TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Fakultät für Mathematik und Informatik
Institut für Informatik

Exploring Data Structures in C: Trie

Fritz Meitner

Angewandte Informatik
Matrikel: 63720

18.10.2021

Inhaltsverzeichnis

1 Einleitung

Wenn Softwareprojekte komplexer werden und ein höherer Anspruch an verwendete Datenstrukturen gestellt wird, reichen die in Standardbibliotheken implementierten Datenstrukturen wie zum Beispiel: Arrays, Listen, etc. nicht mehr aus um die Daten angemessen zu verarbeiten. Eine Möglichkeit Strings effizienter zu speichern und es zu ermöglichen ohne großen Aufwand festzustellen, ob sich eine eingegebene Zeichenkette in einer vorher definierten Menge anderer Zeichenketten befindet, ist der Trie. Trie, was sich vom englischen Wort "retrieval"(Abfragen), ableitet, beschreibt eine Baumdatenstruktur, die vor allem zum Speichern von Zeichenketten geeignet ist und zum Beispiel bei der Auswertung von Eingaben bei Software zur Rechtschreibungsüberprüfung Anwendung findet. [1] Im Folgenden wird eine mögliche Implementation dieser Datenstruktur in C vorgestellt, die nicht an feste Alphabetgrößen gebunden ist und damit Flexibilität wie unter anderem Groß- und Kleinschreibung oder auch Sonderzeichen bietet. Die Zielsetzung der Arbeit besteht darin, eine Implementierung der Datenstruktur Trie in C zu designen und umzusetzen und zu bewerten. Indem unter anderem Aussagen zur Komplexität getroffen werden und anhand von Benchmarks, die Geschwindigkeit in Abhängigkeit von der behandelten Datenmenge, beurteilt wird.

2 Grundlagen

Tries, oder auch Prefix Trees genannt, sind eine Baumdatenstruktur, die keinen Binärbaum darstellt, da die Anzahl der Nodes, die von einer oberhalb im Baum liegenden, ausgeht abhängig von der betrachteten Datenmenge ist. Child-Nodes[2] werden immer genau dann hinzugefügt, wenn sie für einen neuen Eingabewert benötigt werden. Alle Strings, die eingetragen werden gehen dabei von einer einzelnen Parent-Node[3] in der obersten Ebene aus. Diese Root-Node[4] besitzt selbst keinen Key und stellt damit als Leerzeichen die Wurzel des Baumes dar, von der alle eingetragenen Strings ausgehen müssen.[5] Darüber hinaus besitzen nicht alle Nodes einen einzigartigen Wert. Alle Nodes, die nicht das Ende eines gültigen Keys markieren haben einen einheitlichen Wert, meist 0. Die Nodes, die das Ende von gültigen Keys markieren, haben einen von den nicht endständigen Nodes verschiedenen Wert, dieser muss auch nicht einheitlich sein. Je nach Zweck des Tries hat der Wert dieser Nodes nicht unbedingt Aussagekraft, da im Falle der simplen Überprüfung, ob eine gesuchte Zeichenkette innerhalb des Tries gefunden werden kann nur überprüft wird, ob der Wert gleich dem der Nodes ist, die nicht am Ende liegen, und wenn dies nicht der Fall ist, das Wort als gültiger Key erkannt wird.[6]
Tries implementieren grundsätzlich drei Funktionen. [7] Die Suchfunktion ist davon die simpelste. Bei dieser wird lediglich überprüft, ob der jeweils folgende Buchstabe des Strings eine Child-Node, der Node von der ausgehend die Überprüfung stattfindet, besitzt die zu diesem nächsten Key passt. Dies wird solang fortgeführt bis entweder der String als Teil des Tries gefunden wurde oder es zu einem Konflikt bei der Suche kommt. Konflikte können beim Suchen auf zwei Arten entstehen. Zum einen Bricht die Suche sofort ab, wenn eine der einzelnen Suchanfragen fehlschlägt, weil die aktuelle Node keine zum nächsten Zeichen des gesuchten

[1] Vgl. Trie - Data Structure, What is a Trie data structure?
[2] Node, die unterhalb einer bestimmten anderen Node innerhalb der Baumstruktur existiert
[3] Gegenstück zur Child-Node, weiter oben innerhalb der Baumstruktur
[4] Ausgangspunkt der Baumstruktur. Stellt den obersten Punkt innerhalb der Baumstruktur dar
[5] Vgl. Trie - Data Structure, Properties of the Trie for a set of the string
[6] Vgl. Trie | (Insert and Search)
[7] Vgl. Lau, K. Swift Algorithm Club: Swift Trie Data Structure

Keys passende Child-Node hat. Die andere Art von Konflikt, die beim Suchen entstehen kann, ist dass das gesuchte Wort zwar gefunden werden kann, aber es nicht als gültiges Wort markiert ist, in diesem Fall bricht die Suche nach der letzten Node ab und die Suchanfrage wird mit der Meldung, dass das Wort nicht gefunden wurde, beendet.
Die zweite grundlegende Funktion, die von Tries implementiert wird, ist das Einfügen. Bei dieser Funktion wird erst wie bei der Suchfunktion überprüft, ob sich die Zeichenkette, die hinzugefügt werden soll, bereits im Trie befindet, nur, dass die Fälle, in denen es beim Suchen zum Konflikt gekommen wäre, bei der Einfügefunktion, die Fälle sind, bei denen es zu keinen Konflikten kommt. So ist die einzige Möglichkeit, wie das Einfügen fehlschlägt, der Fall , dass das Wort bereits im Trie existiert. Im Falle, dass für eine Stelle in der einzufügenden Zeichenkette keine passende Child-Node gefunden wird, wird diese sowie alle nachfolgenden fehlenden Zeichen des Strings eingefügt, sowie im letzten Zeichen die Markierung, dass es sich um ein gültiges Wort handelt, gesetzt. Wenn das Wort bereits existiert, aber nicht gültig ist, muss nur im letzten Zeichen des einzufügenden Wortes markiert werden, dass es nun als Wort zu erkennen ist, und die Funktion wird erfolgreich beendet.
Die dritte und letzte grundlegende Funktion des Tries ist das Löschen von Wörtern. Bei dieser wird zunächst das zu löschende Wort gesucht, wenn dieses nicht gefunden werden kann bricht die Funktion ab. Sollte das Wort gefunden werden, wird die Markierung, dass es ein gültiges Wort war, entfernt. Nachfolgend hängt das Verhalten der Funktion von eventuell existierenden Child-Nodes ab. Wenn keine vorhanden sind, dann wird die Node entfernt und man bewegt sich entgegensetzt zur Richtung bei der Suche den Trie entlang. Dies wird solange fortgesetzt bis die nächste zu überprüfende Node eine Child-Node besitzt, selbst als gültiges Wort markiert ist oder die Root-Node ist, da man weder eine Node entfernt, die von einem von dem zu löschenden Wort verschiedenem String benötigt wird, noch ausgehend von einem Funktionsaufruf, der sich auf ein einzelnes Wort bezieht den gesamten Trie entfernt. In der Praxis werden Tries vor allem genutzt um eingegebene Zeichenketten auszuwerten. So finden sich Tries zum Beispiel bei Algorithmen wieder, die Eingaben auf Korrektheit überprüfen und Fehler markieren. Ein anderer Nutzen von Tries ist die Volltextsuche, bei welcher ein Text nach eingegebenen Worten und Wortgruppen ausgewertet wird und gefundene Übereinstimmungen markiert werden.[8]

3 Implementierung

Bei meiner Implementierung habe ich mich für eine Variante entschieden, die keine feste Alphabetsgröße nutzt, da ich zum einen sowohl Groß- als auch Kleinschreibung ermöglichen, aber außerdem eine alternative Implementation zur der Variante kreieren wollte, die man Online findet, wenn man nach Tries sucht, bei der in jeder Node ausgehend von der Alphabetsgröße für jedes gültige Zeichen des Alphabets in jeder Node einen Pointer auf eine potentiell existierende Child-Node im Speicher belegt.[9] Der Trie selbst wird initialisiert indem man eine einzelne Node kreiert, die die Wurzel für die Baumstruktur darstellt. Alle weiteren Daten des Tries werden durch Funktionen, relativ zur Wurzel, hinzugefügt. Des Weiteren sind die drei in den Grundlagen beschriebenen Funktionen umgesetzt. Im folgenden ist am Code erklärt, wie die einzelnen Nodes aufgebaut sind und wie die Funktionen umgesetzt werden.

[8] Vgl. Trie | (Insert and Search)
[9] Ein Beispiel für eine Implementation mit fester Alphabetsgröße lässt sich zum Beispiel auf `https://www.geeksforgeeks.org/trie-insert-and-search/` finden.

3.1 Nodes

```
struct node {
    char key;
    struct node** children;
    int child_size;
    int is_word;
    char* value;
    struct node* parent;
    int no_of_children;
};
```

Das Herzstück meiner Trie Implementation stellen die Nodes dar, die durch Verknüpfungen untereinander den Trie aufbauen. Der Wert von Key gibt an welches Zeichen, die Node speichert. Im Falle der Wurzel ist dieses ein Leerzeichen, im Falle anderer Nodes ist jedes ASCII[10]-Zeichen möglich, also können auch weitere Leerzeichen existieren, die nicht die Wurzel sind. Der Pointer-Pointer auf weitere Nodes children ist das Array variabler Größe, was die Child-Nodes verwaltet, die zu einem bestimmten Knoten gehören. Dieses Array hat keine fest definierte Größe um sowohl den Kritikpunkt an anderen Trie Implementationen, die nicht optimale Speichernutzung durch viele nicht verwendete, aber allozierte Pointer zu verbessern, aber auch keine Probleme anzutreffen, die entstehen, wenn doch viele Child-Nodes, eines Knotens existieren. So zum Beispiel muss gewährleistet sein, dass die Root-Node jeden möglichen Anfangsbuchstaben von Strings als Child-Node aufnehmen kann, da sonst Wörter mit bestimmten Anfangsbuchstaben im Trie nicht gespeichert werden könnten. An dieser eignet sich die Betrachtung des Mechanismus nachdem neu erstellte Nodes mit Daten gefüllt werden.

```
struct node * fill_node(char key, char* value, TRIE_BOOL word
                                , struct node* parent)
{
    struct node * empty = create_node();
    empty->key = key;
    if(word == TRIE_TRUE) {
        empty->is_word = word;
        empty->value = value;
    }
    empty->parent = parent;

    if((parent->no_of_children)+1 == parent->child_size) {
        parent->child_size= parent->child_size * 2;
        parent->children = reallocarray(parent->
                children,sizeof(struct node*),parent->child_size);
        memset(&(parent->children[parent->no_of_children])
                                ,0,(parent->child_size) / 2);
    }
    parent->children[parent->no_of_children] = empty;
    parent->no_of_children++;

    return empty;
}
}
```

Bei diesem wird zunächst eine leere Node erstellt, leere Nodes werden initialisiert mit dem Leerzeichen als Key. Der Startwert für die Anzahl der möglichen Child-Nodes liegt bei erstellten Nodes bei 4, was in tieferen Ebenen auch zu ungenutztem, aber alloziertem Speicher

[10] American Standard Code for Information Interchange, Zeichensatz aus 95 druckbaren und 33 nicht druckbaren Zeichen mit Kontrollfunktionen.

führen kann, allerdings zu weniger als bei alternativen Implementationen. Beim Füllen einer Node mit Daten werden als Inputs der zu speichernde Char, die Information, ob mit dieser Node, ein gültiges Wort endet[11], welche Node, diejenige ist, von der die mit Daten zu füllende, ausgeht und ein Char Pointer, in dem ein Wert gespeichert wird, sollte die Node ein gültiges Ende für ein Wort sein. Dieser Wert kann ausgegeben werden, wenn diese Node durch eine Suchoperation erfolgreich gefunden wurde. Dies wurde so implementiert, da es mit meiner Trie Implementation angedacht ist aus eingegebenen Abkürzungen die korrespondierende ausgeschriebene Form zu generieren, wobei dies bei der jetzigen Implementation auch umgekehrt möglich wäre. Der Pointer auf den String, der als Wert übergeben wird, wird allerdings nur dann in der Node gespeichert, wenn es sich bei der Node um das Ende eines Wortes handelt, sonst bleibt es ein Null-Pointer.
Die Parent-Node wird übernommen, anschließend wird überprüft wo im zugehörigen children Array, die neu erstellte Child-Node gespeichert werden kann. Zunächst wird dabei überprüft, ob ein freier Pointer im zugehörigen Array, der Parent-Node, gefunden werden kann. Wenn dies der Fall ist wird eine Referenz auf die Child-Node darin gespeichert und der Integer, der die Zahl der Knoten, die von der Parent Node ausgehen zählt wird inkrementiert. Sollte kein freier Platz im children Array gefunden werden, dann wird das Array neu alloziert mit größerem Speicher. Dafür wird die maximal mögliche Anzahl an Child-Nodes verdoppelt und nachfolgend für die doppelte Menge an Pointern Speicher reserviert. Abschließend wird der neu allozierte Speicherblock durch die memset[12]-Standartfunktion der C Bibliothek auf 0 initalisiert, bevor danach wie im ersten Fall, die Referenz auf die Child-Node im Array eingetragen werden kann.

3.2 Suchen

```
TRIE_BOOL search (char* string, struct node * root) {
    TRIE_BOOL found = TRIE_FALSE;
    struct node * found_child;

    for(int i = 0; i < root->no_of_children; i++)
    {
        if(string[0] == root->children[i]->key) {
            found = TRIE_TRUE;
            found_child = root->children[i];
            break;
        }
    }
```

Das Suchen ist in der hier beschriebenen Implementation eines Tries so umgesetzt, dass man ausgehend von einer Root-Node nach einer Zeichenkette sucht, dabei wird in jedem Schritt erst davon ausgegangen, dass keine Child-Node gefunden wurde. Der Suchvorgang ist so umgesetzt, dass durch das children Array, der Root-Node iteriert wird. Darin liegt ein Nachteil gegenüber der Implementation mit fester Alphabetsgröße, da in dieser, auf die Unterschiede bei der Komplexität beider Varianten wird im Punkt Zeitkomplexität genauer eingegangen. Unterschiede in der Performance werden unter Benchmarks betrachtet. Sollte eine passende Child-Node gefunden werden, wird die Schleife direkt abgebrochen um zur Auswertung

[11] bei TRIE_BOOL handelt es sich um einen Integer, der als true oder false betrachtet wird, da C standardmäßig keine Booleans implementiert. Für die Betrachtung innerhalb von Funktionen ist hierbei nur wichtig ob der Wert 0 ist, dann wird er als false betrachtet, wenn er nicht 0 ist immer als true.

[12] memset übernimmt einen Pointer, der angibt wo sich die zu setzenden Daten befinden, einen Integer, der angibt auf welchen Wert die Daten gesetzt werden sollen und einen unsigned Integer, der die Größe des zu setzenden Speicherblocks angibt.

überzugehen.

```
if(found == TRIE_TRUE) {
        if(string[1] == '\0') {
            if(found_child->is_word != TRIE_TRUE)
            {
                return TRIE_FALSE;
            }
            return TRIE_TRUE;
        }
        return search(string+1, found_child);
    }
    return TRIE_FALSE;
}
```

Die Auswertung wird in drei Stufen durchgeführt. Zunächst wird überprüft, ob eine Child-Node gefunden wurde. Wenn dies nicht der Fall ist wird die Suchoperation beendet, mit der Meldung, dass das gesuchte Wort nicht gefunden werden kann. Sollte ein passender Eintrag im children Array gefunden werden, wird im zweiten Schritt überprüft ob man sich am Ende des gesuchten Strings befindet. Sollte das Wort noch weitere Zeichen enthalten wird dann die Suchfunktion rekursiv aufgerufen, indem der Pointer, der auf das gesuchte Wort zeigt um einen die Größe eines Chars im Speicher erhöht wird und als neue Root das gefundene Child übergeben wird. Falls der String doch an der aktuellen Stelle ist und der betrachtete Char der Terminating Character[13] ist, wird im dritten Schritt überprüft ob man in der letzten Child-Node die Markierung findet, dass es sich um ein gültiges Wort handelt und nur wenn diese Markierung gesetzt ist, wird die Funktion mit dem Ergebnis beendet, dass als Wert zurückgegeben wird, dass das gesuchte Wort im Trie vorhanden ist.

3.3 Einfügen

```
while(*current != '\0') {
            TRIE_BOOL add = TRIE_TRUE;
            for(int j = 0; j < currnode->no_of_children; j++) {
                if (currnode->children[j]->key == *current) {
                    add = TRIE_FALSE;
                    currnode = currnode->children[j];
                }
            }
```

Das Einfügen von neuen Wörtern in den Trie ist implementiert, indem zunächst überprüft wird, ob die Nodes, aus denen sich der String zusammensetzt, bereits in der Datenstruktur vorhanden sind. Umgesetzt ist dies in dem man solange durch den Eingabestring iteriert, bis es zu einem Konflikt beim Suchen kommt, was zeigt, dass das Wort eingefügt werden muss. Dafür wird für jedes Zeichen ausgehend von der Annahme, dass es nicht existiert, überprüft ob die zugehörige Parent-Node, eine passende Child-Node besitzt, die das nächste zu findende Zeichen speichert. Sollte die Child-Node gefunden werden, wird der Pointer, der die Speicheradresse, der aktuell betrachteten Node speichert, angepasst. Dieser speichert dann eine Ebene tiefer in der Baumstruktur die Referenz auf die gefundene Child-Node. Ausgehend von dieser Node wird dann wieder durch die vorhandenen Child-Nodes iteriert, solang das nächste betrachtete Zeichen nicht der Terminating Character ist. Im Falle, dass in einer Node kein zum nächsten Character, der zur überprüfenden Eingabe, passende Child Node gefunden wird, wird im nächsten Schritt der Eingabe die neue Node hinzugefügt.

[13] In C sind Strings als Zeichenkette von ASCII Zeichen implementiert, dessen Ende durch den Character „\0“ markiert wird.

```
if(add == TRIE_TRUE) {
            struct node * new = fill_node(*current, "_", TRIE_FALSE,
                currnode);
            currnode = new;
            if(current[1] == '\0')
            {
                new->value = val;
                new->is_word = TRIE_TRUE;
            }
        }
        current+=1;
```

Dabei wird zu Beginn eine neue Node erstellt, die zur Erstellung nicht den String überreicht bekommt, von dem sie Teil ist. Dieser wird in einer gesonderten Überprüfung nur zu der Node hinzugefügt, die das letzte Zeichen des neuen, gültigen Wortes darstellt. Dies findet in der gleichen Schleife statt, wie die Überprüfung, ob das einzufügende Wort bereits existiert, so wird nach der Erstellung einer Node die Speicheradresse, auf welche der Pointer des aktuellen Zeichens des Strings verweist, um einen Character weiter gerückt und die Schleife wird weiter wiederholt. Der Performanceverlust dadurch, dass die Schleife, die überprüft, ob das nächste Zeichen gefunden werden kann, sollte minimal sein, da diese von Null bis zur Anzahl der Child-Nodes in Einerschritten iteriert, aber die Zahl der Child-Nodes bei gerade neu erstellten Nodes immer nur genau Null sein kann, findet kein Durchlauf durch diese erste Schleife statt, sobald einmal keine passende Child-Node gefunden werden konnte. Wenn der Pointer soweit verschoben wurde, dass das Zeichen, was sich an der n+1-sten Stellen befindet, der Terminating Character ist, wird die aktuelle neueste erstellte Node, als Ende eines gültigen Wortes markiert und ihr durch den Key abrufbarer Wert wird auf den Wert gesetzt, der am Anfang des Einfügebefehls als Eingabewert an die Funktion gegeben wurde.

3.4 Löschen

```
if(search(string, root) == TRIE_FALSE) {
        return;
    }
```

Um ein bereits hinzugefügtes Wort wieder aus dem Trie zu entfernen muss zunächst überprüft werden, ob dieses überhaupt gespeichert war. Dafür wird in der Funktion zum Löschen von vorhandenen Wörtern ein Funktionsaufruf der Suchfunktion durchgeführt. Die Löschfunktion bricht mit der Meldung, dass das zu löschende Wort nicht gefunden werden konnte ab, sollte die Suchfunktion nicht das erwartete Ergebnis liefern.

```
char* current = string;
    struct node * currnode = root;
    while(*current != '\0' && currnode->no_of_children != 0) {
            for(int j = 0; j < currnode->no_of_children; j++) {
                if(currnode->children[j]->key == *current) {
                    currnode = currnode->children[j];
                    current++;
                    break;
                }
            }
        }
    }
currnode->is_word = TRIE_FALSE;
```

Anschließend wird ein zweites Mal durch den Baum iteriert, um einen Pointer auf den letzten Buchstaben des Wortes, was gelöscht werden soll, zu generieren. Wenn dies geschehen ist

wird im ersten Schritt des eigentlichen Löschvorgangs in der letzten Node des Wortes die Markierung, dass es sich um ein gültiges Wort handelte, entfernt.

```
while(currnode->no_of_children == 0) {
        char tmp = currnode->key;
        free(currnode->children);
        currnode = currnode->parent;
        for(int i = 0; i < currnode->no_of_children; i++) {
            if(tmp == currnode->children[i]->key) {
                free(currnode->children[i]);
                memmove(&(currnode->children[i]),&(currnode->children[i+1])
                    ,(currnode->no_of_children-i) * sizeof(struct node *));
                currnode->no_of_children--;
                break;
            }
        }
    }
```

Betreffend des Löschens von einzelnen Nodes, wenn Wörter entfernt werden, wird eine Schleife genutzt, die so lang den Trie in Richtung der Root-Node durchläuft, bis die erste Node gefunden wird, die weitere Child-Nodes besitzt. Wenn die Node, mit der das Wort endete, keine Child Nodes besitzt, dann wird der Character, den die Node speicherte in eine temporäre Variable zwischengespeichert. Nachfolgend wird der Speicher, den das children Array, der zu löschenden Node belegt, freigegeben. Anschließend wird der Pointer auf die aktuell bearbeitete Node im Trie um eine Ebene nach oben versetzt und speichert nun die Adresse an der sich die Parent-Node befindet. Im nächsten Schritt wird durch das Children Array der darauf folgend betrachteten Node iteriert, bis die Child-Node gefunden ist, deren Key mit dem in der temporären Variable zwischengespeichertem übereinstimmt. Daraufhin wird genau dieses Child Node freigegeben und alle weitere Einträge in children um einen Speicherplatz nach vorne verschoben, damit es bei späteren Suchanfragen nicht zu Problemen kommt, wenn in der Mitte des Arrays ein Pointer auf Null gefunden wird, der natürlich keinen Key speichert, den die Suchanfrage allerdings versuchen würde abzufragen, was zum Absturz des Programms führt. Wenn die restlichen Child-Nodes verschoben sind und die Variable, die die Anzahl der Child Nodes speicher um eins reduziert wurde, dann wird aus der Schleife, die durch die children iterierte ausgebrochen und die Schleife, die überprüft, ob weitere Nodes zu löschen sind überprüft ob sie eine weitere Iteration durchführt. Dies wiederholt sich bis die Voraussetzungen zu weiteren Löschungen nicht mehr erfüllt sind.

4 Komplexität

Die Komplexität von Algorithmen beschreibt anhand von den Größenordnungen des Speicherplatzes und der Zeitabhängigen Anzahl von Operationen das Verhalten und die Güte von Algorithmen. Im Folgenden wird die Komplexität von Tries im Bezug auf den verwendeten Speicher und auf die Zeit, die Suchabfragen benötigen, betrachtet.[14]

4.1 Speicherkomplexität

Die Speicherkomplexität lässt sich nicht eindeutig genau für alle Tries festlegen, da sie sehr abhängig von den zugrunde liegenden Daten ist. Allerdings ist es möglich, eine Abschätzung für die im worst-case Szenario benötigte Menge an Nodes, zu treffen. Dies ist möglich, wenn man für die Anzahl an Wörtern annimmt, dass jedes Wort nur aus Nodes besteht, die ausschließlich

[14] Vgl. Lexikon der Mathematik - Komplexität von Algorithmen

für dieses Wort genutzt werden. Dann nimmt man alle Wörter als die durchschnittliche Wortmenge an. Dieses Produkt aus Menge der Wörter und durchschnittlicher Wortlänge, stellt die Menge der Charaktere dar, die man in den Trie einbringt, was die maximal mögliche Menge an Nodes für eine bestimmte Datenmenge darstellt. Deshalb ist die Speicherkomplexität eines Tries nach oben hin abschätzbar mit $\mathcal{O}(n * m)$ wobei n die Menge der Wörter repräsentiert und m die durchschnittliche Wortlänge.

4.2 Zeitkomplexität

Grundlegend ist die Zeitkomplexität von Tries bei Suchanfragen als $\mathcal{O}(n)$ anzusehen, wobei n die Anzahl der Ebenen des Tries darstellt. Dies beruht darauf, dass für jede Ebene, die durchquert wird einmal nach der nächstfolgenden Node gesucht werden muss. Was die Zeitkomplexität für die Suche nach der Nachfolgenode betrifft, gibt es Unterschiede darin je nachdem wie der Trie implementiert ist. Bei der klassischen Variante mit fester Alphabetsgröße sind die einzelnen Suchanfragen $\mathcal{O}(1)$ Zeitkomplex, da im Array an der mit dem Zeichen korrespondierenden Stelle überprüft wird, ob der Pointer auf eine gültige nächste Node verweist oder auf den Null-Pointer, in welchem Fall die Suche fehlschlägt. Während bei der im Rahmen dieser Arbeit betrachteten Implementation, eher von $\mathcal{O}(n)$ ausgegangen werden kann, da aufgrund der veränderlichen Länge des children arrays nicht vorhergesagt werden kann an welcher Stelle sich ein bestimmtes Zeichen befinden wird. Infolgedessen muss über alle Child Nodes iteriert werden, um zur gewünschten Node zu kommen. Meiner Meinung nach ist dieser Tausch es allerdings Wert, da man sehr viel Speicher einspart, der sonst mit überflüssigen Null-Pointern gefüllt wäre. Außerdem werden sehr wenige Nodes viele verschiedene Child Nodes haben, was auf modernen Computern eine sehr kleine Zeitdifferenz bedeutet, da man anstatt einer Leseaktion in den Speicher, nicht übermäßig mehr Speicher auslesen muss sondern in tieferen Ebenen maximal zwei bis drei Pointer ausgewertet werden müssen.

5 Varianten

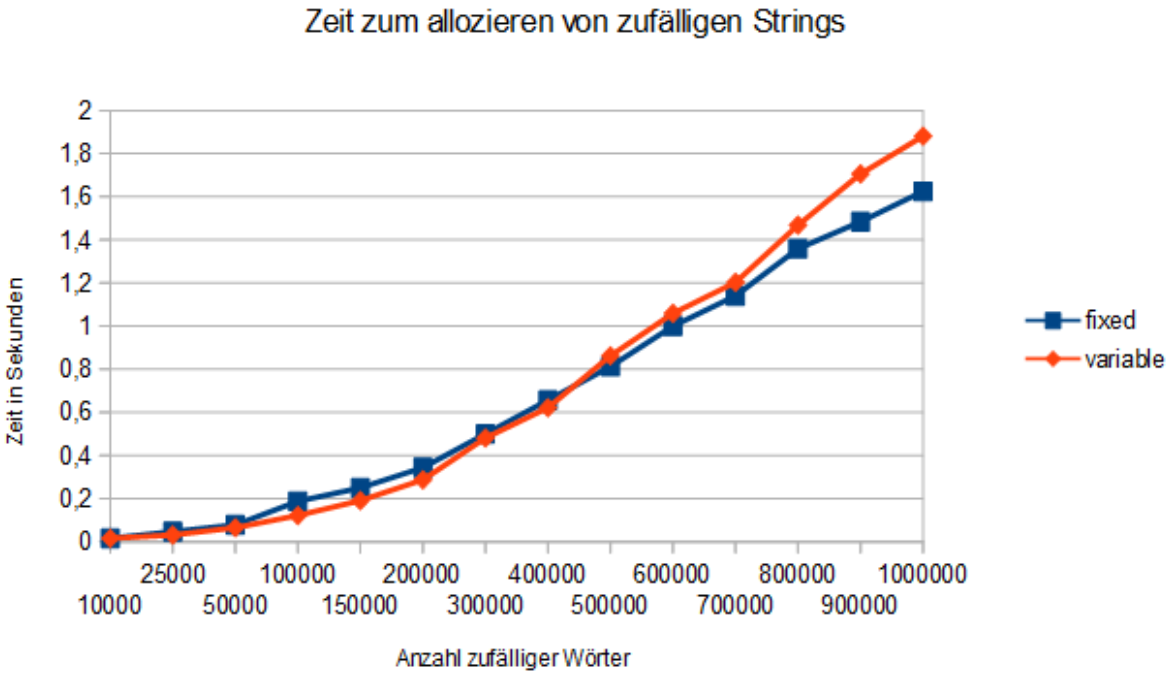
Wie bereits mehrfach erwähnt gibt es neben der Variante, für die ich mich bei meiner Implementation entschied noch eine zweite Möglichkeit. Um beide vergleichbar zu machen habe ich meine Implementation um eine Implementation der Node und der Funktionalitäten mit fester Alphabetgröße ergänzt. So fallen die Integer für die Größe des children array und die aktuelle Menge an children weg, da das array zum einen nun eine statische Größe ist und zum anderen die aktuelle Menge an Child Nodes keine Rolle mehr spielt, da nur anhand der Stellung im ASCII-Code überprüft wird, ob sich an der korrespondierenden Stelle eine Child Node befindet oder nicht. Abgesehen davon ist der Aufbau der Nodes gleich. Die Funktionen betreffend wurde jede iterative Schleife über children arrays, wenn es möglich war, durch direkte Aufrufe ersetzt. Was unter anderem zu einer komplizierteren Funktion zum letztendlichen Abbau des Tries führte, da in dieser weiterhin über alle Pointer iteriert werden muss, da man jedes child mit dieser Funktion erreichen muss, auf der anderen Seite sind die Funktionen zum Löschen und Hinzufügen von Nodes vereinfacht wurden, da nicht mehr realloziert werden muss, im Falle des Hinzufügens sowie die Daten nicht mehr verschoben werden müssen, wenn einige entfernt werden. Welchen Einfluss dies auf die Performance hatte wird im folgenden besprochen.

6 Benchmarks

Um die Leistungsfähigkeit der beiden Implementierungen zu bewerten, wurde zunächst eine Funktion erstellt, die zufällige Zeichenketten generiert um Daten bereitzustellen, die den Trie füllen.

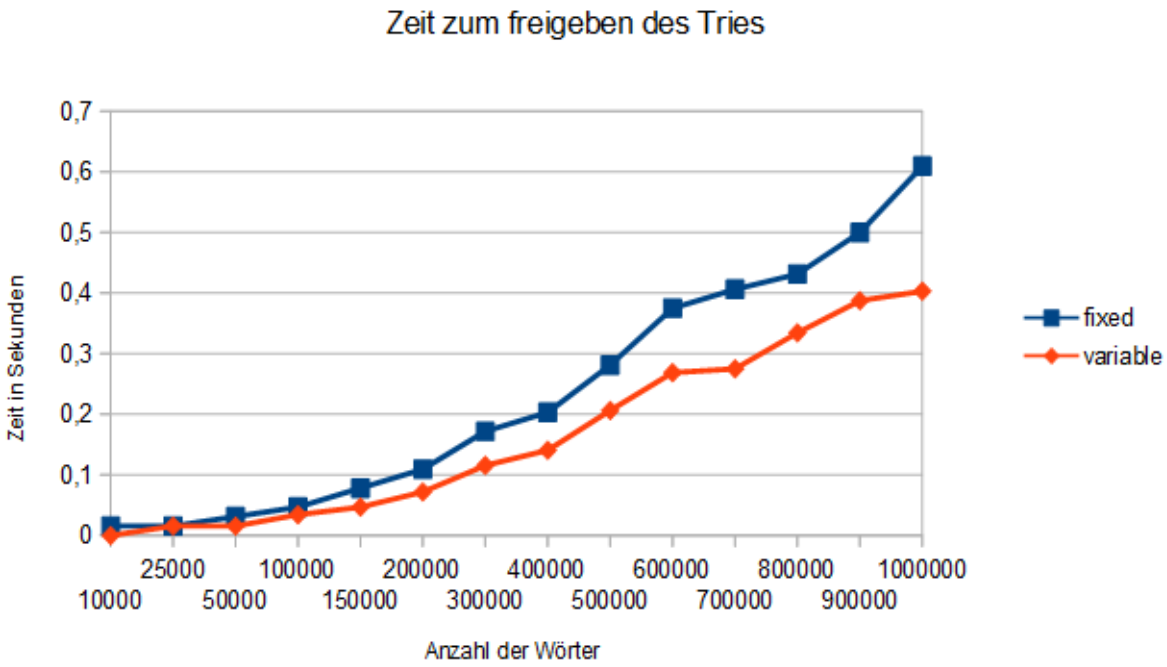
```
char* insert_random(char* string) {
    int length = (rand() % 14 + 2 );
    string = malloc(length);
    for(int i = 0; i < length; i++) {
        string[i] = (rand() % 26) + 'A';
    }
    string[length] = '\0';
    return string;
}
```

Die Zeiten, die zum Einfügen dieser Strings gemessen wurden können, als obere Abschätzung, für die Zeit betrachtet, werden, die eine Implementation eines realen Wörterbuchs benötigen würde, da die eingefügten Strings komplett zufällig sind und reale Sprachen bestimmte oft auftretende Präfixe haben, die nur einmal gespeichert werden müssen. Die Zeiten wurden gemessen, indem für verschiedene Mengen von zufälligen Strings je fünf Mal gemessen wurde, wie viel Zeit benötigt wurde. Die Strings werden nicht auf Einzigartigkeit überprüft, was dazu führt, dass eine zufällige Menge an Strings bei jedem Versuch nicht eingefügt werden kann. Außerdem wurde die Zeit gemessen, die es benötigt um den Speicher, der für den Trie genutzt wurde wieder freizugeben, falls es Anwendungen gibt bei denen ein Trie zwischendurch wieder entladen wird, wenn er für eine bestimmte Zeit nicht benötigt wird. Aus dem Diagramm wird



klar, dass für geringere Zahlen an zufälligen Strings, die Variante mit der variablen Child Node Arraygröße schneller ist, was daran liegt, dass weniger Speicher alloziert werden muss. Je größer die Menge der einzufügenden Wörter allerdings wird, desto mehr Child Nodes müssen

die höher in der Baumstruktur gelegenen Nodes speichern, was dazu führt, dass bei den Testmengen ab 600000 zufälligen Wörtern, die Variante mit statischer Arraygröße schneller den Trie aufbaut als die Variante die bei mehr Child Nodes das Array reallozieren muss. Allerdings tritt dies erst bei so großen String Mengen auf, dass auch abgesehen von der Speichererspar-nis sich variable Arrays besser für Tries einigen, da zum Beispiel die aktuelle Auflage des Dudens 148000 Stichwörter enthält und „der Wortschatz der deutschen Gegenwartssprache im Allgemeinen [auf] zwischen 300000 und 500000 Wörter“ [15] geschätzt wird. Im Bezug auf



die Zeit um den Trie wieder freizugeben, ist die Variante mit dem variablem Array immer schneller als die Variante mit dem statischen Array. Dies liegt vermutlich daran, dass beim Freigeben für die Arrays, fester Größe, durch alle 26 Einträge des Arrays iteriert werden muss, um zu überprüfen, ob sich in den jeweilligen Einträgen Child Nodes befinden, während bei der Variante, die ihre Arraygröße an die zugrunde liegenden Daten anpasst, abgebrochen werden kann, sobald einmal ein Eintrag auf den Nullpointer verweist, da die Child Nodes sich da nur in den ersten n Speicherzellen befinden, wobei n die Anzahl der children beschreibt.

7 Fazit

Abschließend lässt sich sagen, dass das Ziel der Arbeit erreicht wurde, indem eine Implementierung des Tries kreiert wurde, die eingegebene Strings speichern kann und zudem zu gültigen Wörtern einen zusätzlichen String als Wert aufnehmen kann, der bei Eingabe der zugehörigen Zeichenkette abgerufen werden kann, was die Verwendung als Abkürzungsverzeichnis ermöglicht. Eine Schwachstelle der Speichernutzung meiner Implementation ist, dass jede Node, unabhängig von ihrer Position in der Datenstruktur, immer nur ein Zeichen speichert. Die effizientere Variante, die ich leider nicht umsetzen konnte, ist der Radix-Tree, in dem einzigartige Enden von Eingaben in einer Node gespeichert werden, was dazu führt, dass

[15] https://www.duden.de/sprachwissen/sprachratgeber/Zum-Umfang-des-deutschen-Wortschatzes

bei weitem weniger Nodes insgesamt benötigt werden.
Über Tries im Allgemeinen ist zu sagen, dass sie sehr gut für ihren spezifischen Verwendungszweck, das Speichern von Strings und die Überprüfung, ob sich eine Eingabe in der Menge der Strings im Trie befindet, geeignet sind, weshalb sie zum Beispiel bei der automatischen Korrektur verwendet, werden um zu Überprüfen ob ein aktuell eingegebener String zu gültigen Wörtern führen kann. Außerdem ist eine Anwendung, die Vervollständigung von URLs im Browser, die genutzt wird um Seiten erneut zu öffnen, die bereits besucht wurden.
Wie man an den Benchmarks sehen kann, lässt sich dies bei nicht zu großen Mengen von Zeichenketten mit einer Variablen Anzahl von Nodes, die als Child Node gespeichert werden, schnell laden und speziell in der Praxis wird man diese Variante zu Anwendungen gebracht, da mit steigender Alphabetgröße davon auszugehen ist, dass die Menge an Strings, die gespeichert werden muss bevor die Variante mit statischen Arrays zeiteffizienter ist, immer größer wird, was ausgehend von aktuell gängigen Character Set Standards mit über einer Millionen verschieden Zeichen so viel sein sollte, dass Tries mit statischer Alphabetgröße in der Praxis keine Rolle spielen.

8 Quellenverzeichnis

Lau, K., Swift Algorithm Club: Swift Trie Data Structure [online] [Zugriff am 10. Oktober 2021] Verfügbar unter:
https://www.raywenderlich.com/892-swift-algorithm-club-swift-trie-data-structure

Lexikon der Mathematik - Komplexität von Algorithmen [online] [Zugriff am 10. Oktober 2021]
https://www.spektrum.de/lexikon/mathematik/komplexitaet-von-algorithmen/5385

Trie | (Insert and Search) [online] [Zugriff am 10. Oktober 2021] Verfügbar unter https://www.geeksforgeeks.org/trie-insert-and-search/

The Trie Data Structure (Prefix Trie) [online] [Zugriff am 10. Oktober 2021] Verfügbar unter:
https://www.spektrum.de/lexikon/mathematik/komplexitaet-von-algorithmen/5385

What is a Trie data structure? [online] [Zugriff am 10. Oktober 2021] Verfügbar unter:https://www.javatpoint.com/trie-data-structure

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Versicherung bezieht sich auch auf die bildlichen Darstellungen.

18.10.2021 Fritz Meitner