laufzeit liegt in der Suche im Patricia Trie. In der zweiten Phase der Suche muss das Suchmuster mit dem *lcp-Wort* zeichenweise verglichen werden, um die Fehlstelle festzustellen. Im schlechtesten Fall sind beide Worte gleich, so dass sich für diese Operation eine Laufzeit von O(Musterlänge) ergibt[16]. Obwohl sich bei den Untersuchungen der String B-Trees die CPU-Zeit normalerweise nur zu einem geringeren Teil auf die Gesamtlaufzeit auswirkt, ist sie bei einem sehr langen (vorkommenden) Suchmuster die ausschlaggebende Größe.

Dieses ist an dem Ergebnis der Messung in Diagramm 9 deutlich zu erkennen. In diesem sind die Ergebnisse von 7 Messungen (Musterlänge 125 bis 20.000) an einem Einworttext dargestellt. Die Differenz zwischen CPU- und Gesamtzeit liegt bei allen Messpunkten bei nahezu 100 Sekunden, so dass die CPU-Zeit ab einer Patternlänge von ca. 500 Zeichen den größeren Anteil an der Gesamtzeit darstellt. Weiterhin ist das bereits oben erwähnte lineare Laufzeitverhalten zu erkennen.

16 In diesem Falle ist die O-Notation (im Gegensatz zu allen anderen im Text) nicht auf das Modell der Hintergrundspeicherzugriffe bezogen, sondern auf die Zahl der Rechenoperationen.

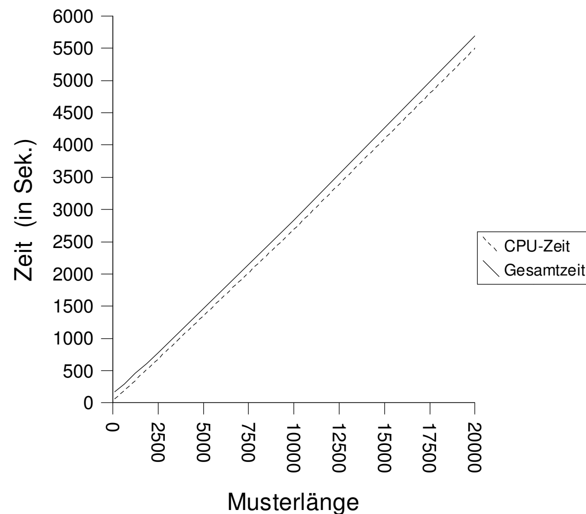


*Diagramm 9: Gesamt- und CPU-Zeit bei 10.000 Suchen auf einem String B-Tree (1.280.000 Elemente) mit verschiedenen Suchmusterlängen*

Es hat sich also gezeigt, dass auch die Betrachtung der CPU-Zeit im Hinblick auf die Musterlänge eine entscheidende Rolle spielt und nicht zu vernachlässigen ist. Trotzdem sind sehr große Musterlängen, wie sie hier getestet wurden, in der Praxis eher als Ausnahme zu betrachten.

## 5.5 Untersuchungen der Alphabetgröße

Im Information Retrieval und in Gendatenbanken besitzen die Daten eine große Bandbreite an Alphabeten und Alphabetgrößen. Beispiele dafür gibt es viele; angefangen von DNA-Daten mit einem 4-elementigen Alphabet bis hin zu Programmiertexten mit einer Alphabetgröße von mehr als 100 Zeichen. Es stellt sich nun die Frage, inwieweit sich diese unterschiedlichen Größen auf die String B-Tree Implementierung auswirken. Durch die einheitliche Binärcodierung der Alphabete in den Patricia Tries dürfte sich dies nicht auf die Laufzeit auswirken (vgl. Abschnitt 3.4.2). Ziel der nun folgenden Untersuchung ist es, diese Behauptung experimentell zu überprüfen und festzustellen, ob die Laufzeiten der Substringsuche in den String B-Trees unabhängig von der Alphabetgröße der verwendeten Eingabedaten sind.

Zwecks isolierter Betrachtung der Alphabetgröße werden, wie schon bereits bei anderen Untersuchungen, künstlich erstellte Texte verwendet. Diese Einworttexte unterscheiden sich nur in der Größe der zugrunde liegenden Alphabete. Die Zeichen sind in allen Texten unabhängig und identisch verteilt. Es wurden Alphabetgrößen aller Zweierpotenzen von 2 -127 (7 Bit-Text) in exponentiellen Schritten untersucht.

Betrachtet man die Gesamtlaufzeiten in Diagramm 10, so stellt man sowohl Unterschiede in den Gesamtlaufzeiten der Suchen, als auch Unterschiede zwischen den Gesamtlaufzeiten der verschiedenen Baumgrößen fest. Bei der Alphabetgröße von 4 zum Beispiel ergibt sich bei einer Baumgröße von 640.000 Elementen der niedrigste und bei

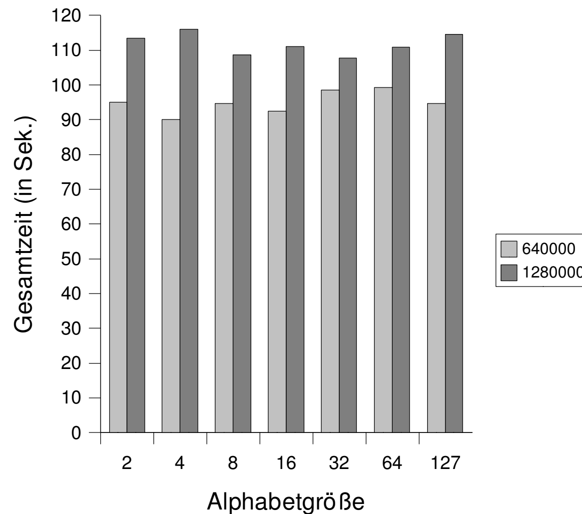


*Diagramm 10: Gesamtlaufzeiten von 10.000 Suchoperationen in String B-Trees von Stringdateien mit verschiedenen Alphabetgrößen*

einer Größe von 1.280.000 Elementen der höchste Gesamtzeitwert der Untersuchung. Weiterhin sind weder lokale noch absolute Extrema oder andere Muster deutlich zu erkennen, welche gerechtfertigte Rückschlüsse auf einen Zusammenhang zwischen Alphabetgröße und Gesamtlaufzeit zulassen. Das deutet darauf hin, dass die Unterschiede in den Ergebnissen vielmehr durch die Einwirkung verschiedener Störfaktoren, wie z.B. den Cache, die Lage der Datei auf der Festplatte oder die Verteilung der Zugriffe auf die Datei hervorgerufen werden.

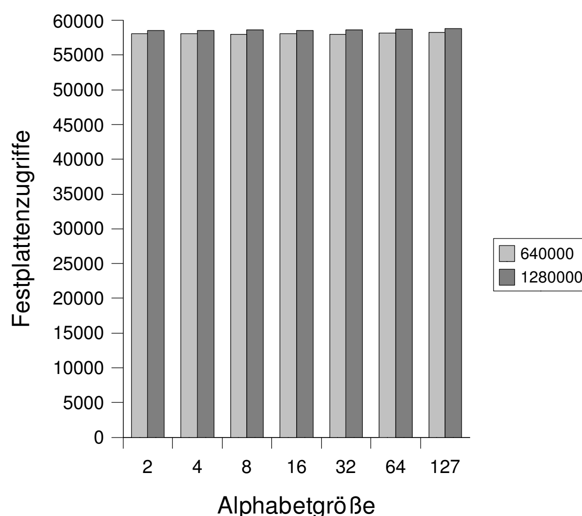


*Diagramm 11: Anzahl der Festplattenzugriffe von 10.000 Suchoperationen bei verschiedenen Alphabetgrößen*

Die Betrachtung der Messergebnisse der Festplattenzugriffe in Diagramm 11 verstärken diese Vermutung. Zieht man die durch die Zufallsverteilung hervorgerufenen geringen Schwankungen in Betracht, so sind die Messergebnisse als praktisch gleich anzusehen. Somit kann ein Zusammenhang zwischen den Festplattenzugriffen und der Gesamtzeit ausgeschlossen werden.

Sieht man die Ergebnisse der CPU-Zeit-Messungen in Diagramm 12 an, so stellt man fest, dass die Ergebnisse der Gesamtzeitmessung auch nicht auf die CPU-Laufzeiten zurückzuführen sind.

Es ist jedoch zu bemerken, dass sich bei zunehmender Alphabetgröße die Laufzeit verringert. Der Grund dafür liegt in der von der Implementierung verwendeten festen Codierungsgröße von 8 Bit. Wie bereits in Kapitel 3 erwähnt, ist diese feste Codierung sehr praktisch und mit schnellen Zugriffen realisierbar, da ein Byte eine typische Zugriffsgröße darstellt. Für kleinere Alphabete müssen jedoch mehr Bits verwendet werden als eigentlich gebraucht würden. Wie bereits erwähnt, stellt dies für die Verwaltung in den Patricia Tries keinen Unterschied dar, da gleiche Teile zusammengefasst werden. In der Verifikationsphase der Patricia-Trie-Suche wird jedoch das Suchmuster bitweise mit dem *lcp-Wort* verglichen, wodurch unnötige Operationen entstehen[17].

Trotz dieses zusätzlichen Aufwandes ist die resultierende Differenz der CPU-Zeit nur gering. Selbst für die Ergebnisse der kleinsten (2) und der größten (127) Alphabetgröße

17 Eine Veränderung der Implementierung wurde bereits während der Untersuchungen vorgenommen. Es wird nun zuerst buchstabenweise und erst an der Fehlstelle bitweise verglichen, was den Laufzeitunterschied bei verschiedenen Alphabeten wesentlich reduziert.

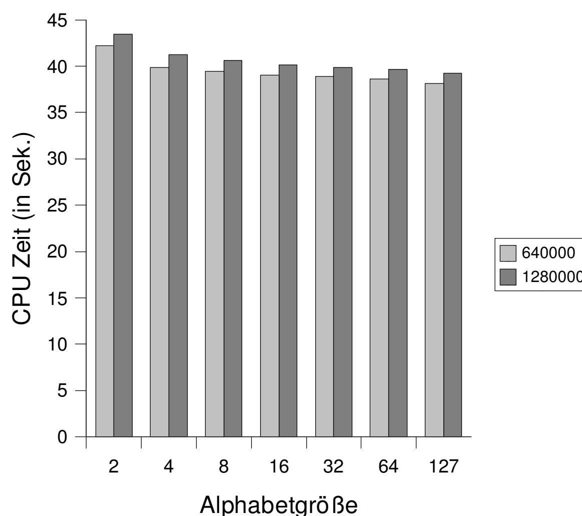


*Diagramm 12: CPU-Zeiten von 10.000 Suchoperationen in String B-Trees von Stringdateien mit verschiedenen Alphabetgrößen*

ergibt sich nur eine Differenz von 10% (bei 1.280.000 Elementen).

Für die beobachteten Unterschiede in der Gesamtlaufzeit bei verschiedenen Alphabetgrößen können also sowohl die CPU-Zeit als auch die Anzahl der Festplattenzugriffe als Ursache ausgeschlossen werden. Zusammen mit der Beobachtung von den Inkonsistenzen der Gesamtlaufzeiten bei unterschiedlichen Baumgrößen lässt sich mit großer Wahrscheinlichkeit behaupten, dass die Schwankungen nicht durch die Implementierung oder die Datenstruktur, sondern durch Einflüsse beim Zugriff auf den Hintergrundspeicher hervorgerufen wurden. Dieselbe Untersuchung mit anderen Texten gleicher Art (Alphabetgröße, Verteilung der Buchstaben) würde höchstwahrscheinlich die gleichen Schwankungen in anderer Ausprägung besitzen aufweisen.

Die Annahme, dass der String B-Tree durch die Verwendung von binären Patricia Tries unabhängig von der Alphabetgröße des Eingabetextes ist, hat sich also bestätigt. Nur die Implementierung zeigt durch die Verwendung einer konstanten Codierungsgröße eine geringe Abhängigkeit von der Alphabetgröße.

## 5.6 Untersuchungen verschiedener Texte

Durch die Verwendung von realen Datenmengen in dieser Untersuchung ist diese am ehesten mit den praktischen Anwendungsfällen vergleichbar. Dabei werden die zuvor untersuchten Einflussgrößen nicht mehr isoliert, sondern im Zusammenspiel betrachtet. Diese Untersuchung stellt somit einen zusammenfassenden Abschluss der Untersuchungen der Einflussgrößen dar.

### 5.6.1 Vorbemerkungen

Da sich diese Untersuchung durch die Verwendung von realen Daten in vielen Aspekten von den bisherigen unterscheidet, sollen zu Beginn dieser Untersuchung einige Erläuterungen gegeben werden, um eine bessere Grundlage für die Analyse der Ergebnisse zu schaffen.

Bei der Auswahl der Beispieldaten wurden speziell die zu untersuchenden Gebiete der Genom-Datenbanken und des Information Retrieval berücksichtigt. Aufgrund der Vielseitigkeit der verwendeten Daten sollen diese zuerst kurz erläutert werden:

- **DNA-Daten**
  Die verwendeten DNA-Daten sind das komplette Genom vom *E. coli* Bakterium. Das Alphabet besteht aus den vier Buchstaben A,C,G,T welche gewichtet verteilt sind. Dabei kommen jeweils G und C bzw. A und T gleich häufig vor, da sie jeweils ein Paar auf der DNA Helix bilden. Die DNA-Daten sind ein Einworttext.
- **Proteindaten**
  Die in der Untersuchung verwendeten Proteindaten sind eine Sammlung von vielen einzelnen Proteinen im FASTA-Format[18]. Die Proteinsammlung ist somit ein Mehrworttext, dessen einzelne Worte die Proteine sind. Die Worte (Proteine) haben ein Länge von 10 bis ca. 10.000 Zeichen, die aus einem 20-elementigen Alphabet stammen. Die Zeichen sind gewichtet verteilt. Zwischen vielen Proteinen besteht ein hoher Grad an Ähnlichkeit und z.T. sogar Gleichheit, da gleiche Proteine aus unterschiedlichen Gründen oft mehrfach vorkommen.
- **CIA World Fact Book**
  Das CIA World Fact Book ist ein natürlichsprachlicher Text, welcher die Eckdaten sämtlicher Länder der Welt beinhaltet. Der Text ist in Englisch und das verwendete Alphabet ist ca. 80 Zeichen (Groß- und Kleinbuchstaben plus Satz und Sonderzeichen) groß und die Zeichen sind gewichtet verteilt. Der Text wird in den Untersuchungen als ein Einworttext behandelt.
- **Bibeltexte (Deutsch, Englisch, Französisch)**
  Um einen Eindruck zu bekommen, wie sich die Verteilung der Zeichen in verschiedenen Sprachen auf die String B-Tree Implementierung auswirken, wurde als Grundlage der Bibeltext (altes und neues Testament) genommen und drei unterschiedliche Übersetzungen (Deutsch: Luther, Englisch: King James, Französisch: Louis Segond) untersucht. Dabei wird nicht nur der Unterschied der Sprache allein untersucht. Es wirken sich auch andere Faktoren (Art der Übersetzung, Musterwiederholungen) auf das Ergebnis aus. Davon abgesehen ähneln sich diese Texte trotzdem mehr als die anderen. Somit ist es interessant zu betrachten, ob sich diese Ähnlichkeiten in den Laufzeitmessungen wiederspiegeln.
  Die Alphabetgröße beträgt ebenfalls ca. 80 Zeichen und die Zeichen sind (wie in jedem natürlichsprachlichen Text) gewichtet verteilt. Die Bibeltexte werden jeweils als

18 Ein in der Genforschung häufig verwendetes Textformat für DNA- und Proteindaten. (vgl. [3])

Einworttext behandelt.

Es werden bei den Untersuchungen wieder String B-Trees mit einer Größe von 640.000 und 1.280.000 Elementen durchsucht. Im Gegensatz zu den bisherigen Untersuchungen wird in dem untersuchten Mehrworttext beim Laden der Suchmuster nicht (wie bisher) wortweise, sondern zeichenweise (wie im Einworttext) weitergesprungen, da es sich um wesentlich längere Worte (bis zu 10.000 Zeichen) als bei den bisher verwendeten Texten handelt und somit der Datenpuffer öfter gefüllt werden müsste, was wiederum eine Verzerrung der Messergebnisse zur Folge hätte.

Durch die Verwendung von realen Daten ergibt sich im Gegensatz zu den bisherigen künstlich erstellten Texten eine gewichtete Verteilung der Buchstaben und der Muster in den Texten. Sowohl in den Gendaten als auch in den natürlichsprachlichen Texten sind eine Vielzahl sich wiederholender Muster enthalten. Gerade diese Muster sind in der Genomforschung der Hauptaspekt: Gleiche Muster (Abschnitte) in Genen weisen auf eine gleiche Funktion der daraus resultierenden Proteine hin. Ist die Funktion eines Proteins bekannt, so kann dessen Aufbau mit anderen unbekannten Daten verglichen werden, um somit eventuell Rückschlüsse auf deren Funktion zu ziehen. In den natürlichsprachlichen Texten sind die Muster ganz einfach durch Worte, Satzteile oder Phrasen im Text gegeben.

Wie bereits beschrieben, werden die Suchmuster aus den Texten selber gelesen, sind also im Baum bereits enthalten. Im Gegensatz zu den künstlichen Texten hat die Wahl der Suchmuster durch die gewichtete Verteilung der Buchstaben und der Muster jedoch eine stärkere Auswirkung auf das Laufzeitverhalten. Durch diese Eigenschaft der realen Daten sind bei den Messergebnissen größere Varianzen zu erwarten als bei den vorangegangenen Untersuchungen.

### 5.6.2 Untersuchungsergebnisse und Analyse

Wie schon in den anderen Untersuchungen soll zuerst die Gesamtzeit betrachtet werden und später die Ergebnisse der CPU-Zeit und der Festplattenzugriffe zu Hilfe genommen werden.

Betrachtet man die Ergebnisse in Diagramm 13, so stellt man fest, dass die Suche auf den beiden Texte mit den Gendaten mehr Zeit verbraucht als bei den natürlichsprachlichen Texten. Dabei ist die Gesamtzeit bei den DNA-Daten in etwa 20% höher und bei den Proteindaten sogar mehr als doppelt so hoch. Die Hauptursache ist dabei wahrscheinlich die hohe Repetitivität beider Texte. Bevor jedoch weitere Schlussfolgerungen gezogen werden, sollen zuerst die anderen Messdaten betrachtet werden.

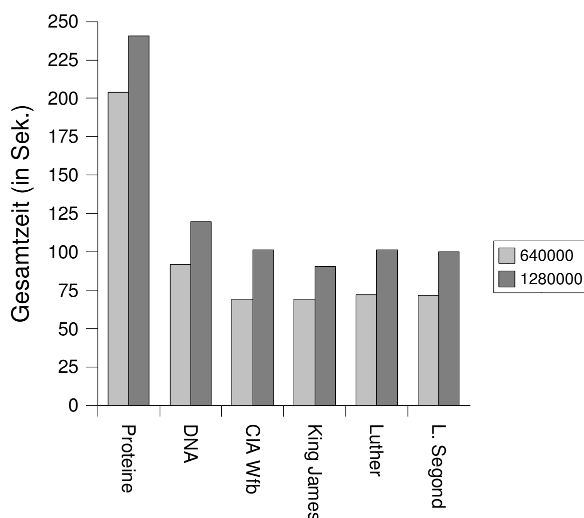


*Diagramm 13: Gesamtzeit für 10.000 Suchoperationen in verschiedenen Texten*

Auch bei der CPU-Zeit Betrachtung in Diagramm 14 zeichnet sich ein ähnliches Bild wie bei den Gesamtzeitmessungen ab. Die Werte des Proteintextes sind wieder wesentlich höher (Faktor größer 3) als die der anderen Texte. Im Diagramm (durch den hohen Wert beim Proteintext) schlecht zu erkennen ist der ebenfalls etwas höhere CPU-Zeit-Wert bei den DNA-Daten, dessen Differenz zu den natürlichsprachlichen Texten sowohl bei 640.000 Elementen als auch bei 1.280.000 Elementen jeweils ungefähr 2 Sekunden (ca. 5%) beträgt.

Bei der Betrachtung der Festplattenzugriffe in Diagramm 15 liegt die Differenz zwischen den DNA-Daten und den natürlichsprachlichen Texten nur noch bei ungefähr 2% (ca. 1000 FPZ). Da die Unterschiede zwischen den natürlichsprachlichen Texten in der gleichen Größenordnung liegen (2% - 3%), ist die eben genannte Differenz zu den DNA-Daten gerade an der Toleranzschwelle und somit nur eingeschränkt für die Schlussfolgerung zu verwenden. Die Messwerte der Suche in den Proteindaten sind auch in diesem Fall wieder deutlich (wenn auch nicht so deutlich wie vorher) höher. Interessanterweise ist die Anzahl der Festplattenzugriffe bei den Proteindaten in einem Baum mit 1.280.000 Elementen etwas geringer als in einem Baum mit 640.000 Elementen, worauf am Ende dieses Abschnittes eingegangen wird.

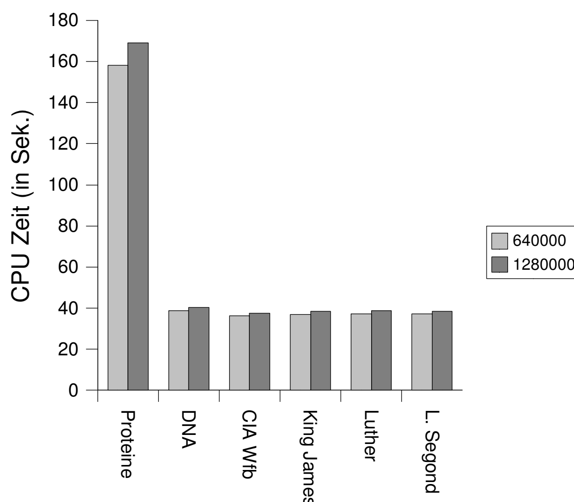


*Diagramm 14: CPU-Zeit für 10.000 Suchoperationen in verschiedenen Texten*

Nachdem alle Messgrößen untersucht wurden, ist zusammenfassend folgendes festzuhalten: Die Suchen in den Protein- und DNA-Daten weisen höhere Laufzeiten auf, wobei zusätzlich eine große Differenz wischen den Daten (DNA, Proteine) selbst besteht. Diese Unterschiede spiegeln sich ebenfalls in den CPU-Zeiten und abgeschwächt in den Festplattenzugriffen wieder. Die Unterschiede zwischen den natürlichsprachlichen Texten sind nur gering und in ihrer Größenordnung im Bereich der Toleranzen, die sich aus den verschiedenen Einflussgrößen (Lage der Datei, Verteilung der Zeichen und Muster des Textes) ergeben. Aus diesem Grund wird dieser Aspekt bei der folgenden Analyse nicht weiter betrachtet, sondern nur auf das Verhältnis zwischen den Gendaten und den natürlichsprachlichen Texten eingegangen.

Wie bereits bei der Betrachtung der Gesamtzeit kurz erwähnt, sind Gendaten sehr repetitiv und deswegen auch allgemein bekannt dafür, dass sie im Gegensatz zu anderen Testdaten oft schlechtere Laufzeiten verursachen. Auch bei dieser Untersuchung scheint dies die Ursache für die schlechten Ergebnisse bei den Gendaten zu sein. Um den konkreten Zusammenhang zwischen der hohen Repetitivität und der schlechten Laufzeit herzustellen hilft die Betrachtung der CPU-Zeit: Die großen Differenzen bei den Ergebnissen der CPU-Zeit-Messungen sind ein Indikator dafür, dass in den String B-Trees der Gendaten anscheinend längere Teile der Strings miteinander verglichen werden als in den String B-Trees der anderen Texte (vgl. Abschnitt 5.4.1).

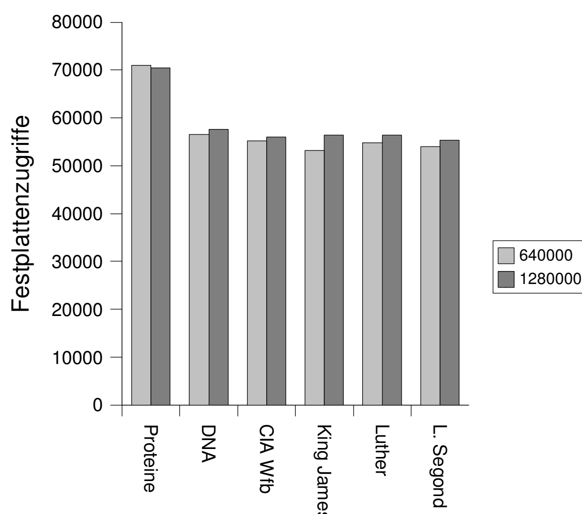


*Diagramm 15: Festplattenzugriffe für 10.000 Suchoperationen in verschiedenen Texten*

Es stellt sich jedoch die Frage, warum längere Vergleiche stattfinden, wenn die Suchmuster bei allen Suchoperationen die gleiche Länge besitzen. Die Antwort auf diese Frage liegt in den Vergleichen, die in den inneren Knoten des String B-Trees stattfinden. Natürlich findet in den Blättern des String B-Trees immer ein kompletter Vergleich des Suchmusters statt, da es im Baum enthalten ist. Aber auch in den inneren Knoten wird das Suchmuster jeweils mit dem *lcp-Wort* zeichenweise verglichen. In den meisten Fällen stimmt dabei nur ein kurzer Präfix überein und somit wird der Vergleich frühzeitig abgebrochen. Bei den Gendaten jedoch, welche einen hohen Anteil an gleichen Mustern besitzen, ist der gemeinsamen Präfix von Suchmuster und *lcp-Wort* bereits in den inneren Knoten oft sehr lang, so dass bereits dort eine große Anzahl an zeichenweisen Vergleichen stattfindet. Trotz gleicher Suchmusterlänge und einem Suchmuster, das im Baum enthalten ist, kann also die Anzahl der durchgeführten (zeichenweisen) Vergleiche, bei einer Suchoperation in Bäumen mit verschiedenen Texten, sehr unterschiedlich sein. Dies wirkt sich auch dementsprechend auf die CPU-Zeit aus.

Durch die längeren Vergleiche bei den Gendaten ist jedoch nicht nur die CPU-Zeit beeinflusst, sondern auch die Zahl der Festplattenzugriffe. Durch die große Länge der zu vergleichenden Worte ist die Wahrscheinlichkeit, dass ein Vergleich über die Länge des Datenpuffers hinausgeht auch größer als bei Texten mit kürzeren Vergleichen. Dabei ist anzumerken, dass der Datenpuffer ein festes Blockraster auf der Textdatei besitzt. Das Wort liegt also beim Laden des Datenpuffers nicht unbedingt am Anfang, sondern an einer beliebigen Stelle des Puffers. Durch dieses häufigere Laden des Datenpuffers sind die erhöhten Werte bei der Messung der Festplattenzugriffe zu erklären. Die Tatsache, dass diese Unterschiede bei den Festplattenzugriffen weniger ausgeprägt sind als bei der CPU-

Zeit, lässt sich damit ebenfalls erklären. Nicht jeder Vergleich über einen längeren Abschnitt zweier Strings erstreckt sich über zwei Datenpufferblöcke.

Es lässt sich also feststellen, dass die hohen Werte der CPU-Zeit und der Festplattenzugriffe und die daraus resultierende Gesamtzeit (mit Vernachlässigung der Varianzen) der Suche auf den Gendaten, auf deren hohe Repetitivität zurückzuführen sind.

Es lässt sich somit für diese Untersuchung an verschiedenen Texten, welche sowohl aus dem Bereich der Gendatenbanken als auch aus dem Bereich des Information Retrieval kommen, zusammenfassend festhalten, dass die String B-Tree Implementierung nur bei Texten mit hoher Repetitivität ein schlechteres Laufzeitverhalten zeigt und ansonsten weitestgehend unabhängig von der Struktur des zugrunde liegenden Textes ist.

### *Analyse der Anomalie bei Proteintexten*

Nachdem die Ursachen für die Laufzeitunterschiede zwischen den Gendaten und den natürlichsprachlichen Texten analysiert wurden, werden nun die vorhin erwähnten ungewöhnlichen Ergebnisse bei der Messung der Festplattenzugriffe (bei den Proteindaten) untersucht. Die beobachtete Anomalie bei den Festplattenzugriffen bezüglich der String B-Tree Größe bei der Suchoperation des Proteintextes lässt sich wahrscheinlich auf die unterschiedliche Anordnung der Elemente in den Bäumen zurückführen. Im allgemeinen nehmen die Festplattenzugriffe mit wachsender Zahl der enthaltenen Elemente zu. Dies ist nicht nur der Fall, wenn der Baum in der Höhe wächst, sondern auch wenn die Höhe des Baumes gleich bleibt. Das liegt zum einen an den hinzukommenden Knoten innerhalb des Baumes, welche die Baumdatei vergrößern, so dass ein größerer Teil der Datei durchsucht werden muss und zum anderen an den höheren Füllungsgraden in den Knoten des String B-Trees, wodurch längerere Vergleiche mit dem *lcp-Wort* durchgeführt werden müssen. Diese beiden Faktoren führen meistens zu einem häufigeren Laden der Datenpuffer und somit zu mehr Festplattenzugriffen. Das muss jedoch nicht immer so sein. Durch die Struktur des Textes kann der Fall auftreten, dass durch das Einfügen weiterer Elemente die Umstrukturierung des Baumes sehr gutartig sein kann und somit bei einigen Suchoperation gleich viel oder sogar etwas weniger Festplattenzugriffe als vorher benötigt werden. Dabei sind nur relativ geringe Abweichungen erklärbar, wie sie bei den String B-Trees des Proteintextes der Fall sind. Größere Differenzen würden sich mit dieser Theorie nicht erklären lassen. Zusätzlich besitzt der Proteintext die Eigenschaft, sehr inhomogen zu sein, da die einzelnen Worte sowohl sehr unterschiedlich in der Länge als auch in der Häufigkeit sind, was ein starkes Indiz für die festgestellte Anomalie ist. Die unerwarteten Messergebnisse bei den Festplattenzugriffen der Suchoperationen auf dem Proteintext lassen sich also auf dessen inhomogene Struktur zurückführen.

## 5.7 Vergleich mit der Suffix Tree Datenstruktur

### 5.7.1 Vorbemerkungen

Nachdem in vielen einzelnen Experimenten untersucht wurde, wie sich der String B-Tree bezüglich unterschiedlichster Parameter der Eingabedaten verhält, soll nun ein kurzer Vergleich mit einer anderen Datenstruktur, dem Suffix Tree stattfinden.

Aus mehreren Gründen eignet sich der Suffix Tree besonders für einen solchen Vergleich: Er ist eine Datenstruktur, die genau in den zu untersuchenden Bereichen der Gendatenbanken und des Information Retrieval Anwendung findet. Konkrete Beispiele für die Anwendung finden sich in verschiedenen Ansätzen zur Mustersuche in Gendaten. Weiterhin ist der Suffix Tree in der hier verwendeten Form eine Hauptspeicherdatenstruktur, seine Zugriffe auf den Hintergrundspeicher werden also nur durch das Betriebssystem umgesetzt und nicht von der Implementierung selbst. Passt der Suffix Tree nicht mehr in den Hauptspeicher, so lagert das Betriebssystem einen Teil davon im Swapfile aus. Welche Teile es dabei auslagert und welche Teile im Hauptspeicher gehalten werden, hängt von den verwendeten Algorithmen und den Heuristiken des Betriebssystems ab.

Der Vergleich mit einer Hauptspeicherdatenstruktur ist insofern interessant, als dass bei der Auslagerung in den Hintergrundspeicher ein völlig anderer Weg gegangen wird. Während bei B-Baum Strukturen die Zugriffe auf den Hintergrundspeicher selbst verwaltet werden, wird diese Aufgabe bei anderen Datenstrukturen komplett dem Betriebssystem überlassen. Beide untersuchten Datenstrukturen (String B-Tree, Suffix Tree) sind also Repräsentanten zweier unterschiedlicher Ansätze für die Umsetzung der Auslagerung der Datenstrukturen in den Hintergrundspeicher. Dieser interessante Aspekt soll in dieser Untersuchung ebenfalls beleuchtet werden. Bei einem Vergleich mit einer anderen B-Baum Datenstruktur wäre dies nicht möglich gewesen. Ein solcher Vergleich ist außerdem bereits in der experimentellen Untersuchung von P. Ferragina und R. Grossi [5] mit dem Präfix B-Baum durchgeführt worden.

Für Suffix Trees gibt es eine Vielzahl von verschiedenen Repräsentationsformen ([13], [9],[11]). Für diese Untersuchung wurde eine Implementierung des *wotd* (write-only top-down) Konstruktionsalgorithmus [9] verwendet. Dies hat hauptsächlich zwei Gründe:

Zum einen den einfachen praktischen Grund, dass ein guter Kontakt zu der Gruppe, welche die *wotd* Implementierung erstellt hat [9], besteht. Somit konnten diverse Anpassungen der Implementierung an die Untersuchungumgebung sehr unkompliziert durchgeführt werden.

Zweitens ist *wotd* aufgrund seiner guten Speicherverwaltung eine sehr effiziente und aktuell verwendete Suffix Tree Implementierung. Um einen sinnvollen Vergleich mit einer aktuellen Datenstrukturen zu haben, würde ein Vergleich mit einer weniger effizienten Implementierung nur eine geringe Aussagekraft besitzen.

Abschließend ist zu dem Vergleich zwischen dem String B-Tree und dem Suffix Tree zu bemerken, dass es sich dabei nur um einen ersten Ansatz handelt, um eine Basis für weitere Untersuchungen zu schaffen. Für einen kompletten und ausführlichen Vergleich zwischen den beiden Datenstrukturen hätte man zusätzlich genauere Betrachtungen der *wotd* Implementierung und weitere Experimente durchführen müssen.

### 5.7.2 Suffix Trees (wotd)

Um die Ergebnisse dieser Untersuchung besser nachvollziehen zu können, soll in diesem Abschnitt der Aufbau eines Suffix Trees und die Vorgehensweise des *wotd* Algorithmus in komprimierter Form erklärt werden.

Ein Suffix Tree ist eine Baumstruktur, welche sämtliche Suffixe eines gegebenen Textes (Wortes) enthält. Die Speicherung der einzelnen Suffixe geschieht dabei auf die gleiche Art und Weise wie bei einem Compacted Trie (vergl. Abschnitt 2.3). Die einzelnen Teilstrings werden auf den Kanten des Baumes gespeichert und ergeben beim Traversieren von der Wurzel bis zu einem Blatt den jeweiligen Suffix.

Die für diese Untersuchung verwendete Implementierung basiert auf dem *write-only top-down* (*wotd*) Konstruktionsalgorithmus, der in anderen Arbeiten bereits mehrfach erwähnt wurde [1,8]. Er unterscheidet sich dabei von den bisherigen Ansätzen durch seine sehr lokalen Zugriffe innerhalb des Suffix Trees. Im Gegensatz zu anderen Algorithmen verwendet er keine sogenannten *Suffix Links*, die zusätzlich Referenzen innerhalb des Baumes sind und dazu dienen, die Zugriffe zu beschleunigen. Beim *wotd* Algorithmus wird die Effizienz des Aufbaus und der Suche durch das Lokalitätsverhalten erreicht. Die Experimente in [9] haben dabei gezeigt, dass sich die Lokalität im Hauptspeicher so stark auswirkt, dass der *wotd* Algorithmus der zeit- und platzeffizienteste Algorithmus für eine Vielzahl verschiedener Eingabedaten ist.

Der Grund für das gute Lokalitätsverhalten des Algorithmus ist, dass der Suffix Tree wie bei einer Breitensuche ebenenweise aufgebaut und verwaltet wird. Dieses soll zum Abschluss in der folgenden Beispielabbildung verdeutlicht werden:

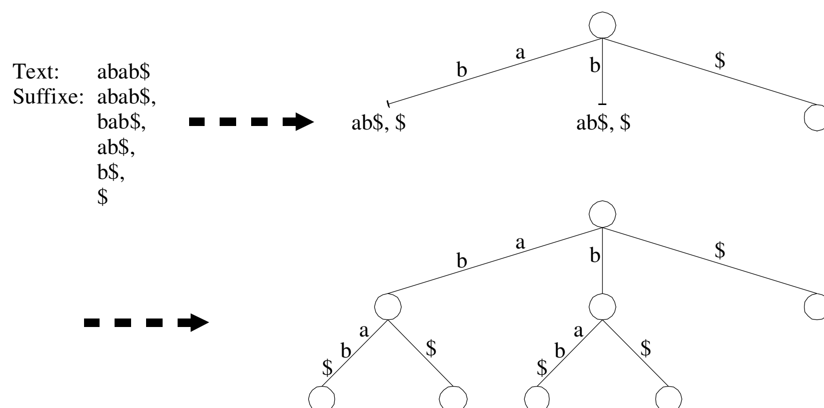


*Abbildung 10: Die write-only top-down (wotd) Konstruktion eines Suffix Trees*

### *5.7.3 Untersuchungsergebnisse und Analyse*

Um einen möglichst aussagekräftigen Vergleich zwischen den beiden Datenstrukturen zu realisieren, mussten einige der bisherigen Parameter angepasst werden. So wurde der Hauptspeicher des Testrechners von 128 auf 64 Megabyte reduziert, damit das Betriebssystem schon bei relativ kleinen Datenmengen auslagern muss. Wie schon mehrfach erwähnt, dienen diese absichtlichen Einschränkungen von Systemressourcen (vgl. Abschnitt 3.4.4) dazu, die gewünschten Effekte, welche sonst erst bei großen Datenmengen auftreten, bereits bei kleineren Szenarien zu erhalten.

Es wurde bei der Untersuchung für beide Implementierung die gleiche Suchoperation (Überprüfung, ob das Muster enthalten ist) betrachtet. Dabei wurde eine Musterlänge von 20 Zeichen verwendet. Im Gegensatz zu den bisherigen Untersuchungen wurden nicht 10.000, sondern nur 500 Suchoperationen auf dem String B-Tree durchgeführt. Einige Testläufe mit verschiedener Anzahl von Suchoperationen auf besonders großen Bäumen haben ergeben, dass bei der geringen Anzahl von 500 Suchoperationen die Cachingeffekte noch weniger ins Gewicht fallen und somit das logarithmische Laufzeitverhalten des String B-Trees besser erkennbar ist.

Da bei der Untersuchung der *wotd* Implementierung bei kleineren Werten größere Schwankungen auftraten und sich somit weniger repräsentative Ergebnisse ergaben, wurden für jeden Durchlauf 50.000 Suchoperationen verwendet. Alle angegeben Werte basieren auf diesen Ergebnissen und wurden im Verhältnis umgerechnet.

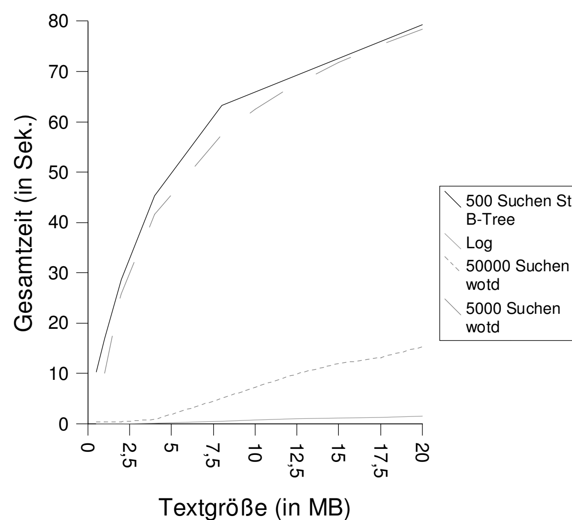


*Diagramm 16: Vergleich der Gesamtzeit der Suchoperation bei String B-Tree und Suffix Tree (wotd)*

In Diagramm 16 sind die Ergebniskurven der Suchoperationen auf String B-Trees und Suffix Trees bei Textgrößen zwischen 0,5 und 20 Megabyte dargestellt. Dabei ist zu

beachten, dass die Anzahl der Suchoperationen verschieden gross ist. Aufgrund der extrem großen Unterschiede der Gesamtzeiten hätte die gleiche Anzahl von Suchoperationen keine sinnvolle Darstellung ergeben.

Zunächst soll die Gesamtzeitkurve des String B-Trees betrachtet werden. Wie es sich bereits bei kleineren Datenmengen angedeutet hat, ist das Gesamtzeitverhalten (wie für eine B-Baum Struktur typisch) logarithmisch. Trotzdem ist der Verlauf der Kurve im Vergleich zu den Festplattenzugriffen wesentlich steiler. Hier bestätigt sich noch einmal, dass auch bei größeren Datenmengen eine Aussage über die Quantität der Festplattenzugriffe ohne deren Qualität in den praktischen Anwendungen wenig Sinn macht. Weiterhin zeigt sich, dass die für den Logarithmus typische Abflachung der Kurve erst ziemlich spät (zwischen 10 und 20 Megabyte) stattfindet. Zur Orientierung wurde die Kurve einer Logarithmusfunktion ( $10 \cdot ((\log_{(1,55)} x) + 1)$ ) in das Diagramm mit eingefügt.

Betrachtet man die beiden Kurven des *wotd* Algorithmus', so stellt man einen "Knick" bei ca. 4 Megabyte fest. Ab dieser Datenmenge reicht der Hauptspeicher anscheinend nicht mehr aus und das Betriebssystem muss Teile der Datenstruktur auslagern. Dieses Ergebnis deckt sich mit den theoretischen Vermutungen: Die Implementierung besitzt im Worst Case einen Platzbedarf von Faktor 12 bezogen auf die Größe des Ursprungstextes und im Average Case einen Faktor von 8,5. Der bei 64 Megabyte Hauptspeicher zur Verfügung stehende Speicher beträgt beim Testrechner 38 Megabyte, was mit den durchschnittlich benötigten $4 \cdot 8{,}5$ Megabyte in etwa übereinstimmt. In dem Bereich, in dem das Betriebssystem Daten auf den Hintergrundspeicher auslagern muss (ab ca. 4 Megabyte), zeigt sich dann ein linearer Kurvenverlauf.

Aufgrund des logarithmischen Verlaufes der String B-Tree Kurve und des linearen Verlaufes der *wotd* Kurve ergibt sich ein Schnittpunkt dieser beiden Kurven. Die Position des Schnittpunktes ist für die Auswertung dieser Untersuchung sehr wichtig, da sie angibt, ab welcher Textgröße die String B-Tree Implementierung effizienter als die *wotd* Implementierung ist. Aufgrund technischer Einschränkung war es bei der Untersuchung jedoch nur möglich Bäume von Texten mit höchstens 20 Megabyte Größe zu erstellen. In diesem Bereich ergibt sich jedoch kein Schnittpunkt. Auch eine Extrapolation wäre mit den vorhanden Angaben zu spekulativ und wird deshalb nicht durchgeführt.

Auch ohne eine genaue Angabe des Schnittpunktes ist jedoch klar erkennbar, dass die Laufzeit des Suffix Tree Algorithmus eine ganz andere Größenordnung besitzt, als die des String B-Trees. Dabei ist zusätzlich zu bedenken, dass bei dieser Untersuchung ein Rechner mit 64 Megabyte Hauptspeicher verwendet wurde. In der Praxis verwendete Rechner besitzen oft Hauptspeichergrößen welche im Gigabyte Bereich liegen. Bei diesen Größen würde sich das Ergebnis noch weiter zu Gunsten des Suffix Tree Algorithmus verschieben.

Dieser Vergleich hat also eindeutig gezeigt, dass die String B-Tree Implementierung wesentlich langsamer ist[19] als eine im Bereich der Genforschung verwendete

19 Diese und folgende Aussagen dieser Art sind alle auf die in der Praxis relevanten Textgrößen

Implementierung des Suffix Trees.

Im Folgenden wird betrachtet, wie dieses Ergebnis zu erklären ist. Dabei wird zuerst die Laufzeit des *wotd* Algorithmus' analysiert und danach abgeschätzt, inwieweit sich Optimierungen der String B-Tree Implementierung auf die Ergebnisse auswirken können.

Bei der Betrachtung der Laufzeit von *wotd* ist in erster Linie die Verwaltung von Haupt- und Hintergrundspeicher des Betriebssystems interessant. Wie bereits erwähnt, muss das Betriebssystem ab einer bestimmten Größe des Suffix Trees einen Teil davon in den Hintergrundspeicher auslagern. Bei den durchgeführten Experimenten betrug das im Extremfall 230 Megabyte an ausgelagerten Daten im Gegensatz zu 38 Megabyte Daten im Hauptspeicher. Auf den ersten Blick müssten sich dadurch wesentlich höhere Werte bei den Laufzeiten ergeben als in den Untersuchungen festgestellt wurde. Um diese geringen Laufzeiten zu erklären, muss die (betriebssystemgesteuerte) Speicherverwaltung der Datenstruktur betrachtet werden.

Ohne genaue Details des verwendeten Betriebssystems (Solaris) zu berücksichtigen, kann man trotzdem sagen, dass das Grundprinzip von virtuellen Speichern immer auf einer Art der LRU (Least Recently Used) Strategie basierend ist. Allein diese Information reicht aus, um sich klarzumachen, dass ein Großteil des "Stumpfes" (oberer Teil) des Suffix Trees im Hauptspeicher gehalten wird, da die Suchoperation diesen am häufigsten durchläuft. Betrachtet man nun die in den Experimenten verwendete Suchoperation, die nur das "enthalten sein" des Suchmusters überprüft, so stellt man fest, dass in einem Suffix Tree spätestens beim Erreichen der Suchmusterlänge die Suchoperation abgebrochen werden kann. Entweder musste bereits vorher abgebrochen werden (Suchmuster nicht enthalten) oder das Ende des Suchmusters ist beim Traversieren des Baumes erreicht worden (Suchmuster enthalten). In der schematischen Abbildung 11 wird nun deutlich klar, dass die große Mehrheit der untersuchten Muster[20] bereits im "Stumpf" des Suffix Trees endet und somit kein Zugriff auf den Hintergrundspeicher erfolgen muss. Die untersuchte Operation ist also für den Suffix Tree besonders gutartig und zeigt dadurch im Vergleich zum String B-Tree ein besonders gutes Laufzeitverhalten.

bezogen.

20 Die untersuchte Suchmusterlänge entsprach 20 Zeichen. Es wurden auch Versuche mit wesentlich höheren Werten durchgeführt, welche jedoch keine grundsätzlichen Abweichungen aufwiesen.

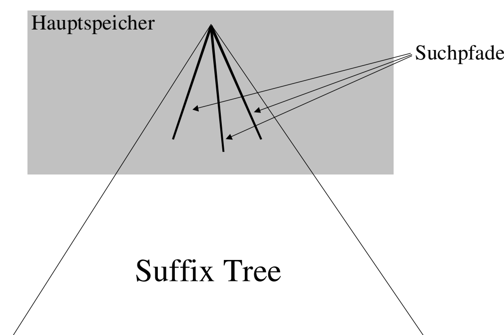


*Abbildung 11: Schematische Darstellung der Suchtiefe der untersuchten Muster im Suffix Tree*

Sollen jedoch alle Worte, welche diesen Präfix enthalten, ausgeben werden, so müsste der komplette Teilbaum des Präfixes traversiert werden. Somit würden sich aller Voraussicht nach wesentlich mehr Hintergrundspeicherzugriffe ergeben.

Aufgrund der fehlenden Implementierung dieser Operation kann diese nicht weiter untersucht werden, es folgt jedoch eine Abschätzung dieser Untersuchung, weil dies eine sinnvolle Ergänzung der bisherigen Ergebnisse darstellt.

Um den Zeitverbrauch für die Hintergrundspeicherzugriffe abzuschätzen, gehen wir von folgenden Werten aus: Wir nehmen an, dass jeder Festplattenzugriff nicht sequentiell, sondern zufällig geschieht und dafür 10 Millisekunden benötigt werden. Weiterhin nehmen wir an, dass zu jedem gesuchten Muster 4 passende Worte existieren und die Hälfte aller Zugriffe durch den Cache abgefangen werden, also sich im Hauptpeicher befinden und somit einen sehr geringen (und nicht berücksichtigten) Zeitverbauch haben. Zwecks Vereinfachung wird für die Ausgabe der Worte keine Zeit berechnet. Es ergibt sich für die 500 Suchoperationen im Suffix Tree somit ein zusätzlicher Zeitverbrauch von 10 Sekunden, um die Ergebnismenge der Suche zu bestimmen.

Für den String B-Tree ergibt sich nicht für jedes gefundene Wort ein Zugriff, da der Knoten, in dem (zumindest) das erste Ergebniswort enthalten ist, bereits geladen wurde. Bei dem verwendeten Knotengrad von 100 und einem angenommenen durchschnittlichem Füllungsgrad von 100 Elementen (199 sind möglich) würde sich nur ungefähr bei jeder 25. Suche ein weiterer Zugriff ergeben. Für den String B-Tree ergibt sich bei den 500 Suchoperationen somit ein zusätzlicher Zeitverbrauch von 0,1 Sekunden.

In Diagramm 17 wurden diese Werte zu den ursprünglichen Werten dazu addiert. Es ist zu beachten, dass in diesem Diagramm im Gegensatz zum letzten im Suffix Tree ebenfalls nur 500 Suchoperationen durchgeführt wurden.

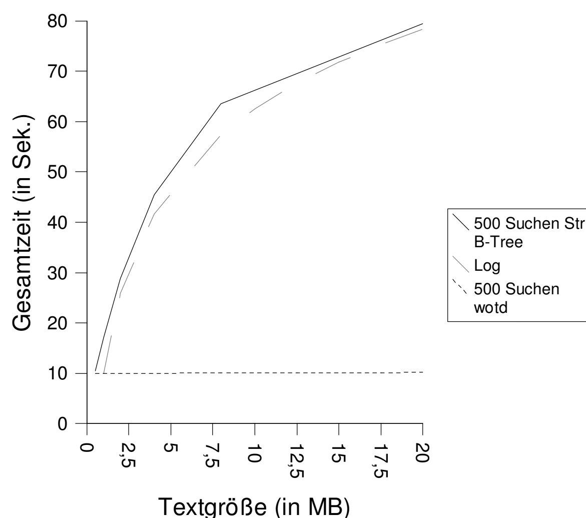


*Diagramm 17: Hypothetischer Gesamtzeitverbauch der erweiterten Suchoperation (mit Ausgabe der Fundelemente) in String B-Tree und Suffix Tree*

Wie bereits in der vorigen Untersuchung, sieht man immer noch einen deutlichen Unterschied zwischen den Laufzeiten der beiden Implementierungen. Die durchgeführte Abschätzung zeigt zumindest, dass das Ablaufen der gefunden Elemente bei Suffix Trees wesentlich mehr Zeit benötigt als der Suchalgorithmus selbst. Die Gesamtlaufzeit der erweiterten Suchoperation (mit Ausgabe) auf den String B-Trees dagegen bleibt fast unverändert, da der String B-Tree durch seine $B^+$-Baum Struktur für diese Operation optimal geeignet ist.

Trotz der abgeschätzten Betrachtung einer veränderten Suchoperation, die besser für den String B-Tree geeignet ist, zeigt sich die *wotd* Implementierung wesentlich schneller als die Implementierung der String B-Trees.

Nachdem die Ursachen für das Laufzeitverhalten der *wotd* Implementierung untersucht wurden, sollen nun die Auswirkungen möglicher Optimierungen der String B-Tree Implementierung abgeschätzt werden.

Grundsätzlich ist bei der Betrachtung der Implementierung zu beachten, dass diese mit dem Hauptaspekt der Erweiterbarkeit und im Hinblick auf die Experimente zur Untersuchung der Eigenschaften des String B-Trees bezüglich unterschiedlicher Einflussgrößen und nicht für den umfangreichen Vergleich mit anderen Datenstrukturen entwickelt worden ist. Es ergibt sich also ein großer Bereich für weiterführende Optimierungen, welche die Implementierung beschleunigen können.

Wie in den vorangegangen Untersuchungen klar erkennbar ist, wird der Hauptanteil der Laufzeit durch die Ausführung der Hintergrundspeicherzugriffe gebildet. Trotz der

verwendeten B-Baum Struktur, die von vornherein auf eine intensive Verwendung des Hintergrundspeichers ausgelegt ist, kann durch die Verwendung eines guten Puffers die Geschwindigkeit des Programmes erheblich gesteigert werden. Je nach Algorithmus und den verwendeten Heuristiken ist diese Steigerung sehr unterschiedlich. Somit kann deren Größenordnung ohne weitere Untersuchungen nur schwer eingeschätzt werden. Aus diesem Grund soll die Optimierung der Hintergrundspeicherzugriffe nach oben hin abgeschätzt, d.h. der Idealfall betrachtet werden. Das würde bedeuten, dass die Zugriffe auf den Hintergrundspeicher keine Zeit verbrauchen würden und somit als Gesamtlaufzeit nur die CPU-Zeit relevant wäre.

In Diagramm 18 ist diese Abschätzung dargestellt. In dieser Betrachtung konnte auch wieder die gleiche Anzahl an Suchoperationen (500) für beide Baumstrukturen verwendet werden. Wie man erkennt, zeigt sich auch für die Kurve der CPU-Zeit ein logarithmisches Verhalten. Trotz der idealisierten Abschätzung durch die CPU-Zeit ergibt sich insgesamt

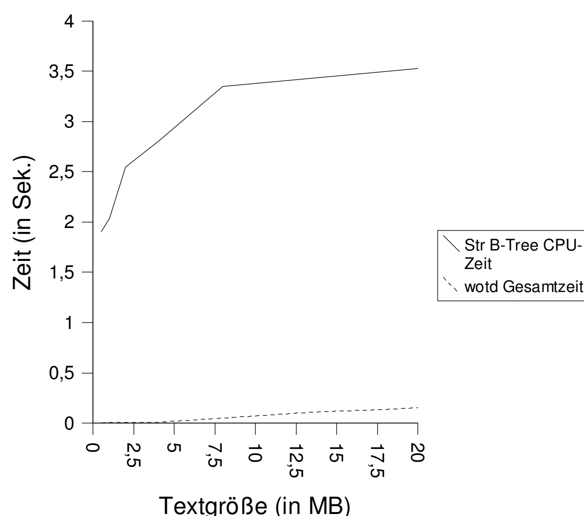


*Diagramm 18: Vergleich von Gesamt- und CPU-Zeit von 500 Suchoperation bei String B-Tree und Suffix Tree (wotd)*

ein ähnliches Bild wie davor.

Die String B-Tree Implementierung würde also (nach ersten Abschätzungen) selbst bei der Verwendung eines idealen Caches noch deutlich langsamer als der *wotd* Algorithmus sein. Es folgt eine abschließende Gesamtbetrachtung der Ergebnisse und der möglichen Ursachen.

Im Gegensatz zum String B-Tree ist das Laufzeitverhalten der *wotd* Implementierung zwar linear, jedoch ist die Steigung nicht so stark wie man es vielleicht auf den ersten Blick erwarten würde, sondern eher verhältnismäßig gering. Dies liegt in erster Linie wahrscheinlich an dem Speichermanagement des Algorithmus'. Wie bereits in dem

vorangegangenen Abschnitt erwähnt, ist der *wotd* Algorithmus schon im Hauptspeicher sehr auf ein lokales Zugriffsverhalten hin optimiert, da sich bereits dort eine weite Verteilung der Zugriffe negativ auf das Laufzeitverhalten auswirkt. Diese Eigenschaft hat ebenfalls einen positiven Einfluss auf das Auslagerungsverhalten des Betriebssystems. Andere Suffix Tree Implementierungen, die mit sogenannten Suffix Links arbeiten, würden dabei wahrscheinlich noch langsamer werden, da sie durch ihr fehlendes Lokalitätsverhalten bereits im Hauptspeicher weniger effizient sind als der *wotd* Algorithmus.

Die Hauptursache für die große Differenz der Laufzeiten von String B-Tree und Suffix Tree Implementierung liegt jedoch wahrscheinlich in deren unterschiedlichem Aufbau. Während beim Suffix Tree der komplette Aufbau sehr stark an die Struktur der enthaltenen Strings angepasst ist, so ist diese Struktur beim String B-Tree durch die B-Baum Hülle aufgebrochen. Durch diesen Einfluss ist der String B-Tree weniger abhängig von der Struktur der zugrunde liegenden Texte. Diese Eigenschaft verhindert zwar besonders schlechte Laufzeiten, das unabhängige Verhalten verhindert jedoch genauso eine besonders effiziente Ausführung und somit bessere Laufzeiten.

# 6. Zusammenfassung und Ausblick

Bevor eine Gesamtauswertung und die daraus resultierende Schlussfolgerung gegeben wird, soll an dieser Stelle noch einmal eine Zusammenfassung der wichtigsten Teile der Arbeit gegeben werden.

### *Der String B-Tree*

Der String B-Tree ist eine $B^+$-Baum Datenstruktur, die für die interne Verwaltung der Knoten Patricia Tries benutzt, um die Zugriffe auf den Hintergrundspeicher zu minimieren. Der String B-Tree benutzt dabei zwei Dateien, da er zusätzlich zur Datei, welche den Baum enthält auf die Textdatei zugreifen muss. Das liegt daran, dass der String B-Tree nicht die Elemente selbst, sondern nur deren Verweise beinhaltet.

Der Patricia Trie innerhalb der Knoten ist eine sehr kompakte Datenstruktur, wodurch in einem Knoten eine große Anzahl an Elementen gehalten werden kann, was in der $B^+$-Baum Struktur zu einem hohen Verzweigungsgrad und damit zu einer geringeren Baumtiefe führt. Weiterhin ist es durch die Verwendung von Patricia Tries möglich, dass für die Suche in den inneren Knoten nur ein Hintergrundspeicherzugriff benötigt wird. Somit werden für eine Suchoperation auf dem String B-Tree nur $2 \cdot (\log_g n) + 1$ Zugriffe auf den Hintergrundspeicher benötigt.

### *Implementierung*

Es wurde ein Prototyp "*sbtree*" in C implementiert. Zielsetzung der Implementierung ist die möglichst exakte Umsetzung des in [6] vorgestellten String B-Tree Modells und ein offener Programmaufbau, welcher leicht zu erweitern ist. Aus diesen Gründen wurden fast ausschließlich die in [6] erläuterteten Algorithmen verwendet und auf weiterführende Optimierungen verzichtet.
Intern codiert der Prototyp die Daten binär und verwendet für die Verwaltung der Elemente in den Knoten binäre Patricia Tries. Dies wurde bereits in [5] vorgeschlagen, da es effizienter zu programmieren ist und die Datenstruktur weitestgehend unabhängig vom Alphabet macht.

### *Planung der Untersuchungen*

Um die experimentellen Untersuchungen möglichst aussagekräftig und erschöpfend durchzuführen, ist eine detaillierte Planung der Untersuchungen notwendig. Dazu müssen sämtliche Einflussgrößen, Messgrößen und die Operationen auf String B-Trees auf ihre Relevanz im Hinblick auf die Zielsetzung der Untersuchungen betrachtet werden.
Die Einflussgrößen unterteilen sich in folgende drei Gebiete: Parameter der Datenstruktur (Baum-, Knoten- und Datenpuffergröße), Datenparameter (Eigenschaften der Texte) und die Umgebungsparameter (Betriebssystem, Hardware). Bei den Untersuchungen sollen zuerst die Parameter der Datenstruktur experimentell untersucht werden. Die sich daraus

ergebenden Erkenntnisse (z.B. optimale Knotengröße, geeignete Baumgröße) sollen dann verwendet werden, um die folgenden Experimente, welche die Datenparameter betrachten, so effektiv und aussagekräftig wie möglich zu gestalten. Die Umgebungsparameter werden als gegeben vorausgesetzt.
Die verwendeten Messgrößen sind die Festplattenzugriffe, die CPU- und die Gesamtzeit. Dabei ist die Gesamtzeit die wichtigste Messgröße, welche sich hauptsächlich aus der CPU-Zeit und der Zeit, die für die Festplattenzugriffe benötigt wird, zusammensetzt. Aufgrund vieler weiterer Faktoren, die sich auf die Gesamtzeit auswirken, besitzt diese jedoch eine relativ hohe Varianz. Die Festplattenzugriffe und die CPU-Zeit werden in den Untersuchungen beide als unterstützende Messgrößen verwendet.
Da die Suchoperation für die Praxis eine hohe Relevanz besitzt, wird bei den Experimenten hauptsächlich die exakte Substringsuche, welche prüft, ob der Substring enthaltenen ist oder nicht, untersucht.

### *Untersuchungsergebnisse*

In diesem Abschnitt werden noch einmal alle Untersuchungsergebnisse zusammengefasst, im nächsten Kapitel folgt die Gesamtauswertung der Ergebnisse.

- **Überprüfung der theoretischen Aussagen**
  In dieser ersten Untersuchung wurde anhand der gemessenen Festplattenzugriffe gezeigt, dass die Implementierung in ihrer Komplexität mit der in [6] angeführten Aussage von $2 \cdot (\log_g n) + 1$ Festplattenzugriffen pro Suchoperation übereinstimmt.
- **Technische Untersuchungen**
  In den technischen Untersuchungen wurden zuerst die Auswirkungen des Caches auf die Messergebnisse untersucht. Dabei stellte sich heraus, dass ab einer String B-Tree Größe von 640.000 Elementen die Auswirkungen des Caches nur noch sehr gering sind. In allen Folgeuntersuchungen wurden deshalb nur Baumgrößen ab 640.000 Elementen betrachtet.
  Weiterhin wurde experimentell der optimale Knotengrad von 100 ermittelt. Dieses entspricht einer Knotengröße von 6444 Byte, die etwas unterhalb der erwarteten Größe einer Betriebssystemseite von 8192 Byte liegt. Die Ursache dafür liegt in der Zuordnung der Betriebssystemseiten, die nicht exakt mit den Knoten übereinstimmt.
- **Ein- und Mehrworttext Untersuchungen**
  Die Untersuchung bezüglich der Ein- und Mehrworttexte hat ergeben, dass die Gesamtlaufzeit der Suche auf Einworttexten um 2,4% bis 4,7% geringer ist. Dies liegt an der Implementierungsspezifischen Funktion *word_length*, die für Ein- und Mehrworttexte unterschiedlich umgesetzt wurde. Die unterschiedliche Gesamtlaufzeit bei der Suche auf Ein- und Mehrworttexten ist also auf die Implementierung und nicht auf die Struktur des String B-Trees zurückzuführen.
- **Untersuchungen bezüglich der Alphabetgröße**
  Diese Untersuchung ergab für die Gesamtlaufzeit keine erkennbaren Abhängigkeiten

bezüglich der untersuchten Alphabetgrößen von 2 bis 127. Lediglich die CPU-Zeit unterscheidet sich minimal, da für die Binärcodierung der Alphabete eine feste Codierungslänge verwendet wird.

- **Untersuchungen bezüglich verschiedener Texte**
  Bei der Untersuchung des Laufzeitverhaltens anhand verschiedener realer Texte wie DNA-, Proteindaten und unterschiedlichen natürlichsprachlichen Texten zeigte sich nur eine Abhängigkeit von der hohen Repetitivität der Gendaten. Bei den restlichen Texten verhielten sich die Laufzeiten der Suche auf der String B-Tree Datenstruktur unabhängig von den Eigenschaften der Texte.
- **Vergleich mit der Suffix Tree Datenstruktur**
  Zum Abschluss wurde ein erster Vergleich mit einer anderen Datenstruktur, dem Suffix Tree (*wotd* Implementierung) durchgeführt. Der Suffix Tree ist eine im Bereich Gendatenbanken und Information Retrieval verwendete Datenstruktur.
  Trotz der Tatsache, dass der Suffix Tree eine Hauptspeicherdatenstruktur ist und die Auslagerung der Datenstruktur in den Hintergrundspeicher vom virtuellen Speicher des Betriebssystems umgesetzt wird, zeigte sich gegenüber dem String B-Tree eine signifikant bessere Laufzeit. Selbst weiterführende Abschätzungen bezüglich diverser Optimierungen des String B-Trees und die Betrachtung einer umfangreicheren Suchoperation änderten nichts an der (für den String B-Tree schlechten) Grundaussage der Ergebnisse.

## 6.1 Ausblick

### *Optimierung der Implementierung*

Wie bereits mehrfach erwähnt, bietet die vorhandene Implementierung viele Ansätze für weiterführende Optimierungen. Ein Teilbereich davon wäre die Umsetzung eines effektiven Puffers. Um eine erste Abschätzung für dessen Effizienz zu bekommen wäre als die Umsetzung eines *Memory Mappings* der beiden vom String B-Tree verwalteten Dateien denkbar. Das somit vom Betriebssystem durchgeführte Caching könnte dann bei Bedarf beliebig in der Implementierung verfeinert werden.
Ein anderer Teilbereich ist die Optimierung der Operationen auf den Patricia Tries. Ein vielversprechender Ansatz ist die effizientere Umsetzung der Dekomprimierung der Hintergrundspeicher-Repräsentation der Patricia Tries.

### *Experimentelle Untersuchungen in anderen Bereichen*

Aufgrund seiner Eigenschaften und seiner mächtigen Operationen (Suche beliebig langer Strings) bieten sich für den String B-Tree weitere Bereiche an, die einer Untersuchung bedürfen. Wie bereits in [5] erwähnt, zeigt sich der String B-Tree effizienter als der Präfix B-Baum. Es wären also besonders die Gebiete interessant, in denen bereits andere B-

Baum Varianten Verwendung finden. Außerdem wären auch experimentelle Untersuchungen anderer Speichermedien (CD-ROMs, Netzwerke) mit anderen Eigenschaften denkbar, in denen die effiziente Nutzung der Hintergrundspeicherzugriffe des String B-Trees besser zur Geltung kämen.

## 6.2 Abschlussdiskussion

Zum Abschluss dieser Arbeit sollen die Ergebnisse der Untersuchungen noch einmal in ihrer Gesamtheit diskutiert und bewertet werden.

Betrachtet man die Vielzahl der Ergebnisse der durchgeführten Untersuchungen, so stellt man dabei eine ähnlich große Übereinstimmung mit den B-Bäumen fest, wie sie bereits vorher in den theoretischen Analysen und den Modellen festzustellen war. Sowohl die Überprüfung der theoretischen Aussagen als auch die technischen Untersuchungen weisen große Gemeinsamkeiten mit der von mir im Vorfeld durchgeführten Untersuchung einer B-Baum Implementierung (vgl. Abschnitt 4.4) auf. Auch bei den weiteren Untersuchungen ist die Ähnlichkeit zu den B-Bäumen zu beobachten, wenn auch nicht mehr so deutlich zu erkennen. Die Ergebnisse der Untersuchungen, welche sich mit den unterschiedlichen Eingabedaten beschäftigen, zeigen bezüglich deren Eigenschaften ein im Grunde unabhängiges Verhalten der Datenstruktur. Ergeben sich Unterschiede in den Laufzeiten, so sind diese ausnahmslos mit den implementierungsspezifischen Eigenschaften und nicht mit der Struktur des String B-Trees selbst zu erklären. Auch das ist eine typische Eigenschaft des B-Baumes, da dieser durch seine ausgeglichene Struktur praktisch unabhängig von den enthaltenen Daten ist. Dieser Aspekt wird auch (zurecht) deutlich in [5] und [6] hervorgehoben, da eine ausgeglichene Struktur bei anderen Datenstrukturen (z.B. Suffix Trees) nicht immer gewährleistet werden kann. Der String B-Tree besitzt somit im Vergleich zu andern Datenstrukturen auch ein gutes Worst-Case Verhalten ([6,5]).

Die Abhängigkeiten des String B-Trees von der Suchmusterlänge und der Repetitivität eines Textes haben nichts mir der eigentlichen Baumstruktur zu tun, sondern sind (wie bei anderen Suchalgorithmen auch) auf den Vergleich der Strings zurückzuführen, da deren Gleichheit nur durch das komplette Ablaufen beider Strings festgestellt werden kann.

Die String B-Tree Datenstruktur zeigt sich also weitestgehend unabhängig von den in Gendatebanken und im Information Retrieval verwendeten Datenmengen.

Betrachtet man jedoch den Vergleich mit der in diesen Gebieten verwendeten Datenstruktur des Suffix Trees, so stellt man fest, dass die absoluten Laufzeiten des String B-Trees deutlich langsamer als die des Suffix Trees sind. Die Erklärung für diesen erheblichen Unterschied ist ebenfalls in der B-Baum Struktur des String B-Trees zu finden. Die in vielen Fällen vorteilhafte Unabhängigkeit von den Daten zeigt sich beim Vergleich mit dem auf Stringsuche spezialisierten Suffix Trees als Nachteil. Natürlich besitzt der Suffix Tree durch seine von den Daten abhängige Struktur ein sehr schlechtes Worst Case Verhalten. Wenn man jedoch die Fälle in der praktischen Anwendung, in

denen mehrheitlich der Average Case auftritt, betrachtet, so ist der Suffix Tree gerade durch seine Spezialisierung sehr effizient. Beim String B-Tree hingegen zeigte sich in den Untersuchungen, dass sich durch dessen Unabhängigkeit die Laufzeiten beim Average Case nur geringfügig von denen des Worst- bzw. Best-Case unterscheiden. Durch seine B-Baum Struktur werden beim String B-Tree also besonders schlechte Laufzeiten verhindert, gleichzeitig wird dadurch aber auch keine besonders effiziente Ausführung der Operationen möglich, wodurch die Substringsuche in den untersuchten Szenarien in den Suffix Trees wesentlich schneller als in den String B-Trees ist.

Es lässt sich also zusammenfassend festhalten, dass der String B-Tree durch seinen optimierten Aufbau und seine verschiedenen Operationen eine sehr effiziente und vielseitig verwendbare B-Baum Datenstruktur ist, die in ihrer Struktur weitestgehend unabhängig von den meisten Einflussgrößen ist. Bei der speziellen Betrachtung der Suchoperation in dem Bereich der Gendatenbanken und des Information Retrieval scheint der String B-Tree im Gegensatz zu anderen Datenstrukturen, wie z.B. dem Suffix Tree, jedoch weniger geeignet zu sein.

# 7. Anhang

## 7.1 Literatur

[1] A. Andersson, S. Nilsson. Efficient implementation of suffix trees. *Software: Practice and Experience*, 25(2):129-141, 1995.

[2] R. Bayer, K. Unterauer. Prefix B-Trees. *ACM Trans. Database Syst. 2*, 1 (Jan.), 11-26, 1977.

[3] Stuart M. Brown. *Bioinformatics: A Biologist's Guide to Biocomputing and the Internet.* Eaton Publishing, 2000.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] P. Ferragina, R. Grossi. An experimental study of SB-trees, *unpublished manuscript, http://citeseer.nj.nec.com/ferragina96experimental.html*, Uni. di Pisa/ Uni. di Firenze, 1996.

[6] P. Ferragina, R. Grossi. The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM*, Vol. 46, No. 2, pp236-280, 1999.

[7] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database System Implementation*, Dept. of Computer Science, Stanford University, 2000.

[8] R. Giegerich, S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. *Sci. Comput. Program.*, 25(2-3):187-218, 1995.

[9] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *In Proc. WAE'99*, LNCS 1668, pages 30--42, 1999.

[10] G. H. Gonnet, R. A. Baeza-Yates, T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, N. J., pp. 66-82, 1992.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[12] U. Manber, G. Myers. Suffix Arrays. A New Method for On-Line String

Searches. *SIAM J. Computers* 22, No 5, pp 935-948, 1993.

[13] E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262-272, 1976.

[14] K. R. Rose, Asynchronous Genereric Key/Value Database. *unpublished manuscript, http://www.krose.org/~krose/history.html,* 2000.

[15] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3), 1995.

[16] P. Weiner. Linear pattern matching algorithm. In *Proceeding of the 14th IEEE Annual Symposium and Automata Theory*, pages 1-11, The University of Iowa, 1973.