



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Projekt dyplomowy

Implementacja drzew klasy trie *Implementation of trie trees*

Autor:	<i>Jacek Krzysztof Oleś</i>
Kierunek studiów:	Informatyka
Opiekun pracy:	<i>dr inż. Radosław Klimek</i>

Kraków, rok 2020

Spis treści

1. Cele i założenia pracy	5
1.1. Wprowadzenie	5
1.2. Plan projektu.....	6
1.3. Spodziewane rezultaty	7
1.4. Metodologia.....	7
2. Wprowadzenie teoretyczne	9
2.1. Przegląd drzew <i>Trie</i>	9
2.1.1. Historia terminu <i>Trie</i>	9
2.1.2. Struktura i definicja	10
2.1.3. Zastosowania praktyczne	10
2.1.4. Przegląd operacji	11
2.1.4.1 Wstaw	11
2.1.4.2 Jest-pusty	12
2.1.4.3 Prefiks	12
2.1.4.4 Czy-należy	12
2.1.4.5 Szukaj	13
2.1.4.6 Usuń	13
2.1.4.7 Poprzednik i Następca	14
2.1.4.8 Minimum i Maksimum	14
2.2. Wariacje drzew <i>Trie</i>	15
2.2.1. Drzewo <i>Trie</i> – dwuwymiarowa tablica jako reprezentacja	15
2.2.1.1 Algorytm T – Przeszukiwanie <i>Trie</i>	18
2.2.1.2 Przeznaczenie	19
2.2.1.3 Złożoność czasowa i pamięciowa	19
2.2.1.4 Drzewo <i>Trie</i> reprezentowane przez skompresowaną tablicę dwuwymiarową	20
2.2.2. Las <i>Trie</i>	22
2.2.2.1 Złożoność czasowa i pamięciowa oraz przeznaczenie i możliwe modyfikacje w ich kontekście	23
2.2.3. Przypadek binarny – Przeszukiwanie cyfrowe drzewa oraz wsta- wianie	25
2.2.3.1 Algorytm D – Wstawianie	28
2.2.3.2 Przeznaczenie	29

2.2.4.	Drzewo <i>Patricia</i> – N-węzłowe drzewo przeszukiwań binarnej reprezentacji kluczy	29
2.2.4.1	Algorytm P – Sprawdzanie należenia argumentu do zbioru prefiksów	34
2.2.4.2	Rozszerzenie Algorytmu P – Wyszukiwanie zbioru węzłów akceptujących argument jako prefiks	36
2.2.4.3	Rozszerzenie Algorytmu P – Tworzenie drzewa <i>Patricia</i> , Wstawianie węzłów i Dodawanie tekstu	37
2.2.4.4	Przeznaczenie	40
2.2.4.5	Złożoność czasowa i pamięciowa	40
2.3.	Koniunkcyjna postać normalna	41
2.3.1.	Format plików <i>DIMACS CNF</i>	42
3.	Implementacja drzewa <i>Patricia</i>	43
3.1.	Wyzwania implementowania w języku programowania <i>Java</i>	44
3.2.	Implementacja kodowania znaków oraz reprezentacji bitowej maszyny <i>MIX</i>	47
3.3.	Implementacja drzewa <i>Patricia</i> – w zgodności z wymaganiami Knuth’a	48
3.4.	Implementacja drzewa <i>Patricia</i> – autorski algorytm	49
3.5.	Koniunkcyjna postać normalna w drzewie <i>Patricia</i> – wykorzystanie implementacji w rozwiązaniu problemu praktycznego	51
3.6.	Reprezentacja graficzna stworzonego drzewa <i>Patricia</i>	54
3.7.	Testy jednostkowe	57
3.8.	Instrukcja uruchomienie programu inżynierskiego	58
4.	Podsumowanie	63
4.1.	Osiągnięcia	63
4.2.	Rezultaty implementacji oraz analiza pracy	64
4.2.1.	Implementacja drzewa <i>Patricia</i> według Knuth’a	64
4.2.2.	Autorska modyfikacja algorytmu drzewa <i>Patricia</i>	66
4.2.3.	Koniunkcyjna postać normalna – wykorzystanie autorskiej implementacji w rozwiązaniu problemu praktycznego	67
4.2.4.	Reprezentacja graficzna stworzonego drzewa	69
4.3.	Możliwości rozwoju	69
	Bibliografia	71
	Załączniki	73

1. Cele i założenia pracy

1.1. Wprowadzenie

Jedną z najbardziej znanych struktur danych są drzewa. Pełnią one kluczową funkcję w informatyce, od drzew decyzyjnych w algorytmach sztucznej inteligencji, po najprostsze drzewa binarne i pojawiają się w prawie każdej tematyce informatyki. Jeżeli w danej tematyce nie występuje wykorzystanie struktury drzewa, to pojawia się rozwiązanie inspirowane hasłem „drzewo”. Wydaje się, iż mimo tego, że do stereotypu informatyka nie pasuje określenie miłośnika natury, to obraz drzewa jest głęboko zakorzeniony w naszych myślach.

Celem niniejszej pracy jest analiza algorytmów tworzących i przeszukujących drzewa *Trie* oraz implementacja tych dotyczących wybranego rodzaju drzewa w języku *Java*. Część teoretyczna pracy oparta jest o wiele źródeł, ale w większości budowana jest na informacjach, które można znaleźć w książce Donalda Knuth’a „Sztuka programowania” [1]. Wykorzystanie tej książki wynika nie tylko ze względu na uznanie, jakim jej autor cieszy się w środowisku informatycznym, ale także z uwagi na to, iż ściśle naukowa literatura dokładniej opisująca strukturę drzew *Trie* jest dość uboga.

Motywacją dla niniejszej pracy jest rozjaśnienie i uściślenie pojęć związanych ze strukturą nazywaną drzewem *Trie* oraz jednym z jego wariantów – drzewa *Patricia*. W literaturze oraz innych źródłach – jak na przykład artykułach znajdujących się na mniej lub bardziej technicznych stronach i serwisach internetowych – pojęcie *Trie* pojawia się często. Mimo iż żadne z nich technicznie rzecz biorąc, nie wprowadza czytelnika w błąd, to rzadko przedstawia pełny obraz wymagań postawionych przed tą strukturą. Przykładami takich internetowych źródeł są [2, 3, 4, 5, 6].

Hipotezą, którą chcemy udowodnić w niniejszej pracy, jest stwierdzenie, mówiące o tym, że można zmodyfikować algorytmy Knuth’a dotyczące drzewa *Patricia* tak, aby klucze mogły przyjmować postać pojedynczego słowa nie wpływając w sposób znaczący na przetwarzanie danych wewnątrz algorytmów oraz na samą strukturę drzewa. Mamy tu na myśli założenie, które mówi o tym, że klucz może rozpoczynać się w pozycji startowej przechowywanej w węźle drzewa – tak jak opisuje to Knuth – ale nie musi się kończyć znakiem końca pliku.

Wstępnie obranie takiej hipotezy wynikało z wyróżniającego się sposobu określania przez Knuth'a czym jest klucz. Klucze we wszystkich pozostałych algorytmach w tym rozdziale przyjmowały postać pojedynczego, rozłącznego słowa. W związku z tym chcieliśmy ujednolicić pojęcie klucza, tak aby czytelnik nie czuł się zagubiony, gdyż Knuth – naszym zdaniem – pozostawia w większości powód takiej reprezentacji klucza w domyśle. Myśl – „Dlaczego definicja tego, czym jest klucz w tym algorytmie, tak bardzo różni się od innych opisanych w tym rozdziale?” – nie opuszczała naszych głów przez cały proces próby zrozumienia algorytmu. Podejrzewaliśmy nawet, że jest to integralną częścią algorytmów opisanych przez Knuth'a i zmiana formy klucza, wiązałaby się z całkowitą modyfikacją przynajmniej części algorytmów.

1.2. Plan projektu

Celami teoretyczno-edukacyjnymi jakie przyjęliśmy w tej pracy są:

1. Przedstawienie w przystępny sposób terminów drzewa *Trie* i jego wariacji – w tym drzewa *Patricia*,
2. Omówienie operacji możliwych do przeprowadzenia na takich drzewach,
3. Zaprezentowanie algorytmów definiujących poszczególne wariacje drzewa *Trie* oraz,
4. Rozważenie zalet i wad każdego z nich.

W tym projekcie inżynierskim podejmujemy nie tylko próbę implementacji algorytmu opisanego przez Donalda Knuth'a, ale dodatkowo rozszerzenia wybranego algorytmu, zmieniając tym samym zastosowanie z bardzo specyficznego na bardziej uniwersalne.

Co więcej, aby zaprezentować trafność naszej implementacji i jej modyfikacji, implementujemy klasy symulujące reprezentację binarną znaków maszyny *MIX*, wykorzystywaną przez Knuth'a w opisach algorytmów. Nasza symulowana maszyna *MIX* ma możliwość zmiany długości pojedynczego bajta oraz gwarantuje istnienie dwóch specjalnych, sparametryzowanych znaków w jej tablicy kodowania znaków.

Kolejnym autorskim wkładem jest próba zastosowania implementacji do rozwiązania problemu praktycznego, zadanego przez promotora tego projektu inżynierskiego – dr inż. Radosława Klimka. W niniejszej pracy podejmujemy się rozwiązania problemu koniunkcyjnej postaci normalnej, zadanej w postaci plików formatu *DIMACS CNF*.

W celu ułatwienia zrozumienia algorytmu, prostszej analizy danych oraz dla walorów edukacyjnych stworzona została również reprezentacja graficzna struktury powstałej w efekcie implementacji drzewa *Patricia*.

1.3. Spodziewane rezultaty

W efekcie wykonania opisanych w poprzednim rozdziale działań spodziewamy się otrzymać rezultaty pozwalające nam na poniższe wnioski.

Istnieje podział na różne rodzaje drzew *Trie*, gdzie każdy z nich jest pewną modyfikacją rodzaju, na którym bazował. Mimo to istnieje jedna, prawdziwa struktura *Trie*, a jego inne wariacje muszą przyjąć jasno rozróżniające je nazwy.

Jest możliwe zaimplementowanie algorytmów przedstawionych przez Knuth'a w jednym z popularniejszych dzisiejszych języków, jakim jest *Java*, mimo wielu założeń uproszczających przyjętych przez niego – bardzo często związanych z reprezentacją binarną znaków fikcyjnej, niskopoziomowej maszyny *MIX*.

Można zmodyfikować algorytmy Knuth'a dotyczące drzewa *Patricia* tak, aby klucze mogły przyjmować postać pojedynczego słowa, nie wpływając w sposób znaczący na przetwarzanie danych wewnątrz algorytmów oraz na samą strukturę drzewa.

Implementacja wybranego rodzaju drzewa jest w stanie odpowiadać przynajmniej na część pytań dotyczącej charakterystyki zawartej w niej koniunkcyjnej postaci normalnej, przyjmując pewne założenia uproszczające.

Wizualizacja wybranego rodzaju drzewa – mimo iż nie może zobrazować dużych, złożonych problemów zawartych w strukturze drzewa – przedstawia sobą walory edukacyjne oraz może pomóc w obrazowaniu uproszczonych problemów i w ten sposób wspierać proces rozumowania, oraz wnioskowania użytkownika.

1.4. Metodologia

Rozdział pierwszy jest rozdziałem wprowadzającym i przedstawiającym aspiracje wobec projektu inżynierskiego.

Rozdział drugi poświęcony jest przedstawieniem wiedzy teoretycznej mającej na celu wyrównanie poziomu wiedzy w zakresie tematu projektu między autorem a czytelnikiem.

Rozdział trzeci jest przeznaczony omówieniu w sposób szczegółowy zastosowanych podczas implementacji rozwiązań, których wyjaśnienie nie pojawiło się w poprzednich rozdziałach.

Celem rozdziału czwartego jest podsumowanie pracy i osiągnięć w odniesieniu do spodziewanych rezultatów oraz hipotezy, które zawarte są w rozdziale pierwszym.

2. Wprowadzenie teoretyczne

2.1. Przegląd drzew *Trie*

Oprócz artykułów na różnych stronach internetowych, w literaturze naukowej można znaleźć wiele terminów czy opisów zawierających hasło *Trie*. Przykładem takiego źródła jest znana wszystkim książka Cormen'a (i innych) "Wprowadzenie do algorytmów" [7], posiadająca pod-rozdział o tytule "y-fast tries" nawiązujący do drzew van Emde Boas'a. Zdecydowanie trudniej znaleźć publikację, które rozlegle i dogłębnie omawiają termin „zwykłego“ drzewa *Trie* i *Patricia*. Na szczęście istnieje dzieło Donalda Knuth'a o nazwie "Sztuka programowania", a w szczególności tom 3 [1], w którym można znaleźć cały rozdział poświęcony terminowi *Trie* i *Patricia*.

2.1.1. Historia terminu *Trie*

Po raz pierwszy drzewo *Trie* zostało przedstawione przez Axel Thue jako abstrakcyjne pojęcie użyte do reprezentacji zbioru ciągów znaków. W pracy o "Ciągach znaków niezawierających przyległych powtarzających się podciągów znaków" [8] z 1912 roku Axel zaprezentował implementację przy użyciu tablicy o rozmiarze rozważanego alfabetu.

Pamięć typu *Trie* w kontekście przeszukiwania komputerowego została szerzej opisana przez René de la Briandais w artykule "Przeszukiwanie pliku używając kluczy o zmiennej długości" z roku 1959 [9, 6]. René zauważył, że korzystając z listy zamiast tablic, oszczędzimy pamięć kosztem czasu wykonywania.

Termin *Trie* został użyty dopiero w 1960 roku przez Edwarda Fredkina w artykule "Trie Memory" [10]. Nazewnictwo to zostało zaczerpnięte z wyrażenia angielskiego *information retrieval*, które oznacza wyszukiwanie informacji. Wymowę terminu *Trie* zmodyfikowano na odpowiadającą słowu *try* na potrzeby odróżnienia go od wymowy *tree*.

2.1.2. Struktura i definicja

Donald E. Knuth w swoim trzecim tomie książki pod tytułem “Sztuka Programowania” [1] wymienia wspomniane oraz inne abstrakcyjne konstrukcje kryjące się pod terminem *Trie*.

Zgodnie z jego definicją *Trie* to drzewo posiadające węzły o co najwyżej M dzieciach. To tak zwane drzewo M -ary [11] lub M -way tłumaczone jako drzewo stopnia M . Oczywiście zamiast litery M w nazwie takiego drzewa mogą zostać użyte inne litery, na przykład N lub K .

Oprócz tego, węzły drzewa *Trie* mają postać M -elementowych wektorów. Komórki tych wektorów są przyporządkowane znakom, na przykład pojedynczym literom bądź cyfrom.

Każdy znak pozwala na przejście gałęzią z jednego węzła znajdującego się na poziomie L do następnego węzła znajdującego się na poziomie $L + 1$, gdzie korzeń drzewa odpowiada pustej sekwencji znaków i jest na poziomie 0. Węzeł na poziomie L reprezentuje ostatni element sekwencji znaków o długości L , która jest nazywana prefiksem omawianego węzła. Prefiks niekoniecznie musi być całym słowem znajdującym się w drzewie *Trie*, a jedynie jego fragmentem. Węzeł na każdym poziomie posiada M rozgałęzień - wychodzących krawędzi - prowadzących na poziom $L + 1$. Elementem decydującym, którą gałąź należy wybrać, jest znak w sekwencji o numerze $L + 1$.

Należy zwrócić uwagę na to, że Knuth definiuje poziom węzła [12] w sposób rekursywny, gdzie korzeń drzewa znajduje się na poziomie $L = 0$, a poziom innych węzłów jest równy odległości liczonej w krawędziach od korzenia do danego, omawianego węzła. Taki mały szczegół może przyprawić czytelnika o poważny ból głowy.

Do każdego z tych dzieci można się dostać poprzez jeden z zestawu kluczy reprezentowanych przez węzeł rodzica. Inaczej mówiąc, węzeł rodzic jest rozgałęzieniem drzewa na M węzłów dzieci – stąd nazwa M -way, bo posiada M dróg wychodzących. Efektem tego założenia jest zależność położenia każdego z węzłów dzieci od sekwencji znaków o długości $L + 1$. Prawdziwe jest też stwierdzenie, że położenie węzła dziecka względem węzła rodzica jest określone przez element ciągu znaków o numerze $L + 1$.

Szczególnie popularną wariacją drzewa *Trie* ze względu na mylącą prostotę rozwinięcia jego koncepcji jest drzewo *Patricia*. Na pytanie na czym polega ta koncepcja oraz o tym czy jest ona rzeczywiście prosta do zastosowania w praktyce odpowiadam w sekcji dotyczącej drzewa *Patricia* 2.2.4.

2.1.3. Zastosowania praktyczne

Hasło *Trie* przewija się często w pracach naukowych oraz rozwiązaniach komercyjnych. Przykładami takich zastosowań są:

1. Zastosowania naukowe:

- (a) Operacje na zbiorach oparte o *Trie* i selektywna komputacja prostych implikantów (ang. „*Tri-Based Set Operations and Selective Computation of Prime Implicates*“) [13],

Wykorzystanie implikantów prostych do uproszczenia formuł logicznych w formach *CNF* i *DNF*;

- (b) Rozwiązania dużych problemów wyłuskiwania informacji (ang. „*large information-retrieval problems*“) [14];
- (c) Trie wykorzystujące pliki dla urządzeń wbudowanych (ang. „*File Based Trie for Embedded Devices*“) [15];
- (d) Samo-stabilizujące się *hashowe* drzewo *Patricia Trie* (ang. *A Self-Stabilizing Hashed Patricia Trie*) [16].

2. Zastosowania komercyjne:

- (a) Programy w języku Fortran dla komputerów CDC-3600 i UNIVAC 1108 [14];
- (b) Funkcja *Speed Dial* w przeglądarce *Opera*;
- (c) Słownik T9 na wielu urządzeniach mobilnych.

3. Możliwe inne zastosowania:

- (a) Tablica routingu IPv4;
- (b) Sprawdzanie poprawności pisowni;
- (c) System podpowiedzi pełnego hasła na podstawie już wpisanego tekstu (ang. *autocomplete*).

2.1.4. Przegląd operacji

Keith Schwarz z uniwersytetu Stanford w swoich wykładach [17] przeznaczonych dla kursu dotyczącego struktur danych porusza temat operacji wykonywanych na drzewach, które pełnią funkcję słowników.

Każdy rodzaj drzewa *Trie* lub *Patricia* można postrzegać jako słownik z określoną kolejnością (ang. *ordered dictionary*), który zawiera zbiór S składający się z elementów na płaszczyźnie U , która również ma swoją, określoną kolejność (ang. *ordered universe*).

Aby drzewo mogło zostać użyte do przedstawienia takiego słownika, powinno wspierać operacje przedstawione w tym podrozdziale.

2.1.4.1. Wstaw

Operacja wstaw (ang. *insert*) dodaje element x , który jest argumentem metody do zbioru S . W kontekście drzew *Trie* oraz *Patricia* należy interpretować tę metodę jako dodanie do nich nowego klucza.

2.1.4.2. Jest-pusty

Operacja jest-pusty (ang. *is-empty*) odpowiada na pytanie, które może być interpretowane jako:

- Czy zbiór S jest pusty ($S = \emptyset$)?
- Czy drzewo jest puste?
- Czy struktura posiada dowiązanie (np. wskaźnik) na zainicjowany węzeł, reprezentującym jakiś klucz? W kontekście niektórych algorytmów opisanych w tym rozdziale dotyczących drzew *Trie* węzeł ten musi być oddzielnym węzłem od korzenia drzewa, a w kontekście pozostałych już sam korzeń drzewa może zawierać informacje na temat klucza.

Wnioskiem płynącym z tego stwierdzenia jest fakt, że nie każda operacja może być zdefiniowana tylko ze względu na budowę danego rodzaju drzewa *Trie*, czy tylko na charakterystykę kluczy. Jest to spowodowane tym, iż pewne operacje dostarczają cenne informacje dopiero po dobraniu odpowiedniej implementacji do danej budowy drzewa.

2.1.4.3. Prefiks

Operacje dotyczące prefiksów (ang. *prefix*) są metodami odpowiadającymi na pytania dotyczące początkowych fragmentów kluczy zawartych w drzewie, względem słowa będącego argumentem operacji. Prefiksem może być pełen klucz lub początkowa część klucza pasująca do argumentu – który jest słowem lub częścią słowa. Inaczej mówiąc operacje dotyczące prefiksu mówią o dopasowaniu - najdłuższym na jaki pozwala długość argumentu operacji.

Na wysokim poziomie abstrakcji reprezentacji drzewa *Trie* o prefiksie można mówić w następujący sposób. Węzeł na poziomie L (gdzie korzeń drzewa znajduje się na poziomie 0) reprezentuje ostatni element sekwencji znaków o długości L , która jest nazywana prefiksem omawianego węzła. Prefiks niekoniecznie musi być całym słowem znajdującym się w drzewie *Trie*, a jedynie jego fragmentem.

Algorytmy opisane przez Knuth'a oraz wyjaśnione szerzej w tej pracy w rozdziale dotyczącym drzewa *Patricia* 2.2.4, a dokładniej Algorytm P 2.2.4.1 oraz Rozszerzenie Algorytmu P (Wyszukiwanie zbioru węzłów) 2.2.4.2 – które są wspomniane w sekcji 2.1.4.5 – odpowiednio: opowiadają na pytanie czy argument jest prefiksem, któregoś z kluczy oraz które dokładnie klucze pozwalają argumentowi być prefiksem - a nie koniecznie całym kluczem.

2.1.4.4. Czy-należy

Operacja czy-należy (ang. *look-up*) – w kontekście klucza – odpowiada na pytanie, czy element x zawiera się w zbiorze S ($x \in S$). W kontekście drzew *Trie* i *Patricia* należy interpretować tę operację jako odpowiedź na pytanie – czy wewnątrz drzewa istnieje klucz, którego ciąg znaków jest równy argumentowi metody?

Operacja *look-up* ma swoje zastosowanie również w kontekście prefiksu. Wtedy odpowiada na pytanie czy argument wyszukiwania jest równy początkowemu fragmentowi klucza. Mówiąc tutaj o początku mamy na myśli fragment klucza zaczynający się pierwszym znakiem klucza, o długości takiej samej jak argument operacji *look-up*. Pozytywny wynik tej operacji w takiej sytuacji może być spowodowany jednym lub większą ilością elementów zawartych w zbiorze S .

2.1.4.5. Szukaj

Operacja szukaj (ang. *search*) jest operacją, która ma w szczególności zastosowanie wewnątrz drzew *Patricia*. Ten rodzaj drzew przedstawiony jest w podrozdziale 2.2.4, w którym szerzej opisuję rozróżnienie pomiędzy operacją *look-up* oraz *search*.

Należy zauważyć, że obie operacje mają swoje zastosowanie w kontekście kluczy jak i prefiksów.

Różnica w obu wyszukiwaniach wynika ze sposobu reprezentacji kluczy, prefiksów oraz ponownego wykorzystania algorytmu operacji *look-up* w algorytmach innych operacji. Używając operacji *look-up* w kontekście prefiksu uzyskuje się odpowiedź na pytanie, czy argument metody jest taki sam jak początek, któregośkolwiek z kluczy. Wiele kluczy w tym rodzaju drzewa *Patricia* może rozpoczynać się w taki sam sposób, ale każdy z nich jest unikalny.

Z kolei w kontekście klucza operacja *look-up* odpowiada na pytanie czy cały klucz jest taki sam jak cały argument przeszukiwania.

Konsekwencją tego, że operacja *look-up* dla kryterium prefiksu może dawać wynik pozytywny ze względu na wiele kluczy jest potrzeba oddzielnej operacji szukającej kluczy spełniających wcześniej opisany warunek wobec argumentu. Operacja *search* odpowiada na pytanie – jeśli element x spełnia warunek prefiksu dla jednego z elementów zbioru S , to to dla których konkretnie kluczy (jednego lub większej ilości) argument przeszukiwania spełnia wymagania prefiksu.

Oczywiście operacja *look-up* dla kryterium klucza daje wynik pozytywny tylko wtedy, gdy istnieje dokładnie jeden klucz identyczny do argumentu wyszukiwania, a więc wynikiem operacji *search* jest węzeł odwzorowujący klucz pozwalający operacji *look-up* na zwrócenie pozytywnego wyniku.

W sekcji 2.2.4.1 został opisany algorytm dla operacji *look-up* w kontekście prefiksu, a w sekcji 2.2.4.2 algorytm dla operacji *search* - również w kontekście prefiksu.

2.1.4.6. Usun

Operacja usun (ang. *delete*) jest operacją usuwającą element x ze zbioru S . W kontekście drzew *Trie* oraz *Patricia* należy interpretować tę operację jako usunięcie z nich klucza. Efektem tej metody jest eliminacja z drzewa wszystkich węzłów, które były potrzebne do reprezentacji tylko danego klucza.

2.1.4.7. Poprzednik i Następca

Operacje poprzednik (ang. *predecessor*) oraz następca (ang. *successor*) są parą przeciwnych do siebie operacji.

Operacja poprzednik jest metodą znajdującą największy element y zbioru S , który jest mniejszy niż x element będący argumentem metody. Następca znajduje najmniejszy element z , większy niż x . „Wielkość” elementów zbioru S w tym kontekście jest poziomem na jakim znajdują się w drzewie węzły reprezentujące dany element.

W kontekście drzew *Trie* oraz *Patricia* należy interpretować operację poprzednika jako znalezienie najbliższego przodka (rodzica) węzła, którego prefiksem jest argument metody. Następca węzła, którego prefiksem jest argument metody jest najbliższy węzeł potomek (dziecko).

Dla uproszczenia mówimy tutaj o węzłach, ale chodzi oczywiście o prefiksy (lub klucze) zawarte w tych węzłach.

Innymi słowy poprzednik i następca są odpowiednio prefiksami węzłów na poziomach $L - 1$ i $L + 1$, gdy x jest węzłem na poziomie L akceptującym jako prefiks argument tych operacji.

Ze względu na różnicę w budowie drzewa *Patricia* względem innych struktur *Trie*, operacja poprzednik i następnik wyróżnia się. Podczas, gdy w pozostałych strukturach poprzednik i następnik mogą być prefiksami, w drzewie *Patricia* węzły zawsze reprezentują klucze. Dlatego poprzednik i następnik mogą być tylko kluczami w tym przypadku.

Oczywiście możliwa byłaby implementacja tych działań, tak aby dawały wynik podobny do spodziewanych wyników użycia tych operacji na innych rodzajach drzewa *Trie*, ale nie miałby to wiele wspólnego z budową drzewa *Patricia*. Na przykład operacja poprzednika mogłaby zwracać argument wyszukiwania krótszy o 1, o ile został on dopasowany jako prefiks, któregoś z kluczy w węzłach. Należy się zastanowić jednak czy daje nam to nowe informacje lub możliwości - czy robimy to tylko ze względu na zgodność z wysoko abstrakcyjnym wyobrażeniem tego działania.

2.1.4.8. Minimum i Maksimum

Operacje minimum i maksimum (ang. *maximum*) podobnie jak poprzednik i następca są metodami przeciwnymi do siebie.

Obie operacje są operacjami bezargumentowymi oraz odpowiadają na pytanie, jaki jest najmniejszy lub największy element zbioru S . Wielkość elementów zbioru S w tym kontekście jest określona długością kluczy zawartych w węzłach drzewa.

W kontekście drzew *Trie* i *Patricia* minimum znajduje najkrótszy klucz, a maksimum znajduje najdłuższy klucz od korzenia. Różnicą w tych dwóch rodzajach drzewa jest fakt, że w drzewie *Patricia* węzły najbliższe korzenia nie zawierają najkrótszych kluczy, a węzły najdalej od korzenia nie zawierają najdłuższych.

Ze względu na różnicę w budowie drzewa *Patricia* te dwie operacje można dwojako interpretować. Minimum może być – tak jak w przypadku pozostałych omawianych struktur *Trie* – najkrótszym kluczem, albo kluczem wymagającym najmniejszej ilości przejść, aby go znaleźć. Podobnie maksimum, może być albo najdłuższym kluczem, albo kluczem wymagającym największej ilości czasu, aby go dopasować do argumentu przeszukiwania. Dla ścisłości nazewnictwa operacji należy jednak przyjąć, że minimum to zawsze najkrótszy klucz, a maksimum – najdłuższy – a dla innych interpretacji tych operacji należy znaleźć nowe nazwy.

2.2. Wariacje drzew *Trie*

Zgodnie z wcześniejszymi stwierdzeniami w sekcji 2.1 termin *Trie* pojawia się w literaturze w wielu miejscach przynajmniej jako fragment hasła lub odwołanie. Można odnieść wrażenie, że jest to termin tak trywialny, że nie wymaga on przytoczenia dokładniejszej definicji, gdyż tak rzadko autor pracy się podejmuje tego przedsięwzięcia. Dzięki dogłębnej analizie problemu, opisanego przez Donalda Knuth'a [1], byliśmy w stanie ułożyć w zorganizowany sposób wiedzę dotyczącą drzew *Trie* i przedstawić ją w tym rozdziale w jak najbardziej przystępny sposób, roz dzielając go na części ze względu na różne rodzaje drzewa *Trie*.

2.2.1. Drzewo *Trie* – dwuwymiarowa tablica jako reprezentacja

Schemat tablicy dwuwymiarowej reprezentującej drzewo *Trie* dla 31 najpopularniejszych słów angielskich zaprezentowany za pomocą tablicy 2.1 pozwala nam uzyskać odpowiedź na pytanie, czy dany argument przeszukiwania należy do zbioru 31 najbardziej popularnych słów angielskich. Przeszukiwanie takiej tablicy oparte jest o powtarzające się podciągi znaków (ang. *substrings*). Reprezentacją graficzną tej tablicy jest schemat pokazany na rysunku 2.1.

Tablicę 2.1 należy interpretować w następujący sposób. Drzewo reprezentowane przez tablicę posiada 12 węzłów. 1^W jest węzłem korzeniem drzewa. Rozpoczynając algorytm przeszukiwania, bierzemy pod uwagę kolumnę korzenia drzewa – 1^W . Puste komórki tablicy interpretujemy jako brak gałęzi lub jako gałąź wskazującą na węzeł *null* odpowiadający nieakceptowanemu słowu, które nie znajduje się w naszym drzewie *Trie*.

Biorąc przykładowe słowo NOT jako argument przeszukiwania tablicy 2.1 patrzymy na przecięcie wiersza N^K , który odpowiada znakowi klucza N oraz kolumny 1^W . Informację znaną w tej komórce należy interpretować jako istnienie jedyne go możliwego pasującego słowa zaczynającego się od litery N.

Tablica 2.1. Schemat tablicy dwuwymiarowej reprezentującej drzewo *Trie* dla 31 najpopularniejszych słów angielskich zasięgnięty z książki Knuth'a [1]. Tytuły kolumn są identyfikatorami węzłów, a tytuły wierszy są identyfikatorami gałęzi (znaków pozwalających na przejście z jednego węzła do drugiego). Znak ‘_’ oznacza brak znaku przy przejściu z jednego węzła do drugiego (nie potrzebne jest przetworzenie znaku, aby wykonać przejście do następnego węzła). W komórkach tabeli można znaleźć identyfikator węzła, do którego należy przejść po napotkaniu odpowiedniego znaku lub pełne słowo (klucz), co oznacza, że jest to jedyny klucz zaczynający się danym ciągiem znaków.

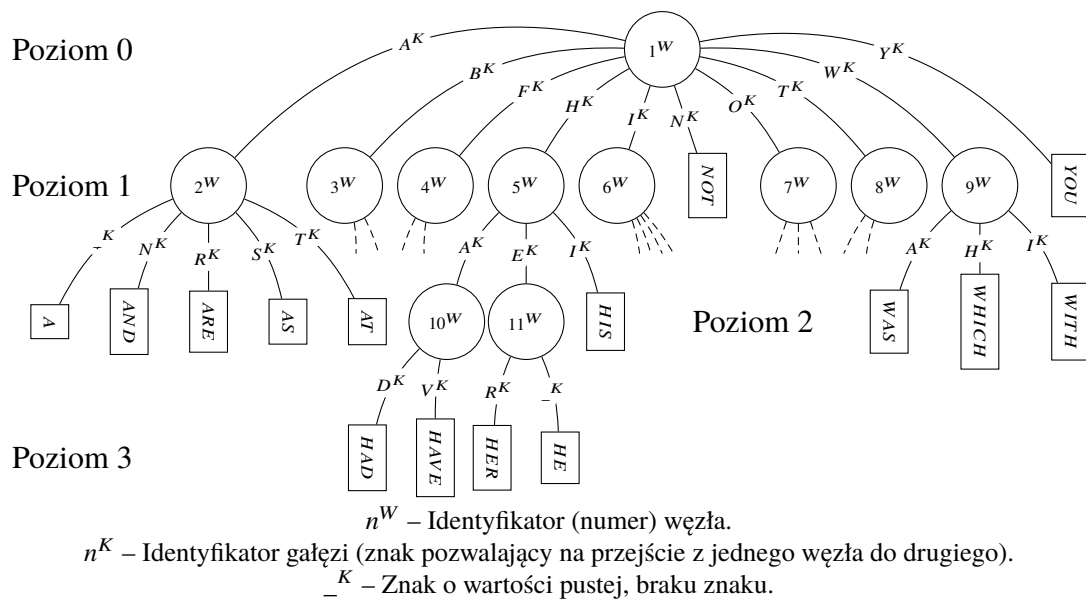
	1 ^W	2 ^W	3 ^W	4 ^W	5 ^W	6 ^W	7 ^W	8 ^W	9 ^W	10 ^W	11 ^W	12 ^W
_ ^K		A				I					HE	
A ^K	2 ^W				10 ^W				WAS			THAT
B ^K	3 ^W											
C ^K												
D ^K										HAD		
E ^K			BE		11 ^W							THE
F ^K	4 ^W						OF					
G ^K												
H ^K	5 ^W							12 ^W	WHICH			
I ^K	6 ^W				HIS				WITH			THIS
θ ^K												
J ^K												
K ^K												
L ^K												
M ^K												
N ^K	NOT	AND				IN	ON					
O ^K	7 ^W			FOR				TO				
P ^K												
Q ^K												
R ^K		ARE		FROM			OR				HER	
Φ ^K												
Π ^K												
S ^K		AS				IS						
T ^K	8 ^W	AT				IT						
U ^K			BUT									
V ^K										HAVE		
W ^K	9 ^W											
X ^K												
Y ^K	YOU		BY									
Z ^K												

n^W – Identyfikator (numer) węzła.

n^K – Identyfikator gałęzi (znak pozwalający na przejście z jednego węzła do drugiego).

_^K – Znak o wartości pustej, braku znaku.

Rysunek 2.1. Uproszczony schemat reprezentujący graficznie abstrakcyjną strukturę drzewa *Trie* stworzoną na podstawie tabeli 2.1. Schemat zawiera wszystkie węzły za wyjątkiem poddrzew węzłów 3^W , 4^W , 6^W , 7^W , 8^W . Okrągłe węzły oznaczają węzły nie-akceptujące (dane słowo nie kończy się w węźle), a prostokątne węzły są akceptującymi dane słowo. Krawędzie ciągle są opisanymi zależnościami z tabeli 2.1, a przerywane oznaczają pominięte zależności - ze względu na uproszczenie. Symbole wewnątrz węzłów w formacie n^W są identyfikatorami węzłów, a symbole przy krawędziach w formacie n^K są identyfikatorami gałęzi (znaków pozwalających na przejście z jednego węzła do drugiego). Znak ‘_’ oznacza brak znaku przy przejściu z jednego węzła do drugiego.



Jeżeli rozważymy przykład słowa **WHERE**, komórka o współrzędnych (W^K ; 1^W) nakazuje nam przejście do węzła 9^W , a z kolei komórka (H^K ; 9^W) nie pozostawia nam wątpliwości, że jedynym słowem pasującym do parametrów przeszukiwania naszej struktury *Trie* przetworzonych do tej pory jest **WHICH**. Oznacza to, że **WHERE** nie znajduje się w naszym zbiorze 31 najczęściej używanych angielskich słów. Można dodatkowo zauważyć, że prefiksem węzła 9^W jest sama litera **W**.

Ciekawszym przykładem jest słowo **HAVE**. Próbując znaleźć to słowo w naszej strukturze *Trie* przechodzimy przez następujące komórki: (H^K ; 1^W), (A^K ; 5^W), (V^K ; 10^W) i rzeczywiście znajdujemy pożądane słowo **HAVE**. Dzięki pomocy tablicy 2.1 lub schematu znajdującym się na rysunku 2.1 łatwo zauważyć, że prefiksem węzła 10^W jest **HA**.

Warto również zwrócić uwagę na schemat drzewa 2.1 ze względu na unikalną gałąź $_K$ z węzła 2^W do węzła **A** w przypadku naszej struktury *Trie*. Prefiksem węzła 2^W jest litera **A**.

Knuth podkreśla w swojej książce [1], że wektory węzłów w tablicy 2.1 są ułożone w kolejności odpowiadającej kolejności ułożenia znaków w maszynie *MIX*. Miało to szczególne znaczenie w momencie, kiedy pisał on swoją książkę, gdyż język *MIX* przypominał kod maszynowy komputerów w czasie jego opracowania – przed 1969 rokiem.

Wszystko to miało wpływ na zwiększenie prędkości przeszukiwania drzewa *Trie*, ponieważ postępując zgodnie z algorytmem Knuth’a jedynie wyciągamy słowa z tablicy, używając znaków naszych kluczy jako indeksów, a nie przeszukujemy wektory, aby znaleźć odpowiednią w niej wartość. Aspektem, który stara się tutaj podkreślić Knuth jest różnica pomiędzy przeszukiwaniem tablicy (ang. *table look-up*), a ”zaglądaniem do tablicy” (ang. *table look-at*) w ściśle określone miejsce.

2.2.1.1. Algorytm T – Przeszukiwanie *Trie*

Inaczej określane w polskiej wersji książki Knuth’a jako „wyszukiwanie w drzewie *Trie*” – algorytm wyszukuje argument K w posiadanej tablicy.

Założenia:

Dana jest tablica wektorów o M elementach, tworzącą M -ary drzewo. Indeksy wektorów zaczynają się od 0, a kończą na $M - 1$. Elementem tych wektorów może być klucz – całe słowo znajdujące się w tablicy – lub odniesienia do następnego wektora. Jeżeli element takiego wektora jest pusty, równy *null*, oznacza to, że słowo, które obecnie próbujemy dopasować nie istnieje wewnątrz naszej struktury *Trie*. Jeżeli jednak znajdziemy całe słowo – klucz – i jest ono takie samo jako ciąg znaków, który próbujemy dopasować. Oznacza to, że argument przeszukiwania K należy do naszego drzewa *Trie*.

1. Inicjalizacja

Ustaw P – odniesienie do obecnego wektora – tak, aby wskazywało na korzeń drzewa.

2. Rozgałęzienie

Ustaw k – kod naszego obecnie przetwarzanego znaku z argumentu wejściowego K – na kod znaku następnego w kolejności od lewej do prawej.

Jeżeli argument wejściowy K został przetworzony w całości należy ustawić k na znak pusty lub znak odpowiadający końcowi słowa.

Reprezentacja znaku powinna przyjmować wartość w zakresie $0 \leq k < M$. Jest to liczba całkowita oznaczająca kod znaku.

X jest wartością komórki o indeksie k wektora P . $P(k) = X$

(a) Jeżeli X jest odniesieniem do kolejnego wektora – idź do punktu 3.

(b) Jeżeli X jest kluczem – idź do punktu 4.

3. Postęp

(a) Jeżeli X jest odniesieniem do jednego z wektorów, nie jest pustym, *null* wskaźnikiem – podstaw X pod P i idź do punktu 2.

- (b) Jeżeli X nie jest odniesieniem do jednego z wektorów – zakończ z niepowodzeniem.

4. Porównanie

- (a) Jeżeli $X = K$, jest całym argumentem wejściowym – zakończ z powodzeniem.
- (b) Jeżeli $X \neq K$ – zakończ z niepowodzeniem.

Dodatkowo należy zwrócić uwagę na fakt, że jeżeli nie znajdziemy słowa w naszej strukturze to znajdujemy najdłuższy fragment pasujący – najdłuższe dopasowanie.

2.2.1.2. Przeznaczenie

Knuth w celu porównania omawianych algorytmów tworzy krótki program na maszynie *MIX*, który jest implementacją algorytmu T 2.2.1.1. Dzięki wykorzystaniu maszyny *MIX* może on przyjąć wiele założeń, które mogą znacząco wpłynąć na wyniki testów wydajnościowych.

Przed wszystkim Knuth zakłada, że pojedyncze znaki mają długość jednego bajtu, a klucze mają co najwyżej długość 5 bajtów. Niestety posługując się językiem *Java* – gdzie na *char* czy pojedynczy znak w *Stringu* do przetrzymywania w pamięci potrzebne są 2 bajty – jesteśmy z góry ograniczeni założeniami języka.

Dodatkowo Knuth przyjmuje, że każdy z kluczy reprezentowany jest jako jedno słowo – o długości 5 znaków – zakodowane przez maszynę *MIX*, gdzie puste, niewypełnione przez klucz miejsca zastąpione są przez znak odstępu – spację. Puste miejsca znajdują się po prawej stronie, a klucz zaczynamy wpisywać od lewej strony słowa.

Ułatwieniem również jest kodowanie znaków *MIX* z czym wiąże się reprezentacja każdego z bajtów argumentu przeszukiwania K w postaci numeru mniejszego od 30. Ponownie *Java* różni się tutaj reprezentacją w kodzie znaków w standardzie UTF-16. Szczególnym problemem jest tutaj sposób dowiązań przyjęty przez Knuth'a, który reprezentowany jest poprzez liczby ujemne w zbiorze $\{0, 1, 2\}$ słowa odpowiadającego węzłowi.

2.2.1.3. Złożoność czasowa i pamięciowa

Najważniejsze tutaj są jednak wnioski, do jakich doszedł Knuth. Wykonywanie jego programu zajmuje $8 * C + 8$ jednostek czasu, gdzie C jest ilością znaków przetworzonych. Jako że $C \leq 5$ to przeszukiwanie nigdy nie zajmie dłużej niż 48 jednostek czasu.

Struktura *Trie* zajmuje natomiast znacznie więcej pamięci. Używamy 360 słów do reprezentacji 31 kluczy. Dla porównania, w przypadku gdy kontenerem przechowującym dane byłoby drzewo binarne, użylibyśmy 62 słów.

Na szczęście Knuth przedstawił pomysł na rozwiązanie tego problemu w postaci reorganizacji struktury tablicowej do skompresowanej tablicy dwuwymiarowej.

Porównując Algorytm T 2.2.1.1 do programu optymalnego przeszukiwania binarnego drzewa dla tego samego zbioru argumentów przeszukiwania K – Knuth zauważa, że:

- Program implementujący algorytm T oraz optymalne przeszukiwanie binarnego drzewa w przypadku sukcesu zajmują 26 jednostek czasu.
- Przeszukiwanie *Trie* jest zdecydowanie szybsze w przypadkach kończących się niepowodzeniem. Dodatkowo w sytuacji, w której użyjemy omawianych danych, dużo częściej będziemy mieć do czynienia z nieznalezieniem poszukiwanego słowa w naszym zbiorze.
- Knuth dodatkowo nawiązuje do odmiennego podejścia do problemu w postaci indeksowania KWIC (ang. *keyword in context*)*. Zauważa on, że w tym przypadku zastosowanie *Trie* nie ma sensu w klasycznym znaczeniu z powodu natury danych zawartych w strukturze. Ponownie przedstawia nietuzinkowe rozwiązanie jakim jest przetwarzanie słów od prawej strony do lewej, tak aby odróżnienie słów takich jak COMPUTATION i COMPUTATIONS nie zajmowało 12 iteracji.

2.2.1.4. Drzewo *Trie* reprezentowane przez skompresowaną tablicę dwuwymiarową

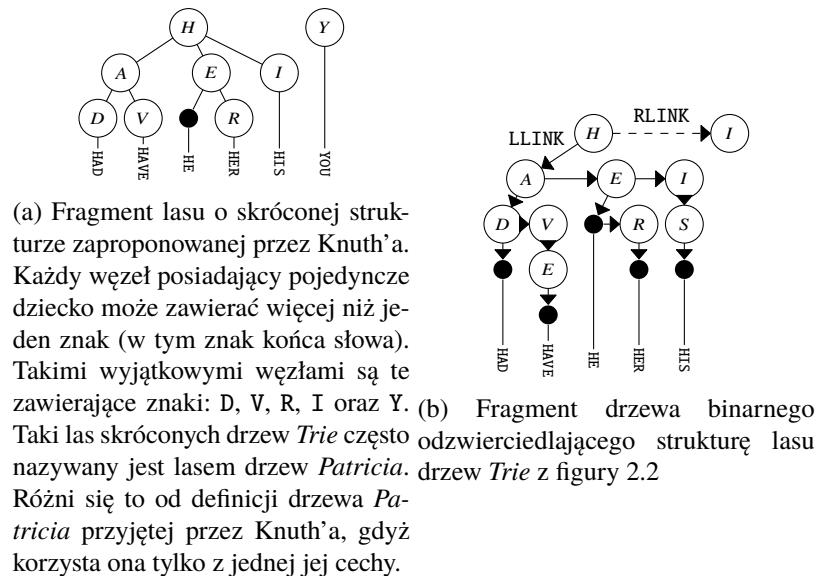
Skompresowana tablica dwuwymiarowa powstaje przez nakładanie komórek zajętych na te, które są puste. W ten sposób otrzymujemy tablicę postaci przedstawionej w poniższej tablicy 2.2.

Do skompresowanej w ten sposób tablicy potrzebny jest oddzielny algorytm przeszukiwania. Knuth opisuje ten algorytm w jednym z wcześniejszych rozdziałów w postaci programu napisanego dla maszyny *MIX* („Program T“ w rozdziale 6.2.2. „Wyszukiwanie w drzewach binarnych“ - ang. „*Binary Tree Searching*“ [1]). Nie będzie on jednak działał tak szybko, jak algorytm dla nieskompresowanej tablicy. Należy jednak wziąć pod uwagę fakt, że większa część z 360 komórek w tablicy 2.1 jest pusta, a przekształcając tablicę do formy skompresowanej, zmniejszamy ilość wszystkich komórek do 49.

*Wykorzystanie w kontekście optymalnego przeszukiwania binarnego w *Sztuce Programowania* na stronach 560 - 561 [1]. Pomysł KWIC zaproponowany w H. P. Luhn, Amer. Documentation 11 (1960), 288–295. Pełne indeksowanie KWIC znajduje się w W. W. Youden, JACM 10 (1963), 583–646.

Tablica 2.2. Schemat skompresowanej tablicy dwuwymiarowej reprezentującej drzewo *Trie* dla 31 najpopularniejszych słów angielskich. Tablica powstała na podstawie tablicy 2.1 poprzez nałożenie komórek zajętych na te, które są puste. Identyfikatory węzłów w tej tabeli (pojawiające się w wierszach „Start węzła” bądź „Zawartość”) odpowiadają identyfikatorom z tabeli 2.1. W wierszach „Zawartość” oprócz identyfikatorów węzłów można znaleźć klucze (pełne słowa). Tablica jest zasięgnięta z książki Knuth’a [1].

Pozycja	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Start węzła	5 ^W 7 ^W		3 ^W 9 ^W	12 ^W			8 ^W							4 ^W
Zawartość		10 ^W		W A S	T H A T	11 ^W	O F	B E	T H E	H I S	W H I C H	W I T H	T H I S	
Pozycja	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Start węzła			6 ^W	11 ^W	2 ^W	1 ^W 10 ^W								
Zawartość	12 ^W	O N	I	H E	A	O R	2 ^W	3 ^W	T O	H A D		4 ^W	B U T	5 ^W
Pozycja	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Zawartość	6 ^W	F O R	B Y	I N	F R O M	A N D	N O T	7 ^W	H E R	A R E	I S	I T	A S	A T
Pozycja	43	44	45	46	47	48	49							
Zawartość	8 ^W		H A V E	9 ^W		Y O U								



Rysunek 2.3. Porównanie fragmentów: (a) skróconego lasu *Trie* oraz (b) binarnego drzewa. Oba rysunki zasięgnięte są z książki Knuth'a [1].

Reprezentacja w postaci drzewa binarnego wiąże się z jego przeszukiwaniem poprzez porównywanie znaku argumentu ze znakiem w drzewie. Jeżeli znaki sobie nie odpowiadają to podążamy za *RLINK* – wskaźnikiem oznaczającym prawą gałąź, a gdy pasują do siebie przechodzimy do węzła wskazanego przez *LLINK* – wskaźnikiem oznaczającym lewą gałąź. Postępujemy tak oczywiście od pierwszego znaku argumentu do ostatniego.

Takie przeszukiwanie jest określane przez Knuth'a *rozgałęzieniem ze względu na porównanie równe-nierówne* (ang. *equal-unequal branching*), które zostaje wykorzystane w miejsce *rozgałęzienia mniejsze-większe* (ang. *less-greater branching*).

2.2.2.1. Złożoność czasowa i pamięciowa oraz przeznaczenie i możliwe modyfikacje w ich kontekście

Wynikająca z tego charakterystyka złożoności obliczeniowej mówi o minimalnej ilości porównań wynoszącej:

$$\lg(N) = \log_2(N) \quad (2.1)$$

gdzie

N – ilość kluczy

Przeszukiwanie drzewa przedstawionego przez rysunek 2.1 średnio wymaga co najmniej takiej samej ilości porównań jak w przypadku przeszukiwania drzewa binarnego. Jednak w drzewie *Trie* z tabeli 2.1 z jednego węzła może wychodzić M rozgałęzień.

Dla dużych ilości kluczy N i dla danych zrandomizowanych skutkuje to czasem przeszukiwania określonych zależnością:

$$\log_M(N) = \frac{\lg(N)}{\lg(M)} \quad (2.2)$$

gdzie

N – ilość kluczy

M – ilość gałęzi wychodzących z każdego węzła, znaków rozważanego alfabetu

Knuth zwraca dodatkowo uwagę na fakt, iż nazywana przez niego „czysta” struktura *Trie* – jak ta zaprezentowana w przypadku omawianego Algorytmu T w sekcji 2.2.1.1 – wymaga około

$$\frac{N}{\ln(M)} \quad (2.3)$$

węzłów by rozróżnić N losowych kluczy podanych jako argument danych wejściowych, a co za tym idzie całkowita ilość pamięci jest proporcjonalna do wzoru:

$$\frac{MN}{\ln(M)} \quad (2.4)$$

Wnioskiem wyciąganym z tej analizy przez Knuth’a jest fakt, że użycie struktury *Trie* zaczyna mieć sens tylko w kilku pierwszych poziomach drzewa. Proponuje on wymieszanie dwóch strategii – używanie *Trie* przez kilka pierwszych znaków argumentu przeszukiwania, a potem zmianę struktury. Knuth przytacza technikę opisaną przez E. H. Sussenguth, Jr. [CACM 6 (1963), 272–279], gdzie następuje zmiana strategii, gdy pozostaje 6 lub mniej możliwych kluczy na przechodzenie przez krótką listę pozostałych kluczy. Skutkuje to zmniejszeniem ilości węzłów *Trie* sześciokrotnie bez znaczących zmian czasu wykonywania przeszukiwania.

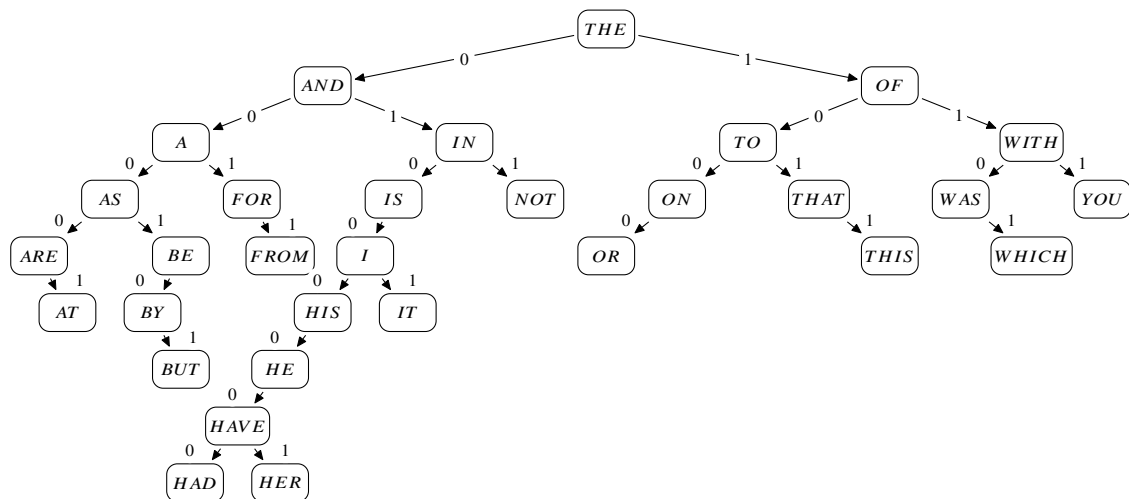
Knuth w swojej książce wyróżnia jeszcze dwa ciekawe pomysły na optymalizację algorytmu zaproponowane przez innych autorów. Pierwszym z nich jest sposób na przetrzymywanie dużych i rosnących drzew w zewnętrznej pamięci autorstwa S. Y. Berkovich w Dokłady Akademii Nauk SSSR 202 (1972), 298 - 299 [English translation in Soviet Physics–Dokłady 17 (1972), 20–21]. Drugim jest uwaga z pracy T.N. Tuba [CACM 25 (1982), 522–526], dotycząca tego, że czasem najwygodniej jest szukać po ilości kluczy zmiennej argumentu przeszukiwania, posiadając po jednym drzewie lub drzewie *Trie* na każdą długość.

2.2.3. Przypadek binarny – Przeszukiwanie cyfrowe drzewa oraz wstawianie

Przypadek binarny to pojęcie, którego Knuth używa do opisanie rodzaju drzewa binarnego, czyli drzewa M -ary dla $M = 2$, w którym przeszukiwanie polega na skanowaniu bitów po kolei. Metoda przeszukiwania takiego drzewa nazywana jest *przeszukiwaniem drzewa cyfrowego* i została opracowana przez E. G. Coffman and J. Eve [CACM 13 (1970), 427–432, 436]. W węzłach drzewa przetrzymywane są pełne klucze, czyli całe słowa. O tym, którą gałąź drzewa należy wybrać, aby znaleźć odpowiednie słowo, decyduje reprezentacja binarna poszczególnych znaków argumentu przeszukiwania.

Rysunek 2.4 przedstawia drzewo binarne, nazywane też drzewem przeszukiwania cyfrowego lub drzewem cyfrowym, stworzone poprzez wstawianie słów w kolejności zależnej od ich częstotliwości występowania. Najpopularniejsze słowa są umieszczone jak najbliżej korzenia drzewa, a mniej popularne głębiej, ale nadal w zgodności z wcześniej opisaną metodą wyboru gałęzi.

Rysunek 2.4. Schemat drzewa przeszukiwania cyfrowego dla 31 najpopularniejszych słów angielskich. Drzewo przeszukujemy odwołując się do wartości binarnej słowa szukanego i analizujemy każdy jego bit zaczynając od lewej strony. Dla 0 idziemy do lewego dziecka a dla 1 do prawego. Rysunek zasięgnięty jest z książki Knuth’a [1].



Jeżeli chcielibyśmy w tym drzewie znaleźć słowo WHICH, to wchodząc do drzewa, porównalibyśmy cały argument przeszukiwania z całym kluczem węzła THE. Nie znajdując dopasowania, musielibyśmy wybrać jedną z gałęzi. Skoro pierwszym bitem reprezentacji binarnej poszukiwanego słowa jest 1 musimy wybrać gałąź prawą. Ponownie należy porównać ze sobą całe słowa węzła i argumentu przeszukiwania – WHICH i OF – wybrać gałąź prawą ze względu na drugi bit równy 1. Proces powtarzamy dla węzła WITH, wybieramy gałąź lewą dla trzeciego bitu równego 0 i ponownie porównujemy z WAS. Ostatni raz analizujemy bit – tym razem czwarty – równy 1 wybierając gałąź prawą. Skoro nie ma już dalszych gałęzi, ostatnie porównanie decyduje o tym, czy dane słowo znajduje się w naszym drzewie, czy jednak go tam nie ma. Na szczęście tym razem słowa argumentu przeszukiwania i węzła pasują do siebie, czyli udało nam się znaleźć poszukiwane słowo. Jeżeli jednak tak by się nie stało, to w przypadku wstawiania musielibyśmy stworzyć nowy węzeł zawierający pełny klucz argumentu przeszukiwania, a następnie wybrać odpowiednią gałąź na podstawie piątego bitu, równego 0 – czyli gałąź lewą – która wskazywałaby na nowo stworzony węzeł.

Tabela 2.3 przedstawia reprezentację binarną wszystkich słów w drzewie zgodnie z kodowaniem znaków maszyny MIX Knuth’a [18]. Słowa ułożone są w kolejności występowania w drzewie, czytane od lewej do prawej, poziom po poziomie. Korzeniem jest najczęściej występujące słowo THE, którego częstotliwość występowania wynosi 15586. Następnym słowem w tablicy jest AND, choć OF ma od niego większą częstotliwość występowania. Jest to związane ze strukturą drzewa, które budowane jest w oparciu o reprezentację binarną poszczególnych słów. Choć częstotliwość występowania jest priorytetowym kryterium w kontekście tworzenia drzewa przeszukiwania cyfrowego, to ze względu na jego charakterystykę, nie zawsze najczęściej występujące słowa będą znajdowały się wysoko w tabeli.

Tablica 2.3. Tablica 31 najpopularniejszych angielskich słów, ich częstotliwości wystąpień oraz reprezentacji binarnej w maszynie Knuth’a *MIX*. Kolumna zawierająca reprezentację dziesiętną jest wynikiem zamiany poszczególnych znaków w słowie na wartości, które Knuth przypisuje znakom w swojej maszynie *MIX* [18]. Np. dla litery T wynosi ona 23, dla H – 8, a dla E – 5. Reprezentacja binarna jest tworzona na podstawie zamiany liczb z reprezentacji dziesiętnej na wartości binarne. Tablica zasięgnięta jest z książki Knuth’a [1].

Słowo	Częstotliwość występowania	Reprezentacja dziesiętna	Reprezentacja binarna
THE	15586	23 8 5	10111 01000 00101
AND	7638	1 15 4	00001 01111 00100
OF	9767	16 6	10000 00110
A	5074	1	00001
IN	4312	9 15	01001 01111
TO	5739	23 16	10111 10000
WITH	1849	26 9 23 8	11010 01001 10111 01000
AS	1853	1 22	00001 10110
FOR	1869	6 16 19	00110 10000 10011
IS	2509	9 22	01001 10110
NOT	1496	15 16 23	01111 10000 10111
ON	1155	16 15	10000 01111
THAT	3017	23 8 1 23	10111 01000 00001 10111
WAS	1761	26 1 22	11010 00001 10110
YOU	1336	28 16 24	11100 10000 11000
ARE	1222	1 19 5	00001 10011 00101
BE	1535	2 5	00010 00101
FROM	1039	6 19 16 14	00110 10011 10000 01110
I	2292	9	01001
OR	1101	16 19	10000 10011
THIS	1021	23 8 9 22	10111 01000 01001 10110
WHICH	1291	26 8 9 3 8	11010 01000 01001 00011 01000
AT	1053	1 23	00001 10111
BY	1392	2 28	00010 11100
HIS	1732	8 9 22	01000 01001 10110
IT	2255	9 23	01001 10111
BUT	1379	2 24 23	00010 11000
HE	1727	8 5	01000 00101
HAVE	1344	8 1 25 5	01000 00001 11001 00101
HAD	1062	8 1 4	01000 00001 00100
HER	1093	8 5 19	01000 00101 10011

2.2.3.1. Algorytm D – Wstawianie

Założenia:

Dana jest tablica rekordów tworzących binarne drzewo opisane tablicą 2.4 oraz tablicą 2.3. Algorytm przeszukuje drzewo w celu znalezienia danego argumentu K . Jeżeli K nie znajduje się w tablicy, nowy węzeł zawierający argument K zostanie dodany do drzewa w odpowiednie miejsce. Zakładamy, że drzewo jest niepuste i każdy z węzłów posiada KEY , $LLINK$ – będący wskaźnikiem, reprezentującym lewą gałąź - oraz $RLINK$ – również będący wskaźnikiem, reprezentującym prawą gałąź.

1. Inicjalizacja

Ustaw P – nasze odniesienie do obecnie przetwarzanego węzła – tak, aby wskazywało na korzeń drzewa.

Ustaw K' – nieprzetworzone bity reprezentacji bitowej argumentu przeszukiwania – na reprezentację bitową całego argumentu przeszukiwania K .

2. Porównanie

(a) Jeżeli K' jest $KEY(P)$ (kluczem P) – zakończ przeszukiwanie z powodzeniem.

(b) W przeciwnym wypadku:

Ustaw b na początkowy bit K' ,

Przesuń K' o jeden bit w lewo, czyli usuń najbardziej znaczący bit, przesuwając pozostałe bity oraz dokładając na puste miejsce, najmniej znaczącego bitu 0 [19].

i. Jeżeli $b = 0$ – idź do punktu 3.

ii. W przeciwnym wypadku – idź do punktu 4.

3. Ruch w lewo

(a) Jeżeli $LLINK(P)$ wskazuje na istniejący węzeł, nie jest pustym, $null$ wskaźnikiem:

Ustaw P na $LLINK(P)$,

Idź do punktu 2.

(b) W przeciwnym wypadku – idź do punktu 5.

4. Ruch w prawo

(a) Jeżeli $RLINK(P)$ wskazuje na istniejący węzeł, nie jest pustym, $null$ wskaźnikiem:

Ustaw P na $RLINK(P)$,

Idź do punktu 2.

(b) W przeciwnym wypadku – idź do punktu 5.

5. Wstawienie do drzewa

Stwórz nowy węzeł Q ,

Ustaw klucz nowego węzła Q na pełen argument przeszukiwania K ,

Ustaw $LLINK(Q)$ oraz $RLINK(Q)$ na $null$.

- (a) Jeżeli $b = 0$ to ustaw $LLINK(P)$ na nowy węzeł Q .
- (b) W przeciwnym wypadku ustaw $RLINK(P)$ na nowy węzeł Q .

2.2.3.2. Przeznaczenie

Głównym ograniczeniem tego algorytmu jest wymóg posiadania częstotliwości lub prawdopodobieństwa zapytania o każdy z zawartych w drzewie kluczy. Bez tej informacji struktura nie może być poprawnie zbudowana, gdyż zakłada ona dopasowanie do przypadku użycia.

W najgorszym wypadku budowa struktury może posiadać najczęściej wyszukiwane klucze oddalone maksymalnie od korzenia drzewa – tym samym zamiast skracać średni czas wyszukiwania, wydłuża go.

2.2.4. Drzewo *Patricia* – N-węzłowe drzewo przeszukiwań binarnej reprezentacji kluczy

Rozszerzeniem algorytmu 2.2.3.1 z binarnego na przypadek przeszukiwania cyfrowego drzewa M -ary, dla dowolnego $M \geq 2$ jest opracowana przez Donald R. Morrison [JACM 15 (1968), 514–534] [14] metoda “Patricia” (ang. *Practical Algorithm To Retrieve Information Coded In Alphanumeric*). Bardzo podobny algorytm został opublikowany w prawie tym samym czasie w Niemczech przez G. Gwehenberger, Elektronische Rechenanlagen 10 (1968), 223–226. *Patricia* jest to metoda tworzenia N -węzłowych drzew przeszukiwań oparta o binarną reprezentację kluczy, bez potrzeby przechowywania kluczy w węzłach. Jest ona szczególnie przystosowana dla długich, różnej długości kluczy – takich jak tytuły czy frazy zawarte w dużych zbiorach plikowych.

Zamysłem metody *Patricia* jest stworzenie binarnego drzewa, unikając tworzenia węzłów z jedną krawędzią wychodzącą, z jednym dzieckiem – czyli takich gałęzi, które nie są rozgałęzieniami, które nie powodują zawężenia ilości możliwych do podjęcia ścieżek, a ich jedyną funkcją jest przesunięcie obecnie analizowanego przypadku o jeden do przodu. Ominięcie takich przypadków jest osiągane przez zawarcie w każdym węźle numeru bitów, które należy pominąć przed przeprowadzeniem testu na następnym.

Istnieje wiele sposobów wykorzystania tego pomysłu w algorytmie. Knuth przedstawia jeden z nich w następujący sposób. Przygotowujemy tablicę bitów reprezentującą tekst, którą określamy później terminem *TEXT*. Taka tablica zazwyczaj jest długa, więc może być przetrzymywana w postaci pliku. Z powodu wolniejszego dostępu do pliku niż do pamięci RAM należy nałożyć na użytkownika ograniczenie ilości dostępów do tego pliku. Limit ten ustalony zostaje w wysokości jednego odwołania do zawartości pliku przy algorytmie sprawdzania czy dany ciąg znaków jest akceptowany przez nasze drzewo.

Każdy z kluczy przechowywany wewnątrz tablicy bitów *TEXT* jest określony numerem miejsca w tekście, w którym zaczyna się dany klucz. Należy sobie wyobrazić taki klucz jako ciąg znaków od miejsca startowego danego klucza, aż do końca tekstu. *Patricia* nie jest przeznaczona do znajdowania równości między kluczem, a argumentem przeszukiwania. *Patricia* jedynie odpowiada na pytanie czy istnieje klucz zaczynający się argumentem wejściowym.

Tablica 2.4 przedstawia przykład zawartości krótkiego pliku w postaci tekstowej. W tym przypadku kluczami przechowywanymi w tablicy *TEXT* są ciągi znaków będące fragmentami tekstu:

1. THIS IS THE HOUSE THAT JACK BUILT;
2. IS THE HOUSE THAT JACK BUILT;
3. THE HOUSE THAT JACK BUILT;
4. HOUSE THAT JACK BUILT;
5. THAT JACK BUILT;
6. JACK BUILT;
7. BUILT;

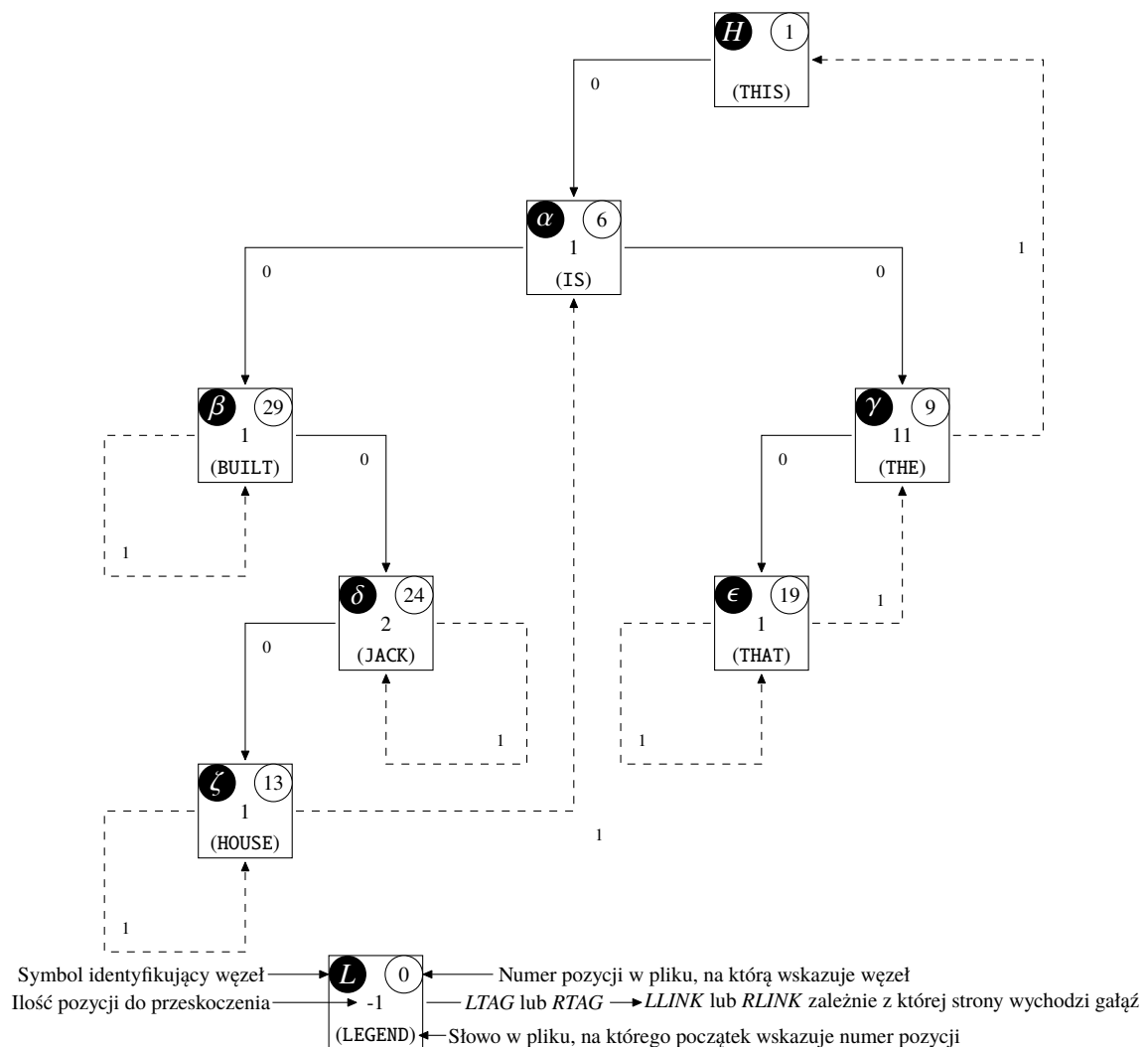
Tablica 2.4. Schemat tablicy przedstawiającej zawartość pliku tekstowego. W wierszu Zawartość przechowujemy poszczególne znaki a w wierszu Numer pozycji, pozycję znaku w pliku, liczoną od lewej i zaczynającą się od numeru 1. Tablica zasięgnięta jest z książki Knuth’a [1].

Zawartość	T	H	I	S		I	S		T	H	E		H	O	U	S	E
Numer pozycji	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Zawartość		T	H	A	T		J	A	C	K		B	U	I	L	T	;
Numer pozycji	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34

Średnik (;) pełni tutaj ważną funkcję – algorytm wymaga bowiem, aby żaden klucz nie był prefiksem innego. Można to osiągnąć w łatwy sposób – wstawiając na koniec naszego pliku znak unikalny, niepojawiający się nigdzie indziej. Taki sam wymóg istniał w Algorytmie T i tam rolę średnika pełnił znak końca słowa (wypełniona w środku, czarna kropka).

Drzewo *Patricia* przedstawione na rysunku 2.5 powinno być przetrzymywane w pamięci RAM. Składa się ono z N węzłów, w tym szczególnego, pierwszego węzła nazywanego przez Knuth’a *header’em*. Każdy z węzłów dla ułatwienia został oznaczony symbolem. *Header’owi* przypadła litera *H*, a pozostałym zostały przydzielone kolejne litery alfabetu greckiego – od α (alfa) do ζ (zeta). Drzewo zostało zbudowane w oparciu o tablicę 2.5.

Rysunek 2.5. Drzewo *Patricia*. Numery w prawym górnym rogu węzła odpowiadają pozycji w pliku, na którą wskazuje węzeł. W lewym górnym rogu znajduje się symbol identyfikujący węzeł. Na środku węzła podany jest numer, który oznacza ilość pozycji do przeskoczenia. Słowa w nawiasach, to słowa, na którego początek wskazuje numer pozycji w pliku. *TAG* to wartość, która wynosi 0 lub 1 i na schemacie znajduje się obok linii. Linia przerywaną oznaczone są linie, których wartość *TAG* wynosi 1 i prowadzą do przodka węzła. Linia ciągłą oznaczone są linie, których *TAG* wynosi 0 i prowadzą do dziecka. Linie wychodzące z lewej strony węzła noszą nazwę *LLINK* a z prawej *RLINK*. *TAG* przypisany do *LLINK* nazywamy *LTAG*, a do *RLINK* – *RTAG*. Rysunek zasięgnięty jest z książki Knuth’a [1].



Poniżej przedstawiony jest zestaw pól, z których składa się każdy z węzłów wraz z rodzajem zmiennej oraz przeznaczoną ilością pamięci dla każdej z nich. Wszystkie omawiane tutaj charakterystyki złożoności pamięciowej zakładają reprezentację zgodną z tą, która obowiązuje wewnątrz maszyny *MIX*.

1. *KEY*

Wskaźnik na pewne miejsce w tablicy *TEXT*. W naszym przykładzie będzie on reprezentowany przez liczbę całkowitą zgodną z numeracją tablicy 2.4.

Wymagana ilość pamięci to co najmniej $\log_2(C + 1)$ bitów, gdzie C to ilość znaków.

2. *LLINK*, *RLINK*

Wskaźniki na inne węzły wewnątrz drzewa – przodka lub dziecka rozważanego węzła. Na rysunku 2.5 oznaczone są liniami, które wychodzą z jednego węzła do drugiego. Linie wychodzące z lewej strony węzła nazywamy *LLINK* a z prawej *RLINK*.

Wymagana ilość pamięci to co najmniej $\log_2(N)$ bitów, gdzie N to ilość węzłów w drzewie *Patricia*.

3. *LTAG*, *RTAG*

Wartość wskazująca na to, czy odpowiednio *LLINK*, *RLINK* wskazują na dziecko czy przodka rozważanego węzła. *LTAG* jest oznaczeniem wartości przypisanej do *LLINK*, a *RTAG* wartości przypisanej do *RLINK*. W przykładzie przedstawionym na rysunku 2.5 przerywane linie z przypisaną do nich jedynką oznaczają, że węzeł na który wskazuje gałąź jest przodkiem. Ciągłe linie z przypisanym do nich zerem oznaczają, że węzeł na który wskazuje gałąź jest dzieckiem.

Wymagana ilość pamięci to dokładnie 1 bit.

4. *SKIP*

Numer mówiący o tym ile należy przeskoczyć pól przy przeszukiwaniu.

Wymagana ilość pamięci jest zależna od maksymalnej ilości pól do przeskoczenia w drzewie. Ściślej określając tę wartość można powiedzieć, że pole to musi być w stanie przechowywać największą liczbę k , taką że wszystkie (co najmniej dwa, różne) klucze z prefiksem σ są takie same przez następne k bitów.

W praktyce – Knuth wyjaśnia – liczba przechowywana przez pole k nie powinna być zbyt duża. Dodaje on jednak, że warto zabezpieczyć przypadek w którym k jest zbyt duże poprzez indyktor błędu.

Węzeł H składa się jedynie z 3 pól: *KEY*, *LLINK* oraz *LTAG*.

W tablicy 2.5 przy wartościach dla średnika znajduje się gwiazdka. Oznacza ona korektę, jakiej musieliśmy dokonać chcąc odwzorować algorytm Knuth’a. Na potrzeby pogodzenia wymogu zmieszczenia się w 5 bitach reprezentacji binarnej oraz implementacji w języku programowania *Java* zmodyfikowaliśmy wartość dla znaku ‘;’. Oryginalnie w książce Knuth’a znakiem końca pliku był znak zapytania ‘?’, który nie występuje w tablicy znaków maszyny *MIX*. Z tego powodu – dla spójności – został on zastąpiony średnikiem, a jego wartość kodowaniu (53 – 110101 – wymagająca 6 bitów do zapisania) została zastąpiona wartością 20 – mieszczącą się na 5 bitach. Wartość 20 w tablicy kodowania znaków maszyny *MIX* przypada znakowi ‘ Φ ’.

Tablica 2.5. Tablica zawartości pliku tekstowego *TEXT* – słów w nim zawartych oraz reprezentacji bitowej w maszynie *MIX*. Na podstawie tego tekstu zostało zbudowane drzewo *Patricia* z rysunku 2.5. Kolumna zawierająca reprezentację dziesiętną jest wynikiem zamiany poszczególnych znaków w słowie na wartości, które Knuth przypisuje znakom w swojej maszynie *MIX* [18]. Np. dla litery T wynosi ona 23, dla H – 8, a dla I – 9. Reprezentacja binarna jest tworzona na podstawie zamiany liczb z reprezentacji dziesiętnej na wartości binarne. Tablica zasięgnięta jest z książki Knuth’a [1].

Słowo	Reprezentacja dziesiętna	Reprezentacja binarna
THIS	23 8 9 22	10111 01000 01001 10110
IS	9 22	01001 10110
THE	23 8 5	10111 01000 00101
HOUSE	8 16 24 22 5	01000 10000 11000 10110 00101
THAT	23 8 1 23	10111 01000 00001 10111
JACK	11 1 3 12	01011 00001 00011 01100
BUILT	2 24 9 13 23	00010 11000 01001 01101 10111
‘ ’ (spacja)	0	00000
;	20*	10100*

Przyjmując za słowo, którego szukamy słowo THE, którego reprezentację bitową można odnaleźć w tabeli 2.3, przeszukiwanie drzewa *Patricia* zaczyna się startem w węźle *H* oraz przejściem lewą, jedyną, istniejącą gałęzią do węzła α i jego analizą.

Przed samą analizą następnego węzła, należy sprawdzić wartość pola *LTAG* lub *RTAG* poprzedniego węzła *H* – zależnie od tego czy wartość bitu nakazała nam wybrać odpowiednio lewą gałąź czy prawą. Skoro wartość pola *LTAG* jest równa 0 przechodzimy do analizy następnego węzła.

Węzeł α nakazuje – przez wartość pola *SKIP* równego 1 – przejście o jeden bit dalej, czyli na bit numer 1. Pierwszy bit słowa THE ma wartość 1, więc należy podążać za prawą gałęzią do węzła γ .

Wartość pola *RTAG* węzła α jest równa 0, nakazując analizę węzła γ .

Pole *SKIP* węzła γ wynosi 11, nakazując analizę bitu dwunastego, równego 0. Lewa gałąź węzła γ prowadzi nas do węzła ϵ .

Wartość pola *LTAG* węzła γ jest równa 0, nakazując analizę węzła ϵ .

Pole *SKIP* węzła ϵ (o wartości 1) nakazuje analizę trzynastego bitu. Bit trzynasty – równy 1 – wskazuje gałąź prawą, a z kolei ona węzeł γ .

Następuje sprawdzenie pola *RTAG* węzła ϵ o wartości 1, który oznacza, że należy sprawdzić, czy tekst, na który wskazuje węzeł (już γ) wewnątrz naszej tablicy *TEXT*, rzeczywiście zaczyna się argumentem wejściowym przeszukiwania (słowem THE).

Taka weryfikacja jest potrzebna ze względu na to, że do tej pory sprawdziliśmy jedynie czy 3 bity są zgodne – wszystkie argumenty o bitach 1XXXX XXXXX X01[...] spełniałyby ten warunek.

Bardziej skomplikowanym przykładem jest poszukiwanie kluczy zaczynających się fragmentem słowa TH, który ma jedynie 10 bitów (10111 01000). Cała procedura przebiega dokładnie tak samo do momentu dotarcia do węzła γ , czyli:

1. Przejście jedyną, lewą, gałęzią węzła H do węzła α .
2. Analiza *LTAG* węzła H :
 $LTAG = 0 \rightarrow$ analiza węzła α .
3. Analiza węzła α :
 $SKIP = 1 \rightarrow$ numer bitu wskazującego gałąź $= 0 + 1 \rightarrow$ wartość bitu wskazującego gałąź $= 1 \rightarrow$ prawa gałąź \rightarrow węzeł γ .
4. Analiza *LTAG* węzła α :
 $LTAG = 0 \rightarrow$ analiza węzła γ .
5. Analiza węzła γ :
 $SKIP = 11 \rightarrow$ numer bitu wskazującego gałąź $= 1 + 11 \rightarrow$ nie istnieje w reprezentacji bitowej argumentu przeszukiwania bit o numerze 12 (ma on tylko 10 bitów) \rightarrow porównanie początkowego fragmentu tekstu, na który skazuje węzeł γ do argumentu:
 - (a) jeżeli argument i fragment tekstu pasują do siebie:
 argument jest początkiem każdego klucza, na który wskazują przerywane gałęzie – czyli te *LLINK*, *RLINK* dla których *LTAG*, *RTAG* są równe 1 – wychodzące z węzła γ lub jego potomków, czyli: THIS, THAT, THE;
 - (b) jeżeli argument i fragment tekstu nie pasują do siebie:
 argument nie jest początkiem żadnego klucza.

2.2.4.1. Algorytm P – Sprawdzanie należenia argumentu do zbioru prefiksów

Dla istniejącej tablicy *TEXT* oraz drzewa, zawierającego co najmniej jeden klucz, który składa się z pól opisanych powyżej, algorytm ten odpowiada na pytanie – czy istnieje klucz rozpoczynający się argumentem wejściowym K .

Dla przypadku kiedy istnieje r kluczy spełniających ten algorytm, gdzie $r \geq 1$, to jest możliwe znalezienie wszystkich tych kluczy w złożoności obliczeniowej równej $O(r)$. Algorytm ten jest wytłumaczony w sekcji 2.2.4.2.

1. Inicjalizacja

Ustaw P – wskaźnik na obecnie analizowany węzeł – na *header* – wejściowy węzeł drzewa, węzeł oznaczony literą H , wskazujący na klucz zaczynający się słowem **THIS** w figurze 2.5 w przykładowym drzewie.

Ustaw j – numer ostatnio analizowanego bitu, gdzie indeksy bitów zaczynają się od 1 – na 0.

Ustaw n – na numer bitów w K (argument wejściowy przeszukiwania). Idź do punktu 2 (ruch w lewo – ponieważ *header* ma tylko lewą gałąź).

2. Ruch w lewo

Ustaw Q – wskaźnik na węzeł poprzednio analizowany – na P .

Ustaw P na $LLINK(Q)$ – pole (wskaźnik) $LLINK$ węzła Q , na który wskazuje na następny na lewo węzeł dziecko.

(a) Jeżeli $LTAG(Q) = 1$ to idź do punktu 6.

(b) Jeżeli $LTAG(Q) = 0$ to idź do punktu 3.

3. Ominięcie bitów

W tym momencie algorytmu jeśli pierwsze j bitów argumentu K jest zgodne z jakimś kluczem, to kluczem pasującym do argumentu wejściowego przeszukiwania jest klucz zaczynający się w miejscu wskazywanym przez $KEY(P)$ – pole węzła na które wskazuje zmienna P , numeru znaku zaczynającego dany klucz.

Ustaw j na $j + SKIP(P)$.

(a) Jeżeli $j > n$ to idź do punktu 6.

(b) Jeżeli $j \leq n$ to idź do punktu 4.

4. Sprawdzenie bitu

W tym momencie algorytmu jeśli pierwsze $j - 1$ bitów argumentu K jest zgodne z jakimś kluczem, to kluczem pasującym do argumentu wejściowego przeszukiwania jest klucz zaczynający się w miejscu wskazywanym przez $KEY(P)$.

(a) Jeżeli bit argumentu K o numerze j jest równy 0 idź do punktu 2.

(b) Jeżeli bit argumentu K o numerze j jest równy 1 idź do punktu 5.

5. Ruch w prawo

Ustaw Q na P .

Ustaw P na $RLINK(Q)$.

(a) Jeżeli $RTAG(Q) = 1$ to idź do punktu 6.

(b) Jeżeli $RTAG(Q) = 0$ to idź do punktu 3.

6. Porównanie klucza do argumentu przeszukiwania

W tym momencie algorytmu jeśli cały argument K jest zgodny z jakimś kluczem, to kluczem pasującym do argumentu wejściowego przeszukiwania jest klucz zaczynający się w miejscu wskazywanym przez $KEY(P)$.

Porównanie argumentu K i klucza rozpoczynamy w miejscu o numerze $KEY(P)$ przez n bitów (ilość bitów w reprezentacji binarnej K).

- (a) Jeżeli każda para n bitów klucza i argumentu jest ze sobą zgodna algorytm kończy się powodzeniem.
- (b) Jeżeli którykolwiek z analizowanych n bitów klucza nie jest taki sam jak ten odpowiadający mu z argumentu K algorytm kończy się niepowodzeniem.

2.2.4.2. Rozszerzenie Algorytmu P – Wyszukiwanie zbioru węzłów akceptujących argument jako prefiks

Algorytm ten znajduje wszystkie węzły w drzewie *Patricia*, dla których argument przeszukiwania jest prefiksem.

Założenia:

Dla przypadku, w którym istnieje r słów spełniających Algorytm P (opisany w sekcji 2.2.4.1), gdzie $r \geq 1$, jest możliwe znalezienie wszystkich tych kluczy w złożoności obliczeniowej równej $O(r)$.

1. Sprawdzanie należenia argumentu do zbioru prefiksów

Wykonaj Algorytm P 2.2.4.1 – Sprawdzanie należenia argumentu do zbioru prefiksów

- (a) Jeżeli $j \leq n$ to istnieje tylko jeden klucz w miejscu wskazywanym przez $KEY(P)$.
- (b) Jeżeli $j > n$ to istnieje więcej niż jeden klucz – idź do punktu 2.

Zbiór tych kluczy można uzyskać przechodząc po pod-drzewie, dla którego korzeniem jest węzeł wskazywany przez zmienną P i znajdując przodków wszystkich węzłów wewnątrz tego pod-drzewa. Każdy z tych przodków wskazuje na miejsce w tablicy *TEXT*, które jest początkiem klucza zgodnego z argumentem przeszukiwania.

2. Inicjalizacja

Stwórz pusty zbiór wskaźników na węzły *SUBS* – zbiór węzłów pod-drzewa, którego korzeniem jest węzeł P .

Stwórz pusty zbiór wskaźników na węzły *KEYS* – zbiór przodków węzłów należących do zbioru *SUBS*, który będzie odpowiedzią na pytanie zadane algorytmowi.

3. Znalezienie węzłów pod-drzewa i ich przodków

Odbywa się przez sprawdzenie $LTAG(P)$ i $RTAG(P)$.

- (a) Jeżeli $TAG(P) = 1$ to odpowiadający mu $LINK(P) - LLINK(P)$ lub $RLINK(P)$ – wskazuje na węzeł będący przodkiem węzła, na który wskazuje zmienna P .
 - i. Jeżeli $LTAG(P) = 1$ to dodaj $LLINK(P)$ do zbioru *KEYS* bez powtórzeń.
 - ii. Jeżeli $RTAG(P) = 1$ to dodaj $RLINK(P)$ do zbioru *KEYS* bez powtórzeń.
- (b) Jeżeli $TAG(P) = 0$ to odpowiadający mu $LINK$ wskazuje na węzeł będący dzieckiem węzła, na który wskazuje zmienna P .

- i. Jeżeli $LTAG(P) = 0$ to sprawdź czy $LLINK(P)$ znajduje się w zbiorze $SUBS$:
 - A. Jeżeli $LLINK(P)$ się już znajduje w zbiorze $SUBS$ to nie rób nic.
 - B. Jeżeli $LLINK(P)$ nie znajduje się jeszcze w zbiorze $SUBS$ to dodaj $LLINK(P)$ do zbioru $SUBS$,
Wywołaj punkt 3. dla przypadku, gdzie za P podstawiony zostaje $LLINK(P)$,
Dodaj zwrócony zbiór wywołania do zbioru obecnego wywołania bez powtórzeń - $SUBS$ do $SUBS$, $KEYS$ do $KEYS$.
- ii. Jeżeli $RTAG(P) = 0$ to sprawdź czy $RLINK(P)$ znajduje się w zbiorze $SUBS$:
 - A. Jeżeli $RLINK(P)$ się już znajduje w zbiorze $SUBS$ to nie rób nic.
 - B. Jeżeli $RLINK(P)$ nie znajduje się jeszcze w zbiorze $SUBS$ to dodaj $RLINK(P)$ do zbioru $SUBS$,
Wywołaj punkt 3. dla przypadku, gdzie za P podstawiony zostaje $RLINK(P)$,
Dodaj zwrócony zbiór wywołania do zbioru obecnego wywołania bez powtórzeń - $SUBS$ do $SUBS$, $KEYS$ do $KEYS$.

Zwróć zbiory $SUBS$ i $KEYS$ idź do punktu 4.

4. Weryfikacja ilości znalezionych kluczy

Jeżeli pod-drzewo, którego korzeniem jest węzeł wskazywany przez P , zawiera $r - 1$ węzłów (licząc również węzeł na który wskazuje zmienna P) to liczebność zbioru kluczy zgodnych z argumentem wejściowym wyszukiwania musi wynosić r .

- (a) Jeżeli liczebność zbioru $SUBS$ jest mniejsza o 1 od liczebności zbioru $KEYS$ to zwróć zbiór $KEYS$ i zakończ algorytm z powodzeniem.
- (b) Jeżeli liczebności zbioru $KEYS$ nie jest większa o 1 od liczebności zbioru $SUBS$ to zakończ algorytm wiadomością o błędzie.

Z punktu 4 Algorytmu P – Wyszukiwania zbioru kluczy 2.2.4.2 można wyodrębnić osobny algorytm obliczający ilość kluczy pasujących do algorytmu wyszukiwania.

2.2.4.3. Rozszerzenie Algorytmu P – Tworzenie drzewa *Patricia*, Wstawianie węzłów i Dodawanie tekstu

Algorytm ten przedstawia w jaki sposób budowane jest drzewo *Patricia*. Również zostanie opisane w jaki sposób można dodawać do tekstu i wstawiać nowe klucze przy założeniu, że nowo wstawiany tekst zawsze kończy się unikalnym przerywnikiem. Efekt unikalnego przerywnika można uzyskać dokładając na koniec symbol końca tekstu, po którym wstawiony zostaje numer seryjny.

Algorytm będzie odwoływał się do *TEXT* jedynie dwa razy.

1. Inicjalizacja

Stwórz pusty węzeł *HEAD*.

Ustaw *KEY(HEAD)* na początek pierwszego słowa tekstu (zwykle na numer miejsca równy 1).

Ustaw *LLINK(HEAD)* na *HEAD*, a *LTAG(HEAD)* na 1.

SKIP(HEAD), *RLINK(HEAD)* i *RTAG* mają pozostać nieustawione.

Ustaw *K* na klucz, który chcielibyśmy dodać z tablicy *TEXT* (będzie to pierwsze odwołanie do tej tablicy). W tym wypadku kluczem jest cała zawartość pliku, od pozycji 1 do końca pliku.

2. Sprawdzanie nowego klucza

Wykonaj Algorytm P 2.2.4.1 (punkt 6. wykonuje drugie odwołanie do tablicy *TEXT*).

(a) Jeżeli wykona się z niepowodzeniem idź do punktu 3.

(b) Jeżeli wykona się z powodzeniem przerwij program z powodu błędu (ponieważ żaden klucz nie może być prefiksem innego klucza).

3. Znajdywanie maksymalnej ilości zgodnych bitów

W tym miejscu algorytmu klucz znaleziony w kroku szóstym Algorytmu P 2.2.4.1 jest zgodny z argumentem przeszukiwania *K* przez pierwsze *g* bitów, ale różni się w pozycji *g* + 1.

K ma w pozycji *g* + 1 bit o wartości *b*, a klucz ma wartość odwrotną ($1 - b$).

Przeszukanie w Algorytmie P 2.2.4.1 mogło spowodować, że *j* jest zdecydowanie większe niż *g*, ale nie wpływa to na fakt, że opisana tutaj procedura znajdzie najdłuższy zgodny bitami fragment między *K*, a dowolnym już istniejącym kluczem w drzewie (*prefix*).

Dzięki temu wiadomo, że wszystkie klucze, które zaczynają się takimi samymi *g* bitami jak argument *K*, posiadają na pozycji *g* + 1 bit odwrotny ($1 - b$) do bitu *K*(*b*). Dlatego do algorytmu 2.2.4.1 należy dodać wewnątrz punktu 6. (Porównanie klucza do argumentu przeszukiwania) zapisywane największego indeksu zgodnych bitów w danym uruchomieniu algorytmu P do zmiennej *g*. Czyli dodatkowo należy dodać w punkcie 1. (Inicjalizacja) – również tego samego algorytmu – resetowanie zmiennej przechowującej największy indeks zgodnych bitów, na przykład na wartość 0.

Pobierz wartość zmiennej *g* z uruchomienia algorytmu P z punktu 2. (Sprawdzanie nowego klucza) obecnie opisywanego rozszerzenia algorytmu P.

Idź do punktu 4.

4. Znajdywanie węzła o maksymalnej ilości zgodnych bitów

Powtórz Algorytm P 2.2.4.1 przyjmując za nowy argument przeszukiwania *K*, pierwsze *g* bitów argumentu *K* poprzedniego wywołania Algorytmu P – czyli:

- (a) Ustaw *K* na *g* bitów *K*.
- (b) Wywołaj Algorytm P(*K*).

- i. Jeżeli Algorytm P wykona się z powodzeniem to:

Tym razem Algorytm P nie wchodzi do punktu szóstego, więc nie ma trzeciego odwołania do *TEXT*.

Idź do punktu 5 (obecnie omawianego Rozszerzenia Algorytmu P – Tworzenie drzewa *Patricia* 2.2.3).

- ii. Jeżeli Algorytm P wykona się z niepowodzeniem to:

Przerwij program z powodu błędu (taki przypadek nie powinien się pojawić, skoro w poprzednim wywołaniu Algorytmu P dopasowaliśmy te pierwsze g bitów oryginalnego argumentu K).

5. Wstawianie nowego węzła na miejsce wcześniej znalezionego

Stwórz nowy, pusty węzeł R .

Ustaw $KEY(R)$ na numer pozycji startowej nowego klucza w tablicy *TEXT*.

Ustaw nowy węzeł R jako dziecko poprzedniego węzła Q zamiast węzła P , zachowując dotyczące go dane – czyli:

- (a) Jeżeli $LLINK(Q) = P$ to:

Ustaw t – wartość lewego lub prawego $TAG(Q)$ przed wstawieniem nowego węzła – na $LTAG(Q)$,

Ustaw $LLINK(Q)$ na R ,

Ustaw $LTAG(Q)$ na 0.

- (b) Jeżeli $RLINK(Q) = P$ to:

Ustaw t na $RTAG(Q)$,

Ustaw $RLINK(Q)$ na R ,

Ustaw $RTAG(Q)$ na 0.

Idź do punktu 6.

6. Konfiguracja nowego węzła i wstawianie wcześniej wyciętego węzła

Ustaw relacje (dziecko lub przodek) nowego węzła R i węzła P taką samą jak była wcześniej relacja między P i węzła Q , dodatkowo ustaw jako przodka R sam węzeł R – czyli:

- (a) Jeżeli $b = 0$ to:

Ustaw $LTAG(R)$ na 1,

Ustaw $LLINK(R)$ na R ,

Ustaw $RTAG(R)$ na t ,

Ustaw $RLINK(R)$ na P .

- (b) Jeżeli $b = 1$ to:

Ustaw $RTAG(R)$ na 1,

Ustaw $RLINK(R)$ na R ,

Ustaw $LTAG(R)$ na t ,

Ustaw $LLINK(R)$ na P .

Ustaw pole *SKIP* nowego węzła R :

- (a) Jeżeli $t = 1$ to:
Ustaw $SKIP(R)$ na $1 + g - j$.
- (b) Jeżeli $t = 0$ to:
Ustaw $SKIP(R)$ na $1 + g - j + SKIP(P)$,
Ustaw $SKIP(P)$ na $j - g - 1$.

Zakończ algorytm z powodzeniem.

2.2.4.4. Przeznaczenie

Drzewo *Patricia* nie przechowuje kluczy w węzłach (pamięci RAM), tylko w plikach, więc jest ona szczególnie przystosowana dla długich, różnej długości kluczy (tytuły, frazy).

Nie tworzy też zbędnych węzłów (z pojedynczymi dziećmi), więc jest przystosowane dla zbioru zróżnicowanych kluczy.

Dopasowuje argument wyszukiwania do początkowego fragmentu, któregoś z kluczy, więc nie dopasowuje dokładnie argumentu całego klucza.

Okazuje się, że można zmodyfikować algorytm Knuth'a tak, aby kluczami były pojedyncze słowa – to znaczy kluczem był fragment pliku zaczynający się pozycją startową i kończący w indeksie poprzedzającym pozycję startową następnego klucza. Osiągamy to w swojej implementacji poprzez dodanie unikalnego, zarezerwowanego znaku, rozdzielającego kolejne słowa w zawartości pliku, określanego później jako znak końca klucza lub *EOK* (ang. *End Of Key character*). Dzięki zawarciu takiego znaku w reprezentacji bitowej podczas analizy węzłów drzewa *Patricia* jesteśmy w poprawny sposób różnić słowa przedstawione w poniższym przykładzie.

Analizując przykład dwóch kluczy mających postać słów 'THE' oraz 'TH', gdzie znakiem *EOK* jest ' ' (spacja), jesteśmy w stanie zauważyć, że gdybyśmy porównywali reprezentację bitową pierwszych 2 znaków (nie biorąc pod uwagę znaku *EOK*), nie byłibyśmy w stanie ich rozróżnić. Natomiast dodając znak zarezerwowany, który nie może wystąpić w środku słowa, który chcielibyśmy, że by stał się kluczem drzewa, widzimy różnicę między tymi dwoma słowami, analizując bity reprezentujące trzeci znak każdego z tych słów.

Konsekwencją braku możliwości rozróżnienia tych dwóch słów byłby brak możliwości wstawienia, któregoś z nich jako klucza do drzewa zgodnie z algorytmem opisanym przez Knuth'a.

2.2.4.5. Złożoność czasowa i pamięciowa

Knuth w swojej książce [1] poświęca prawie 8 stron analizie złożoności omówionych algorytmów. Na szczęście w podsumowaniu tego podrozdziału można znaleźć kluczowe informacje.

Informuje on nas, że przy N losowych kluczach w drzewach M -ary ilość węzłów do ich przechowywania wynosi $\frac{N}{s \cdot \log_2(M)}$, gdzie M to liczebność słownika, a s to ilość „pod-plików” kluczy*.

Zakładając, że dla każdego węzła drzewa *Patricia* potrzebujemy następującą ilość pamięci:

1. *KEY* – $\log_2(C + 1)$ bitów, C – ilość znaków w tekście
2. *LLINK*, *RLINK* – $\log_2(N)$ bitów, N – ilość węzłów w drzewie
3. *LTAG*, *RTAG* – 1 bit (0 lub 1)
4. *SKIP* – nieduża wartość całkowita, dodatnia

to wymagania pamięciowe nie są zbyt wygórowane.

Dodatkowo dzięki podsumowaniu Knuth’a wiemy, że ilość znaków, które muszą zostać przeanalizowane podczas przeszukiwania z losowymi danymi* wynosi w przybliżeniu $\log_M(N)$ dla wszystkich omówionych metod. Gdy $M = 2$ dalsze przybliżenia tego równania dla poszczególnych metod pokazane są w znajdującej się poniżej tablicy 2.6.

Tablica 2.6. Te przybliżenia mogą być wyrażone przy użyciu fundamentalnych matematycznych stałych. Na przykład 0.31875 to tak na prawdę $\frac{\ln(\pi) - \gamma}{\ln(2) - \frac{1}{2}}$.

	Udane przeszukiwanie	Nieudane przeszukiwanie
Przeszukiwanie <i>Trie</i>	$\log_2(N) + 1.33275$	$\log_2(N) - 0.10995$
Drzewo przeszukiwania cyfrowego	$\log_2(N) - 1.71665$	$\log_2(N) - 0.27395$
Drzewo <i>Patricia</i>	$\log_2(N) + 0.33275$	$\log_2(N) - 0.31875$

Metody przeszukiwania oparte o analizę reprezentacji binarnej są nieczułe na kolejność wstawiania kluczy do struktury (poza Algorytmem D 2.2.3.1), są one jednak bardzo czułe na rozkład liczb (kodów znaków).

2.3. Koniunkcyjna postać normalna

Kanoniczna koniunkcyjna postać normalna [20, 21] (ang. *(canonical) conjunctive normal form* lub *CNF* lub *CCNF*) to w algebrze Bool’a (ang. *Boolean logic* lub *Boolean algebra*) rodzaj kanonicznej postaci normalnej [22]. Jest to postać formuły logicznej, w której formuła przyjmuje postać koniunkcji klauzul. Klauzula [23, 24] jest alternatywą literałów, gdzie literał może być formułą atomową [25] lub zaprzeczeniem atomu. Innymi słowy koniunkcyjna postać normalna to dysjunkcja alternatyw literałów.

Przykładem formuły logicznej $(v_1 \wedge v_2) \vee v_3$ w postaci *CNF* jest:

$$(v_1 \vee v_3) \wedge (v_2 \vee v_3) \quad (2.5)$$

*To przybliżenie jest prawdziwe dla dużych N , małych s i małych M .

*Losowe oznaczają tutaj, że liczby odpowiadające kodowi znaków są rozłożone w sposób równomierny.

Przeciwną formą zapisu formuły logicznej do formy *CNF* jest dysjunkcyjna postać normalna (ang. *(canonical) disjunctive normal form* lub *DNF* lub *CDNF*) [26]. *DNF* jest to alternatywa koniunkcji literałów.

Przykładem formuły logicznej $(v_1 \vee v_2) \wedge v_3$ w postaci *DNF* jest:

$$(v_1 \wedge v_3) \vee (v_2 \wedge v_3) \quad (2.6)$$

2.3.1. Format plików *DIMACS CNF*

Format *DIMACS CNF* [27, 28, 29] został stworzony przez *DIMACS* (ang. *Center for Discrete Mathematics and Theoretical Computer Science*). Pozwala on na przetrzymywanie wyrażeń logicznych w postaci tekstowej, gdzie formuły logiczne zapisywane są w kanonicznej koniunkcyjnej postaci normalnej. Pliki w tym formacie posiadają rozszerzenie *.cnf* a ich zawartość jest zapisana w formacie *ASCII*. Jej przykładową formę przedstawiam w kodzie źródłowym 1.

Kod źródłowy 1. Przykładowa zawartość pliku *DIMACS CNF*.

```

1      c This is example template for file in DIMACS CNF file. This is a first
      comment line.
2      c This is a second comment line.
3      c The line after the next one is problem line.
4      c Problem line takes the format of "p FORMAT VARIABLES_AMOUNT
      CLAUSES_AMOUNT"
5      p cnf 6 3
6      1 2 3 0
7      1 2 3 4 0
8      1 2 3 4 5 6 0

```

Plik może zaczynać się liniami komentarzy, a każda z nich zaczyna się znakiem małej litery ‘c’. Linie komentarzy mogą się przewijać również w późniejszych fragmentach pliku.

Na początku pliku, po liniach komentarzy, znajdują się linie problemu, które zaczynają się znakiem małej litery ‘p’. W linii problemu pierwszą informacją jest typ problemu (zwykle jest nim ‘cnf’). Drugą informacją jest ilość literałów, a trzecią ilość klauzul. Każda z informacji jest rozdzielona znakiem spacji (‘ ‘).

Reszta pliku zawiera linie klauzul. Zwykle klauzula zapisana jest na pojedynczej linii, jednak może się ona rozciągać na więcej niż jedną linię oraz dwie klauzule mogą się znajdować w jednej linii. Każda klauzula składa się z listy numerów przypisanych literałom oddzielonych spacją. Literał będący zaprzeczeniem danego atomu przyjmuje postać numeru atomu z minusem na początku. Żadnemu atomowi nie może być przypisany numer 0, gdyż jest on zarezerwowany jako unikalny znak oznaczający koniec klauzuli. Numery przypisane atomom nie muszą zaczynać się od 1, ani nie muszą być przypisywane w kolejności rosnącej (choć zwykle są). Plik kończy się po zdefiniowanej ostatniej klauzuli. Ostatnia klauzula nie musi być zakończona znakiem 0.

3. Implementacja drzewa *Patricia*

Zgodnie z celem jaki postawiliśmy sobie w tej pracy, poddaliśmy wnikliwej analizie wszystkie algorytmy dotyczące drzew *Trie* opisane przez Knuth’a w jego książce w rozdziale „6.3. Wyszukiwanie Cyfrowe” (ang. *Digital Searching*). Aby dobrze zrozumieć zasady ich działania i dokładnie opisać je w swojej pracy postanowiliśmy przetestować swoje podejrzenia, tworząc pierwszą iterację implementacji drzewa *Patricia*.

Pierwszą wersję implementacji wykonaliśmy według algorytmu opisanego w sekcji dotyczącej drzewa *Patricia* 2.2.4. Celem było stworzenie implementacji, której rezultatem będzie taka sama struktura, jaką Knuth opisuje w swojej książce. Aby to osiągnąć, stworzyliśmy klasę symulującą reprezentację binarną maszyny *MIX* oraz oddzielną klasę symulującą kodowanie znaków odpowiadającą tej z maszyny *MIX*. Napisaliśmy dla niej serię testów, która pozwalała – metodą prób i błędów – modyfikować implementację oraz dociekać prawidłowego toku rozwoju implementacji.

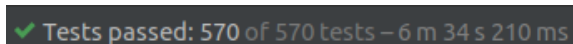
Wprowadzając zmiany do implementacji, testując poprawność działania algorytmów i rozjaśniając nieścisłości lub trudne do zrozumienia koncepcje opisane w książce źródłowej, zdobyliśmy wiedzę pozwalającą na szczegółowe opisanie tych algorytmów w niniejszej pracy. Powtarzając wielokrotnie ten proces udało się nam osiągnąć implementację zgodną z opisem Knuth’a oraz – wprowadzając pewne modyfikacje – naszą autorską wersję.

To wszystko nie byłoby możliwe bez stworzenia i zastosowania ogromnej ilości testów jednostkowych. Pod koniec pracy nad programem zbiór testujący zawierał 570 złożonych testów, gdzie każdy z nich składał się z wielu porównań (ang. *assert(s)*).

Rysunek 3.1 przedstawia wynik oraz czas testowania programu.

Długi czas wykonywania testów wynika z przetwarzania dwóch, dużych plików. Jeden z nich ma rozmiar 8,4MB, a drugi 1,5MB. Obydwa zawierają dane wejściowe w formacie *DIMACS CNF*.

Rysunek 3.1. Przykładowy wynik wykonania wszystkich testów zawartych w programie projektowym. Obrazek przedstawia fragment konsoli środowiska oprogramowania o nazwie *IntelliJ IDEA*, w którym pisaliśmy oraz testowaliśmy program projektowy.

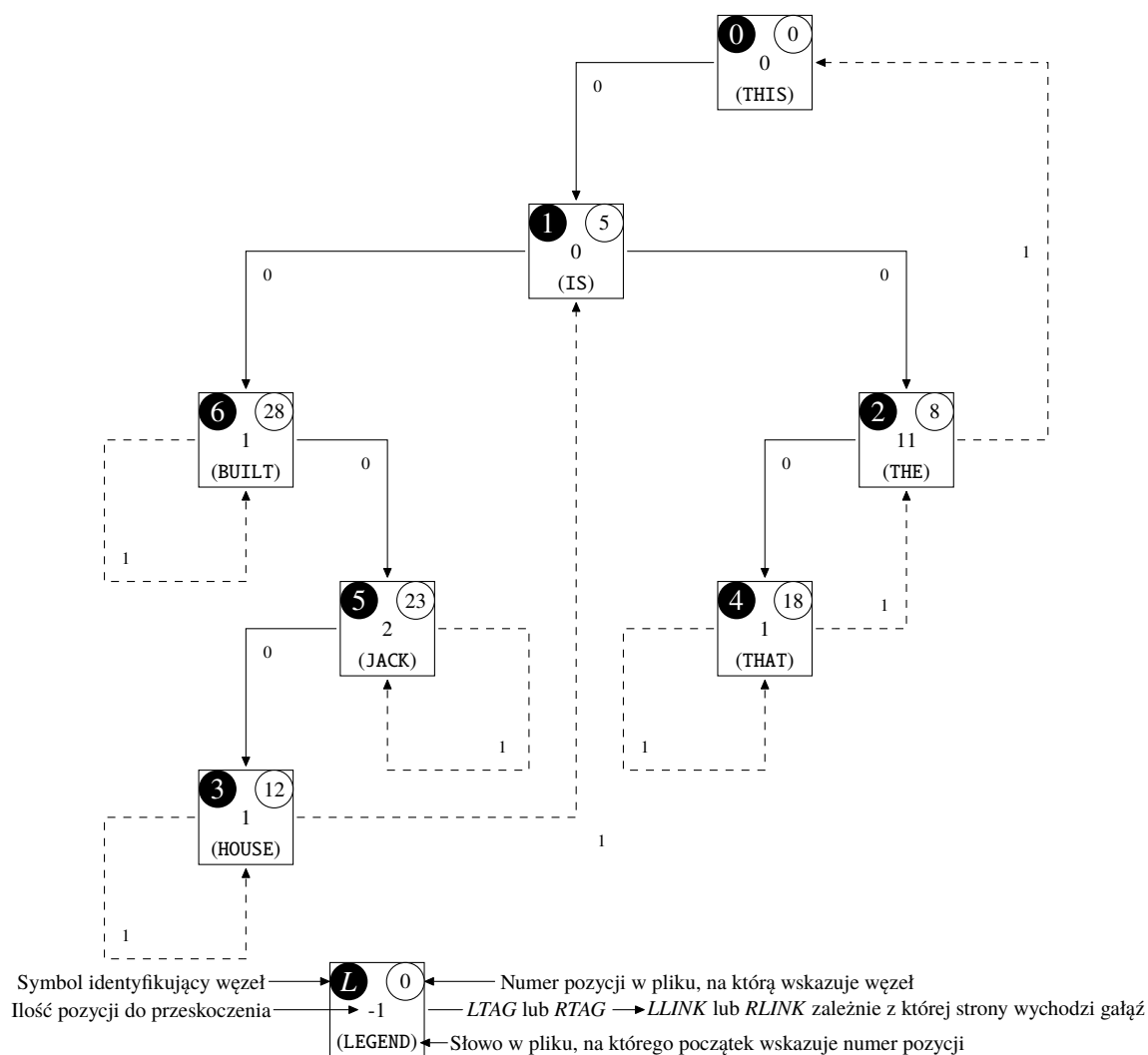


✓ Tests passed: 570 of 570 tests - 6 m 34 s 210 ms

3.1. Wyzwania implementowania w języku programowania Java

Należy zwrócić szczególną uwagę na fakt, iż Knuth w swoim tłumaczeniu zakłada iteracje po indeksach zaczynających się od 1, podczas gdy *Java* naturalnie przyjmuje indeksowanie od 0. W związku z tym należy dostosować algorytm tak, by brał to pod uwagę. W efekcie nasza implementacja drzewa *Patricia* przyjmuje postać przedstawioną na rysunku 3.2.

Rysunek 3.2. Schemat drzewa *Patricia* stworzonego przez implementację w języku *Java* w zgodności z algorytmem opisanym przez Knuth’a w „Sztuka programowania tom 3” [1]. Obrazuje przesunięcie pozycji klucza w pliku spowodowaną różnicą początkowego indeksu. Różnice można zauważyć porównując tę figurę względem figury 2.5.



Kolejną przeszkodą stojącą na drodze implementacji dokładnie takiego samego przykładu jak ten podany w książce „Sztuka programowania tom 3” [1] jest używana przez Knuth’a reprezentacja binarna, której bajty mają długość 5 bitów.

Na dodatek, Knuth dla swojej maszyny *MIX* zakłada kodowanie znaków, które znacząco różni się od tego występującego w języku *Java*. Kodowanie, którego używa pokazane jest w tablicy 3.1. Oprócz różnicy w tym, jaki kod znaku (liczba całkowita) przypada znakowi (na przykład literze) w maszynie *MIX* pojawiają się znaki niewystępujące w tablicy *ASCII*, takie jak θ , Φ czy Π . Mówiąc w tym miejscu o kodzie znaku, mamy na myśli wartość całkowitą otrzymywaną po rzutowaniu zmiennej typu *char* (znaku) na *int*. Wiąże się to z brakiem możliwości ograniczenia z góry zawartości plików do znaków kodowanych na pojedynczym bajcie przy kodowaniu *UTF-8*.

Tablica 3.1. Tablica kodowania znaków maszyny *MIX* zasięgnięta z [https://esolangs.org/wiki/MIX_\(Knuth\)](https://esolangs.org/wiki/MIX_(Knuth)) [18]. Znak posiadający kod 0 to spacja, z tego powodu jest niewidoczny.

Kod znaku Znak	0	1	2	3	4	5	6	7	8	9
		A	B	C	D	E	F	G	H	I
Kod znaku Znak	10	11	12	13	14	15	16	17	18	19
	θ	J	K	L	M	N	O	P	Q	R
Kod znaku Znak	20	21	22	23	24	25	26	27	28	29
	Φ	Π	S	T	U	V	W	X	Y	Z
Kod znaku Znak	30	31	32	33	34	35	36	37	38	39
	0	1	2	3	4	5	6	7	8	9
Kod znaku Znak	40	41	42	43	44	45	46	47	48	49
	.	,	()	+	-	*	/	=	\$
Kod znaku Znak	50	51	52	53	54	55				
	<	>	@	;	:	'				

Ma to znaczenie, gdyż w swojej implementacji chcieliśmy wykorzystać klasę *RandomAccessFile*, która odczytuje plik niekoniecznie od jego początku, krok po kroku, przechodząc do fragmentu w którym chcemy wykonać daną operację. Ma ona możliwość odczytu pliku w podawanym poprzez parametr miejscu – danym indeksem bajtu zaczynającym się od 0.

Występowanie znaków z poza tablicy *ASCII* w tablicy znaków maszyny *MIX* oznacza, że nie możemy przyjąć założenia ułatwiającego – sprawiającego, że indeks bajtu zawsze jest równy indeksowi znaku. Oprócz śledzenia tej różnicy musimy obsłużyć odczytywanie odpowiedniej ilości bajtów (1, 2, 3 lub 4 bajty) na podstawie pierwszego bajtu znaku oraz dekodowanie ich tworząc obiekt klasy *String*, która z kolei jest przechowywana w pamięci wirtualnej maszyny języka *Java* zgodnie z tablicą kodowania znaków *UTF-16*. Dodatkowo Knuth wykorzystuje w swoim oryginalnym przykładzie znak zapytania (?), który nie pojawia się w tablicy znaków maszyny *MIX*. Na szczęście pełni on funkcję unikatowego znaku końca klucza, co oznacza, że można zastąpić go wolnym znakiem obecnym w tablicy znaków maszyny *MIX*. Nadpisując metody *toString()* odpowiednich klas możemy uzyskać reprezentację tekstową drzewa, która może przyjmować postać przedstawioną na następnej stronie, w kodzie źródłowym 2.

Kod źródłowy 2. Przykładowa tekstowa reprezentacja obiektu klasy implementującej drzewo *Patricia*. Reprezentuje ona to samo drzewo co figura 3.2 tylko w postaci tekstowej. Wynik metody nadpisanej *toString()*.

```

1 PatriciaTree{
2     // [...]
3     header = PatriciaNode{
4         id = 0,
5         key = 0,
6         skip = 0,
7         isLeftAncestor = false ,
8         isRightAncestor = false ,
9         leftLink = PatriciaNode{
10            id = 1,
11            key = 7,
12            skip = 1,
13            isLeftAncestor = false ,
14            isRightAncestor = false ,
15            leftLink = PatriciaNode{
16                id = 6,
17                key = 42,
18                skip = 1,
19                isLeftAncestor = true ,
20                isRightAncestor = false ,
21                leftLink.id = 6,
22                rightLink = PatriciaNode{
23                    id = 5,
24                    key = 35,
25                    skip = 2,
26                    isLeftAncestor = false ,
27                    isRightAncestor = true ,
28                    leftLink = PatriciaNode{
29                        id = 3,
30                        key = 20,
31                        skip = 1,
32                        isLeftAncestor = true ,
33                        isRightAncestor = true ,
34                        leftLink.id = 3,
35                        rightLink.id = 1
36                    },
37                    rightLink.id = 5
38                },
39            },
40            rightLink = PatriciaNode{
41                id = 2,
42                key = 14,
43                skip = 13,
44                isLeftAncestor = false ,
45                isRightAncestor = true ,
46                leftLink = PatriciaNode{
47                    id = 4,
48                    key = 28,
49                    skip = 1,
50                    isLeftAncestor = true ,
51                    isRightAncestor = true ,
52                    leftLink.id = 4,
53                    rightLink.id = 2
54                },
55                rightLink.id = 0
56            },
57            rightLink = null
58        }
59    }
60    // [...]
61 }
```

3.2. Implementacja kodowania znaków oraz reprezentacji bitowej maszyny MIX

Implementując pierwszą wersję klasy drzewa *Patricia* chcieliśmy osiągnąć wyniki jak najbardziej zbliżone do podanych przez Knuth'a przykładów.

W tym celu stworzyliśmy klasę *MixByte* symulującą reprezentację binarną maszyny *MIX*, a dokładniej jej pojedynczego bajtu. Klasa *MixByte* w swoim konstruktorze, jako parametr, przyjmuje pożądaną długość bajtu. Utworzyliśmy również klasę *MixEncoding* symulującą kodowanie znaków maszyny *MIX* według tablicy 3.1. Zmodyfikowaliśmy ją jednak tak, aby zagwarantować, że podawane przez parametry konstruktora znaki końca pliku i końca klucza zawsze będą zawarte w tablicy znaków symulowanej maszyny. Znak końca pliku za każdym razem jest umieszczany w miejsce znaku '1' (kod znaku 31), a znak końca klucza zawsze w miejsce znaku '0' (kod znaku 30) zgodnie z tabelą przedstawioną w tablicy 3.2. Oznacza to, że gwarancja istnienia tych znaków w tablicy symulowanej maszyny dotyczy tylko maszyny, której bajt ma długość co najmniej 5 bitów. Znaczenie tych znaków zostaje wytłumaczone w następnych rozdziałach 3.3 oraz 3.4.

Tablica 3.2. Zmodyfikowana tablica kodowania znaków maszyny *MIX* tak, aby gwarantować istnienie sparametryzowanych znaków końca pliku i końca klucza, w symulowanej maszynie, której długość bajtu wynosi co najmniej 5 bitów. Znak oznaczony przez *EOF* to znak końca pliku (ang. *End Of File character*), a znak oznaczony przez *EOK* to znak końca klucza (ang. *End Of Key character*). Tablica zasięgnięta z [https://esolangs.org/wiki/MIX_\(Knuth\)](https://esolangs.org/wiki/MIX_(Knuth)) [18]. Znak posiadający kod 0 to spacja, z tego powodu jest niewidoczny.

Kod znaku Znak	0	1	2	3	4	5	6	7	8	9
		A	B	C	D	E	F	G	H	I
Kod znaku Znak	10	11	12	13	14	15	16	17	18	19
	θ	J	K	L	M	N	O	P	Q	R
Kod znaku Znak	20	21	22	23	24	25	26	27	28	29
	Φ	Π	S	T	U	V	W	X	Y	Z
Kod znaku Znak	30	31	32	33	34	35	36	37	38	39
	EOK	EOF	2	3	4	5	6	7	8	9
Kod znaku Znak	40	41	42	43	44	45	46	47	48	49
	.	,	()	+	-	*	/	=	\$
Kod znaku Znak	50	51	52	53	54	55				
	<	>	@	;	:	'				

Scalając te dwie klasy w postać klasy *MixMachine* osiągnęliśmy rezultat prawie identyczny do przykładu przedstawionego przez Knuth'a w jego książce. Jediną różnicą jest przesunięcie o 1 wartości pola *KEY*. Wynika ono z innego indeksowania pozycji znaku w pliku, które w naszej implementacji wynosi 0, a nie 1, jak w oryginale. Różnicę można zauważyć porównując rysunek 2.5 (przykład Knuth'a) oraz rysunek 3.2 (nasza implementacja).

3.3. Implementacja drzewa *Patricia* – w zgodności z wymaganiami Knuth’a

Algorytmy wyszukiwania listy węzłów akceptujących argument wyszukiwania jako prefiks (opisany w niniejszej pracy w rozdziale 2.2.4.2) oraz wstawiania nowych węzłów do drzewa (opisany w niniejszej pracy w rozdziale 2.2.4.3) przedstawione w książce Knuth’a [1] były dla nas z początku niejasnymi fragmentami. Dzięki zabiegom opisanym wcześniej mogliśmy pozwolić sobie na przetestowanie naszych domysłów. Takie podejście pozwoliło nam zdecydowanie wcześniej przystąpić do implementacji, rozwiązać wątpliwości, odpowiedzieć na pojawiające się w trakcie pytania oraz opisać w sposób bardziej szczegółowy wszystko, czego dowiedzieliśmy się w tym procesie.

Algorytm P 2.2.4.1 (sprawdzający czy słowo dane jako parametr jest prefiksem, któregoś z kluczy w drzewie) jest w zaimplementowany w postaci metody *isContainingPrefix(String searchWord)* klasy *PatriciaTree*.

findNodesMatchingPrefix(String searchWord) tej samej klasy jest metodą implementującą algorytm rozszerzający algorytm P 2.2.4.2 (wyszukujący listę węzłów zawierających klucze akceptujące słowo dane parametrem jako prefiks).

Metoda *insertNextKeyIntoTree()* również należąca do klasy *PatriciaTree* jest metodą wstawiającą następny węzeł zawarty do struktury drzewa. Działa ona zgodnie z algorytmem opisanym w sekcji 2.2.4.3. Oprócz zasady działania opisanej w tej sekcji kluczowymi metodami wykorzystywanymi w naszej implementacji są metody *findNextWordStartIndex(int latestInsertedNodeKeyPosition)* oraz *getWordStringFromFileStartingAtPosition(int newKeyStartIndex)* klasy *FileOps* (a dokładniej – jej pola klasy rozszerzającej klasę *FileOpsStrategy*).

Metoda *findNextWordStartIndex(int latestInsertedNodeKeyPosition)* zwraca pozycję (indeks bajtu w pliku) początku następnego do wstawienia klucza – na podstawie argumentu będącego pozycją początku ostatnio wstawionego klucza.

Metoda *getWordStringFromFileStartingAtPosition(int newKeyStartIndex)* zwraca klucz (w postaci obiektu klasy *String*) – na podstawie pozycji w której on się zaczyna w pliku.

insertAllKeysIntoTree() jest metodą, która jest naturalnym rozszerzeniem operacji wstawienia pojedynczego klucza do drzewa *Patricia*. Jej działanie jest zaskakująco proste. Wykorzystuje metodę *insertNextKeyIntoTree()* w pętli bez warunku końca. Zatrzymanie pętli odbywa się poprzez wyłapanie (ang. *catch*) wyjątku *NextWordStartIndexNotFoundException* wyrzucanego (ang. *throw*) przez metodę odpowiedzialną za znajdowanie pozycji startowej następnego klucza do dodania do drzewa.

Tak jak wspomnieliśmy w poprzednim podrozdziale – oprócz tego udało się nam osiągnąć prawie identyczne wyniki jak te podane przy opisach Knuth’a. Dzięki temu mogliśmy przetestować możliwości implementacji naszych klas, ich parametryzowania oraz ograniczenia zaimplementowanych algorytmów. Natchnęło nas to do podjęcia próby modyfikacji tych algorytmów tak, aby klucze w drzewie *Patricia* przypominały bardziej klucze opisywane w innych algorytmach (dotyczących drzew *Trie*). Chcieliśmy, aby klucze mogły być pojedynczymi słowami, a nie były ograniczone do bycia fragmentem jednego zdania.

3.4. Implementacja drzewa *Patricia* – autorski algorytm

Pod koniec procesu opisanego w poprzednim podrozdziale 3.3 zdaliśmy sobie sprawę z faktu iż, algorytmy dotyczące drzewa *Patricia* opisane przez Knuth’a nie mają możliwości wykorzystania w praktyce różnicy między prefiksem, a kluczem dla opisanych przez niego operacji. Jest to konsekwencją unikalnego sposobu definiowania klucza w porównaniu do pozostałych algorytmów opisanych w tym rozdziale.

Knuth w swojej książce w kontekście drzewa *Patricia* definiuje klucz jako fragment pliku zaczynający się w pozycji startowej x klucza, aż do końca pliku, jednocześnie zakładając, że plik – będący zbiorem kluczy – jest bardzo długi. Prawdą jest, że dla bardzo specyficznego przypadku, kiedy każdy z kluczy jest prefiksem drugiego (nie licząc unikalnego znaku końca pliku) algorytm Knuth’a jest optymalny. Chcieliśmy jednak, aby nasza implementacja była przystosowana do szerszego zakresu problemów. W związku z tym w naszej modyfikacji algorytmu skupiliśmy się na tym, aby klucze mogły być pojedynczymi słowami. W efekcie udało się nam zmodyfikować algorytm Knuth’a tak, aby kluczami mogły być jednocześnie słowa i frazy. Różnica w sposobie definiowania klucza przedstawiona jest na rysunku 3.3 znajdującym się na następnej stronie.

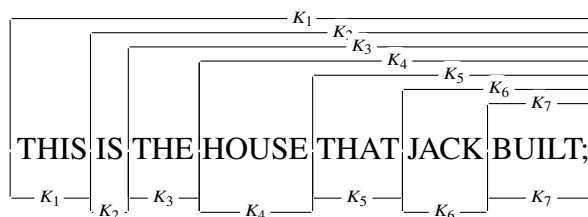
Motywacją do podjęcia się tej modyfikacji było to, że dla przypadku, który nie jest tym szczególnym, lecz tym bardziej ogólnym, w którym chcielibyśmy, aby dało się w drzewie przetrzymywać zbiór kluczy, w którym niekoniecznie każdy jest prefiksem innego (nie licząc znaku końca pliku) – algorytm Knuth’a jest naszym zdaniem nieoptymalny.

Nie widzimy powodu, dla którego sprawdzanie czy dany klucz należy do zbioru kluczy akceptowanych przez drzewo *Patricia*, czy wyszukiwanie węzła zawierającego klucz jest praktyczne w przypadku, gdy klucz jest długim ciągiem znaków (należących nie tylko do klucza, który nas na prawdę interesuje).

Dodatkową wadą szkicu implementacji przedstawionej przez Knuth’a dla sytuacji kiedy klucze są oddzielnymi ciągami znaków i klucz, o który chcemy zapytać drzewo *Patricia* nie jest ostatnim w pliku jest fakt, że musimy znać nie tylko wszystkie klucze występujące po tym kluczu w pliku, ale również ich kolejność występowania.

Rysunek 3.3. Rysunek przedstawia różnicę w sposobie definiowania klucza w implementacjach definiujących klucz. Obie klasy dziedziczą po abstrakcyjnej klasie *FileOpsStrategy*. Wybór wykorzystywanej implementacji odbywa się poprzez przekazanie parametru klasy *enum WordStrategy* do konstruktora klasy *FileOps*. Jest to przykład wykorzystania wzorca projektowego strategia (ang. *strategy*). Oznaczenia i linie znajdujące się nad zdaniem dotyczą założeń Knuth’a a te, znajdujące się na samym dole – naszego założenia.

Implementacja klucza zgodnie z wymaganiami algorytmów Knuth’a
 Obiekt klasy: *new FileOps(..., WordStrategy.START_POSITION_TO_EOF, ...)*;
 Pole klasy: *WordStartPositionToEOFStrategy extends FileOpsStrategy*



Implementacja autorska klucza postaci pojedynczego słowa
 Obiekt klasy: *new FileOps(..., WordStrategy.SINGLE, ...)*;
 Pole klasy: *WordSingleStrategy extends FileOpsStrategy*

Z tego powodu postanowiliśmy stworzyć oddzielny wariant drzewa *Patricia*, który definiowałby klucz jako fragment pliku, który zaczynałby się w pozycji startowej *x*, ale kończył po napotkaniu znaku końca klucza, który nazwaliśmy *End Of Key character*. Dzięki temu pojedyncze klucze byłyby zdecydowanie krótsze i mogłyby być wykorzystywane w praktyce jako argumenty operacji przeszukiwania drzewa.

Omawianą wariację drzewa *Patricia* postanowiliśmy zaimplementować w postaci wzorca projektowego strategia (ang. *strategy*). Wybór strategii odbywa się poprzez przekazanie jako parametru obiektu klasy *WordStrategy* typu *enum* do konstruktora klasy *FileOps*. Obiekt klasy *enum WordStrategy* przyjmuje 2 wartości *WordStrategy.SINGLE* oraz *WordStrategy.START_POSITION_TO_EOF*. Dzięki temu klasa *FileOps* wie, której klasy konstruktor wywołać - odpowiednio: *WordSingleStrategy* lub *WordStartPositionToEOFStrategy*.

Dodatkowo klasy wykorzystywane w implementacji wzorca projektowego strategia przyjmują jako parametry typu *char* znak końca pliku (*charEOF*) oraz znak końca klucza (*charEOK*). Dzięki *charEOF* wiemy, jaki znak interpretować jako kończący listę kluczy. Dzięki *charEOK* klasa *WordSingleStrategy* wie, który znak oznacza koniec klucza. Znaki końca klucza i końca pliku są zaliczane do klucza i pojawiają się na jego końcu. Jeżeli kilka znaków końca klucza pojawia się po kluczu to wszystkie są uznawane za jego część (ostatni znak końca klucza kończy go). Jeżeli jakkolwiek znak istnieje po znaku końca pliku to nie jest on brany pod uwagę przy analizie kluczy.

Innymi słowy – jeżeli *charEOF* pojawi się na pozycji *x* to nie ma klucza na pozycji większej od *x*. Jeżeli istnieje ciąg znaków końca klucza po jakimś kluczu, a po tym ciągu pojawia się znak końca pliku to wszystkie znaki *charEOK* i *charEOF* są uznawane za część klucza.

Oprócz metod opisanych w poprzedniej sekcji, konsekwencją implementacji nowej strategii była implementacja dwóch dodatkowych metod, które odpowiadały na pytania dotyczące kluczy zamiast prefiksów.

Metoda *isContainingKey(String searchWord)* klasy *PatriciaTree* jest metodą analogiczną do wcześniej opisanej i zaimplementowanej funkcjonalności *isContainingPrefix(String searchWord)*, a dodatkowo ją wykorzystuje. Zastosowaliśmy tutaj właściwość metody dotyczącej prefiksów. Zauważyliśmy, że wystarczającym warunkiem, aby prefiks był kluczem jest równość długości argumentu *searchWord* oraz klucza zawartego w węźle akceptującym argument jako prefiks. Jeżeli jednak argument *searchWord* nie jest prefiksem żadnego z kluczy to nie jest on też kluczem.

Metoda *findNodeMatchingKey(String searchWord)* klasy *PatriciaTree* jest metodą analogiczną do wcześniej opisanej i zaimplementowanej funkcjonalności *findNodesMatchingPrefix(String searchWord)*. Różnicą między tymi dwoma metodami jest fakt, że *findNodesMatchingPrefix(String searchWord)* zwraca tablicę węzłów drzewa *Patricia* (*PatriciaNode[]*) – akceptujących argument przeszukiwania jako prefiks klucza zawartego w nich – podczas gdy *findNodeMatchingKey(String searchWord)* zwraca jeden węzeł drzewa – którego zawarty w nim klucz jest równy argumentowi wyszukiwania. Nowa metoda odpowiadająca na pytanie dotyczące charakterystyki kluczy drzewa, podobnie jak metoda dotycząca charakterystyki prefiksów, wykorzystuje wcześniej zaimplementowaną metodę *isContaining*, lecz jest nią metoda *isContainingKey* zamiast *isContainingPrefix*.

Rysunek A.1 w załączniku do pracy przedstawia diagram klas paczki *agh.jo.knuth* wyeksportowany ze środowiska programowani *IntelliJ IDEA*.

3.5. Koniunkcyjna postać normalna w drzewie *Patricia* – wykorzystanie implementacji w rozwiązaniu problemu praktycznego

Praktycznym problemem postawionym przed implementacją autorską drzewa *Patricia* stały się pliki *DIMACS CNF* reprezentujące koniunkcyjną postać normalną. Pliki te, ich format oraz samo pojęcie koniunkcyjnej postaci normalnej zostały opisane w sekcji 2.3. Autorska implementacja drzewa *Patricia* miała przetrzymywać zbiór klauzul umieszczony w pliku o rozszerzeniu **.cnf*. Przykład fragmentu takiego pliku wykorzystanego do testowania tej implementacji można zobaczyć w kodzie źródłowym 3.

Kod źródłowy 3. Przykład fragmentu pliku w formacie *DIMACS CNF* o nazwie “Analiza1-gss-20-s100.cnf” wykorzystanego w kodzie źródłowym 5. Plik ma rozmiar równy 1.5 Mb.

```

1 | c 100 20
2 | // [...]
3 | p cnf 31503 94748
4 | 7272 0
5 | 30549 0
6 | 7270 0
7 | 30550 0
8 | 30551 0
9 | 30552 0
10 | 7314 0
11 | 30553 0
12 | 7268 0
13 | 7267 0
14 | // [...]
15 | -31503 -442 -24184 0
16 | 31503 -442 24184 0
17 | 31503 442 -24184 0
18 | // NL at the end of file

```

Implementacja ta miała pozwalać uzyskać odpowiedź na pytania pokroju – czy klauzula x znajduje się w zbiorze klauzul? Jednocześnie powinna ona jak najlepiej grupować klauzule o podobnych podzbiorach literałów. Doszliśmy do wniosku, że odpowiedź na pierwsze pytanie można uzyskać poprzez użycie metody *isContainingKey*, czy *findNodeMatchingKey* klasy *PatriciaTree*. Do znalezienia klauzul o podzbiorze literałów podobnym do zbioru innej klauzuli najlepiej nadaje się metoda *findNodesMatchingPrefix*. Nie jest to oczywiście idealne rozwiązanie ze względu na definicję prefiksu, gdyż różnica w pierwszych literałach dyskwalifikuje dany węzeł jako ten zawierający klucz akceptujący argument przeszukiwania jako prefiks.

Dzięki wysokiemu poziomowi możliwości dostosowania klasy *PatriciaTree* problem ten nie stawiał specjalnych wyzwań przed autorską implementacją. Istniała jednak mała trudność – w pliku *CNF* mogą teoretycznie znajdować się dwie identyczne klauzule z punktu widzenia logicznego, ale zapisane w innej kolejności literałów. Na przykład klauzula $1\ 2\ -3\ 0$ w formacie *DIMACS CNF* jest równa w sensie logicznym klauzuli o postaci $-3\ 2\ 1\ 0$. Gdybyśmy przetwarzali te klauzule jako ciągi znaków wewnątrz drzewa *Patricia* to były one uznane za 2 różne klucze i dodane do drzewa – a byłoby to błędem. Prostem rozwiązaniem tego problemu jest sortowanie literałów wewnątrz każdej z klauzul. Po takim sortowaniu obie przykładowe klauzule przyjęłyby postać $-3\ 1\ 2\ 0$.

W związku z tym chcieliśmy rozszerzyć możliwość personalizacji autorskiego projektu poprzez dostosowanie pliku źródłowego, na podstawie którego budowana jest klasa drzewa *Patricia* i jednocześnie sortować literały wewnątrz każdej klauzuli.

Logiczną odpowiedzią na ten pomysł był konwerter, który na podstawie plików w formacie *DIMACS CNF* tworzyłby pliki, które mogą przyjąć inny format układu tekstu zawierającego informacje o koniunkcyjnej postaci normalnej zawartej w środku, a byłyby zrozumiałe dla klasy *PatriciaTree*.

Klasą implementującą konwerter jest *CNFConverter*. Składa się ona z 2 głównych pól typu *CNFReader* oraz *CNFWriter*. Klasy te odpowiadają odpowiednio za odczytywanie pliku formatu *DIMACS CNF* oraz za zapis do oddzielnego pliku danych odczytanych. Obie klasy wykorzystują klasę *RandomAccessFileContainer* obsługującą funkcjonalność niskopoziomowej klasy *RandomAccessFile*. Sama klasa *CNFConverter* zajmuje się zamianą znaku rozdzielającego numery literałów wewnątrz klauzuli (zamianę spacji na inny znak) oraz znaku rozdzielającego same klauzule (którym w praktyce – w testowanych przykładowych plikach – zawsze jest znak końca linii). W autorskiej implementacji wziętymi pod uwagę kombinacjami znaków końca linii są: ‘\n’, ‘\r’, ‘\n\r’. Kod źródłowy 4 przedstawia plik wynikowy konwertera, na podstawie będzie tworzona struktura drzewa *Patricia*.

Kod źródłowy 4. Przykład fragmentu pliku stworzonego po przekonwertowaniu pliku z kodu źródłowego 3 na plik na podstawie którego zostanie stworzony obiekt klasy *PatriciaTree* o nazwie “Analiza1-gss-20-s100_sorted.txt” wykorzystanego w kodzie źródłowym 5. Plik ma rozmiar równy 1.5 Mb.

```
1 | 7272_0|30549_0|7270_0|30550_0|30551_0|30552_0|7314_0|30553_0|7268_0|// [...]
2 | -31503_-24184_-442_0|-442_24184_31503_0|-24184_442_31503_0|;
```

Szczególnie ciekawą cechą klasy *CNFConverter* w stosunku do *PatriciaTree* jest relacja argumentów ich konstruktorów. Parametr *charOutputEOF* konstruktora klasy *CNFConverter* musi być równy wartości parametrowi *charEOF* konstruktora klasy *PatriciaTree*. Dokładniej mówiąc parametr *charOutputEOF* mówi o tym, jaki znak umieścić na końcu pliku, na podstawie którego zostanie stworzony obiekt klasy *PatriciaTree*. Podobnie *charEOCNF* konstruktora klasy *CNFConverter* musi być równy wartości parametrowi *charEOK* konstruktora klasy *PatriciaTree*. Parametr *charEOCNF* określa na jaki znak zostanie zamieniony znak końca linii z pliku *DIMACS CNF* rozdzielający jedną klauzulę od drugiej. Relacje argumentów tych konstruktorów obrazuje kod źródłowy 5.

Kod źródłowy 5. Przykład użycia konstruktorów klas *CNFConverter* oraz *PatriciaTree* obrazujący relację między argumentami ich konstruktorów.

```

1 final String cnfFilePath = "/src/main/resources/CNF/dimacs";
2 final String cnfFileName = "Analiza1-gss-20-s100.cnf";
3 final char charEOCNF = '|';
4 final char charInputCNFdelimiterReplacement = '_';
5 final char charOutputEOF = ' ';
6 final String patriciaFilePath = "/src/main/resources/CNF/processed";
7 final String patriciaFileName = "Analiza1-gss-20-s100_sorted.txt";
8 final char charEOF = charOutputEOF;
9 final char charEOK = charEOCNF;
10 final WordStrategy wordStrategy = WordStrategy.SINGLE;
11 final Encoding encoding = Encoding.JAVA;
12 CNFConverter cnfConverter = new CNFConverter(
13     cnfFilePath,
14     cnfFileName,
15     patriciaFilePath,
16     patriciaFileName,
17     charEOCNF,
18     charInputCNFdelimiterReplacement,
19     charOutputEOF
20 );
21 cnfConverter.convert();
22 PatriciaTree patriciaTree = new PatriciaTree(patriciaFilePath, patriciaFileName,
    charEOF, charEOK, wordStrategy, encoding);

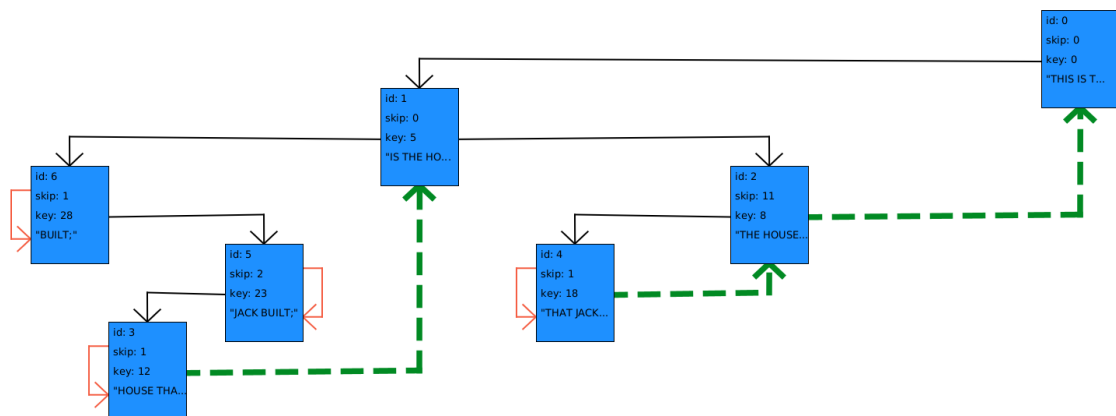
```

3.6. Reprezentacja graficzna stworzonego drzewa *Patricia*

Kolejnym wyzwaniem postawionym przed autorskim programem projektowym była reprezentacja graficzna struktury drzewa *Patricia*. Na szczęście w rozdziale omawianym w tej pracy dotyczącym książki Knuth’a, przedstawia on propozycje schematu struktury drzewa *Patricia*. Jest ona również zamieszczona w niniejszej pracy w postaci rysunku 2.5 oraz rysunku 3.2. Wzorując się na schemacie tych dwóch rysunków udało się nam osiągnąć skalowalną reprezentację graficzną, której przykład przedstawiamy na rysunku 3.4. Odległości pomiędzy węzłami na poziomie drzewa równym 1 zależą od głębokości drzewa gwarantując, że nie zależnie od tego jak głębokie i gęste jest drzewo, jego liście, ani wychodzące z nich krawędzie nie będą na siebie nachodzić.

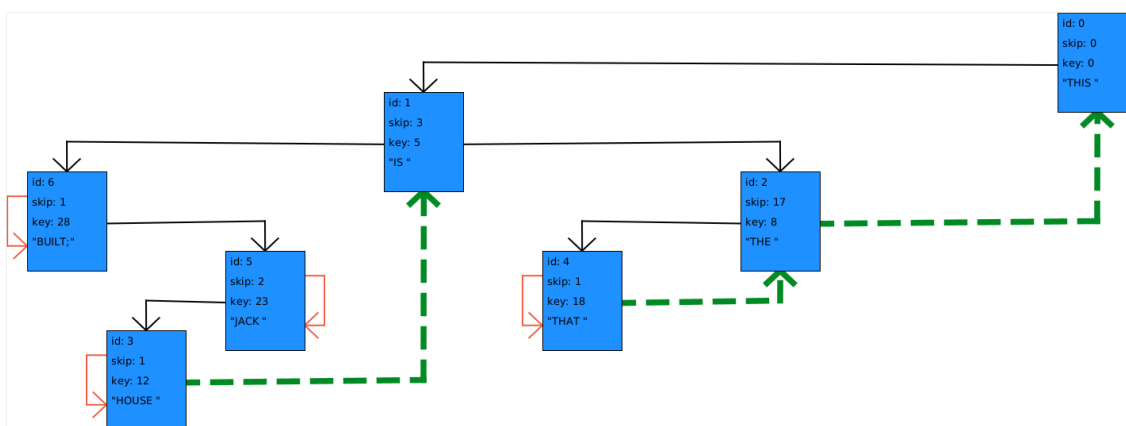
Główną klasą reprezentacji graficznej jest *PatriciaTreeVisualRepresentation* w paczce *agh.jo.ui*. W implementacji głównej klasy i klas, z której korzysta, użyliśmy wtyczki (ang. *plugin*) *org.openjfx:javafx-maven-plugin:0.0.4* do *Apache Maven 3.6.0*. Dodatkowo wykorzystaliśmy biblioteki *org.openjfx:javafx-controls:13* oraz *com.sirolf2009:fxgraph:0.0.3*. Rozszerzenia klas z tych bibliotek wykonaliśmy na podstawie przykładu podanego przez autora drugiej wymienionej biblioteki w popularnym serwisie *StackOverflow* pod adresem <https://stackoverflow.com/questions/30679025/graph-visualisation-like-yfiles-in-javafx>.

Rysunek 3.4. Przykładowa reprezentacja graficzna struktury klasy *PatriciaTree* odpowiadająca przykładowi schematu drzewa *Patricia* 3.2 wykonanego ręcznie w \LaTeX . Jest to reprezentacja graficzna struktury klasy *PatriciaTree* stworzonej na podstawie pliku o zawartości odpowiadającej zawartości z przykładu z książki Knuth'a (oraz omawianego w tej pracy w sekcji 2.2.4). Konstruktor obiektu klasy *PatriciaTree* został wywołany z parametrami: *Encoding Encoding.MIX* oraz *WordStrategy WordStrategy.START_POSITION_TO_EOF*. Interpretacją wartości parametru konstruktora *Encoding* jest fakt, iż klasa korzysta z symulowanej maszyny *MIX* przy zamianie znaków na kod znaku i zamianie z kolei tego kodu na reprezentację binarną (o domyślnej długości bajtu równej 5 bitom). Interpretacją wartości parametru *WordStrategy* jest przyjęcie strategii, gdzie klucz jest fragmentem pliku zaczynającego się w pozycji startowej klucza (przetrzymanywanej w węźle drzewa w postaci pola *key*) i kończącym się znakiem końca pliku.

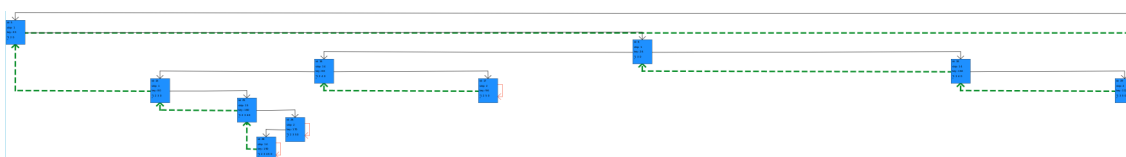


Reprezentacja graficzna była testowana manualnie na 7 przykładach struktury klasy *PatriciaTree*. Część najciekawszych rezultatów testów manualnych została przedstawiona w postaci rysunków 3.5, 3.6, 3.7, które znajdują się na następnej stronie.

Rysunek 3.5. Przykładowa reprezentacja graficzna struktury klasy prostego przykładu *PatriciaTree*, różniącego się od przykładu z rysunku 3.4 jedynie wartościami parametrów wywołania konstruktora. Odmiennymi wartościami konstruktora klasy były: *Encoding Encoding.JAVA* oraz *WordStrategy WordStrategy.SINGLE*. Interpretacją wartości parametru konstruktora *Encoding* jest fakt, iż klasa korzysta z kodowania domyślnego *JAVA* przy zamianie znaków na kod znaku i zamianie z kolei tego kodu na reprezentację binarną (o domyślnej długości bajtu równej 5 bitom). Interpretacją wartości parametru *WordStrategy* jest przyjęcie strategii, gdzie klucz jest fragmentem pliku zaczynającego się w pozycji startowej klucza (przechowywanej w węźle drzewa w postaci pola *key*) i kończącym się znakiem końca klucza, przed pozycją startową następnego klucza w pliku.



Rysunek 3.6. Przykładowa reprezentacja graficzna struktury bardziej złożonego przykładu obiektu klasy *PatriciaTree* różniącego się od przykładów przedstawionych na rysunkach 3.4 oraz 3.5. Widok przybliżony, przedstawia tę samą strukturę co rysunek 3.7.



Rysunek 3.7. Przykładowa reprezentacja graficzna struktury bardziej złożonego przykładu obiektu klasy *PatriciaTree* różniącego się od przykładów przedstawionych na rysunkach 3.4 oraz 3.5. Widok oddalony, przedstawia tę samą strukturę co rysunek 3.6.



3.7. Testy jednostkowe

Przez cały okres tworzenia i rozwijania autorskiej implementacji stopniowo dodawaliśmy testy jednostkowe pozwalające zweryfikować, czy nanoszone zmiany skutkowały spodziewanymi rezultatami. Mimo, że tworzenie ich zajęło nam bardzo dużo czasu, nie wyobrażam sobie, abyśmy bez nich byli w stanie znaleźć wiele pomyłek w naszym rozumowaniu, czy w tak efektywny sposób testować nasze podejrzenia co do trudniejszych do zrozumienia fragmentów algorytmów opisanych przez Knuth'a.

Testy jednostkowe sprawdzają poprawność tworzonej struktury na różnych etapach, przy różnych konfiguracjach konstruktora klasy *PatriciaTree* oraz na różnych przykładach plików źródłowych drzewa *Patricia*.

W skład różnych testowanych etapów tworzenia struktury wchodzi między innymi:

1. Odczyt każdego ze znaków,
2. Zamiana tego znaku na reprezentację binarną zgodną z wybranym kodowaniem,
3. Odczytywanie odpowiedniego fragmentu pliku zgodnie z przyjętą strategią.

W skład testowanych konfiguracji konstruktora klasy *PatriciaTree* wchodzi wszystkie możliwe kombinacje:

1. Ilość bitów przyjętych dla symulowanej maszyny *MIX*,
2. Kodowanie znaków według tablicy znaków maszyny *MIX* lub *UTF-16*,
3. Strategii: *WordStrategy.SINGLE* lub *WordStrategy.START_POSITION_TO_EOF*,
4. Różne kombinacje parametrów znaków *charEOF* oraz *charEOK*,
5. Różne kombinacje konfiguracji konstruktora klasy *CNFConverter*.

W skład testowanych plików, na podstawie których tworzone są obiekty klasy *PatriciaTree* wchodzi 7 plików źródłowych. Zawartość jednego z większych testowanych plików o rozmiarze 1.5 MB została przedstawiona fragmentem kodu źródłowego 3 oraz 4. Drugi większy plik ma rozmiar 8.4 MB i na jego podstawie jest tworzony plik źródłowy drzewa *Patricia*, którego zawartość przypomina plik w formacie *DIMACS CNF*. Oznacza to, że plik wejściowy i wyjściowy konwertera różni się jedynie posortowanymi literałami wewnątrz klauzul, dodanym znakiem końca pliku ';', zamianą znaku końca linii na znak '\n', brakiem linii komentarzy oraz linii problemu. Istnieje również trzeci plik w formacie *DIMACS CNF* testujący współdziałanie klas *CNFConverter* oraz *PatriciaTree*, którego efektem jest reprezentacja wizualna z rysunków 3.6 oraz 3.7. Dodatkowo stworzyliśmy 6 plików na podstawie zmodyfikowanego przykładu omawianego przez Knuth'a w swojej książce [1]. Treść tych plików wygląda następująco:

1. THIS IS THE HOUSE THAT JACK BUILT;
2. THISΦISΦTHEΦHOUSEΦTHATΦJACKΦBUILTΦΠ
3. THISΦISΦΦTHEΦHOUSEΦTHATΦJACKΦBUILTΦΦΦΦΠ This Is Not A Part Of Any Key Because Of EOF Character Π

4. THIS ΦIS ΦΦTHE ΦHOUSE ΦTHAT ΦJACK ΦBUILT ;ΦΦΦΦΠ This Is Not A Part Of Any Key Because Of EOF Character Π
 5. THIS IS THE TEST FILE NUMBER 1 FOR DELETION ONE OF NODES. LONGESTWORDINTREE TODELETE. MORE NODES AFTER NODE TO DELETE;
 6. THIS IS THE TEST FILE NUMBER 1 FOR DELETION ONE OF NODES. LONGESTWORDINTREExTODELETE. MORE NODES AFTER NODE TO DELETE;
- Z dumą możemy powiedzieć, że wyżej omówione testy pokrywają następujący zakres klas, metod oraz linii kodu:

1. Względem głównej paczki (ang. *package*) jaką jest *agh.jo*:
 - Pokrycie klas: 43%,
 - Pokrycie metod: 50%,
 - Pokrycie linii kodu: 45%;
- (a) Względem paczki *agh.jo.cnf.converter* (zawierającą klasy odpowiadające za przetwarzanie pliku w formacie .cnf):
 - Pokrycie klas: 100%,
 - Pokrycie metod: 88%,
 - Pokrycie linii kodu: 94%;
- (b) Względem paczki *agh.jo.knuth* (zawierającą klasy związane ściśle z implementacją drzewa *Patricia*):
 - Pokrycie klas: 100%,
 - Pokrycie metod: 88%,
 - Pokrycie linii kodu: 80%;
- (c) Względem paczki *agh.jo.ui* (zawierającą klasy związane z reprezentacją graficzną struktury drzewa *Patricia*):
 - Pokrycie klas: 0%,
 - Pokrycie metod: 0%,
 - Pokrycie linii kodu: 0%;
- (d) Względem paczki *utils* (zawierającą klasy używane w wielu innych paczkach, na przykład te odpowiadające za obsługę klasy *RandomAccessFile*):
 - Pokrycie klas: 60%,
 - Pokrycie metod: 75%,
 - Pokrycie linii kodu: 63%.

3.8. Instrukcja uruchomienie programu inżynierskiego

Program inżynierski został napisany w sposób pozwalający na uruchomienie go w kilku wariantach. Decyzja podziału na kilka wariantów uruchamiania spowodowana była długim czasem wykonywania i brakiem przejrzystości, gdy wszystkie warianty programu zostawały uruchomione jeden po drugim.

Uruchomienie podstawowej funkcjonalności programu odbywa się po przez przejście do folderu zawierającego foldery *target* oraz *src* i wpisaniu w konsoli komendy przedstawionej w postaci kodu źródłowego 6.

Kod źródłowy 6. Komenda pozwalającą uruchomić program w postaci pliku wykonywalnego *jar* będąc w folderze zawierającym foldery *target* oraz *src*.

```
1 java -jar
   target / TrieTreeImplementations_complete_standalone.jar
   RUN_MODE EXAMPLE_NUMBER OPTIONAL_DO_CONVERT_CNF_FILE
```

Zawartością folderu *target* jest plik wykonywalny w formacie *jar*. Konsekwencją zawarcia programu w postaci pliku wykonywalnego jest konieczność zawarcia dodatkowych plików, na których program przeprowadza operacje w oddzielnym folderze *src/main/resources*. Są to pliki w formacie tekstowym o rozszerzeniu *.cnf* lub *.txt*. Pliki *.cnf* są wykorzystywane przy konwersji ich do plików *.txt*, które z kolei używane są do budowy struktury drzew *Patricia* zawartych w postaci obiektów klasy *PatriciaTree*.

Komenda uruchamiająca plik wykonywalny, przedstawiona kodem źródłowym 6 przyjmuje 3 parametry jako argumenty przedstawianego programu.

1. Pierwszym argumentem jest tryb w jakim chcielibyśmy uruchomić program - **RUN_MODE**:

tekstowy – dla którego wartość parametru powinna być równa **text**;

Jest to tryb w którym informacje na temat drzewa przedstawiane są postaci tekstowej. Krótkie przykłady wyświetlają informacje o poszczególnych kluczach oraz prefiksach drzewa. Długie przykłady ze względu na czytelność tego rozwiązania nie robią tego. Oprócz tego, jeżeli dany przykład ma taką możliwość i zdecydujemy się na to, wyświetlana jest również informacja o dokonaniu konwersji pliku *CNF* na plik źródłowy drzewa *Patricia* (oczywiście przed jego stworzeniem).

graficzny – dla którego wartość parametru powinna być równa **visual**;

Jest to tryb reprezentacji graficznej struktury drzewa zawartej w postaci obiektu klasy *PatriciaTree*. Wykonuje on operacje opisane w trybie tekstowym oraz na ich koniec przedstawia użytkownikowi wizualizację stworzonego drzewa *Patricia*.

2. Drugim argumentem jest numer przykładu jaki chcielibyśmy, aby program przedstawił – **EXAMPLE_NUMBER**:

Przyjmuje on jako swoje wartości liczby całkowite z zakresu od 0 do 9 odpowiadające indeksowi przykładu.

Przykłady o numerach indeksów 5 i 6 są zbyt duże, aby obecna implementacja reprezentacji graficznej sobie z nimi poradziła, dlatego są one zabezpieczone wyjątkiem niepozwalającym na uruchomienie ich w trybie uruchomienia **visual**.

2.0. Wartość 0 odpowiada przykładowi o indeksie 0.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w symulowanej maszynie *MIX*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *START_POSITION_TO_EOF*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:
„THIS IS THE HOUSE THAT JACK BUILT;“
- v. Dodatkowo w pracy można zobaczyć wynik reprezentacji graficznej dla tego przykładu w postaci figury 3.4.

2.1. Wartość 1 odpowiada przykładowi o indeksie 1.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:
„THIS IS THE HOUSE THAT JACK BUILT;“
- v. Dodatkowo w pracy można zobaczyć wynik reprezentacji graficznej dla tego przykładu w postaci figury 3.5.

2.2. Wartość 2 odpowiada przykładowi o indeksie 2.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:
„THISΦISΦTHEΦHOUSEΦTHATΦJACKΦBUILTΦΠ“

2.3. Wartość 3 odpowiada przykładowi o indeksie 3.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:
„THISΦISΦΦTHEΦHOUSEΦTHATΦJACKΦBUILTΦΦΦΦΠ This Is
Not A Part Of Any Key Because Of EOF Character Π“

2.4. Wartość 4 odpowiada przykładowi o indeksie 4.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:
`„THIS ΦIS ΦΦTHE ΦHOUSE ΦTHAT ΦJACK ΦBUILT ;ΦΦΦΦΠ
This Is Not A Part Of Any Key Because Of EOF Character
Π“`

2.5. Wartość 5 odpowiada przykładowi o indeksie 5.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Plik zawierający koniunkcyjną postać normalną ma rozmiar 1,5 Mb, a fragment jego zawartości przedstawia kod źródłowy 3.
- iv. Plik źródłowy obiektu klasy *PatriciaTree* ma rozmiar 1,5 Mb, a jego fragment przedstawia kod źródłowy 4.

2.6. Wartość 6 odpowiada przykładowi o indeksie 6.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Plik zawierający koniunkcyjną postać normalną ma rozmiar 8,4 Mb, a jego zawartość przypomina strukturę pliku *CNF* przykładu o indeksie 5.
- iv. Plik źródłowy obiektu klasy *PatriciaTree* ma rozmiar 8,4 Mb, a jego zawartość jest zbliżona do zawartości pliku *CNF* tego przykładu (o indeksie 6). Rozróżnia go zakończenie każdej linii w postaci znaku '\n', znak końca pliku postaci znaku ';', posortowane literały w każdej z klauzul oraz brak linii komentarzy oraz linii problemu.

2.7. Wartość 7 odpowiada przykładowi o indeksie 6.

- i. Kodowanie znaków jest zgodne z kodowanie znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Plik zawierający koniunkcyjną postać normalną ma rozmiar 532 bajtów, a jego zawartość przypomina strukturę innych plików *CNF* (z przykładów o indeksie 5 i 6).
- iv. Plik źródłowy obiektu klasy *PatriciaTree* ma rozmiar 203 bajtów, a jego zawartość jest zbliżona do zawartości pliku *CNF* tego przykładu (o indeksie 7).

Rozróżnia go zakończenie każdej linii w postaci znaku '\n', znak końca pliku postaci znaku ';', posortowane literały w każdej z klauzul oraz brak linii komentarzy oraz linii problemu.

- v. Dodatkowo w pracy można zobaczyć wynik reprezentacji graficznej dla tego przykładu w postaci figur 3.6 i 3.7.

2.8. Wartość 8 odpowiada przykładowi o indeksie 8.

- i. Kodowanie znaków jest zgodne z kodowaniem znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:

```
„THIS IS THE TEST FILE NUMBER 1 FOR DELETION ONE OF
NODES. LONGESTWORDINTREE TODELETE. MORE NODES AFTER
NODE TO DELETE;“
```

- v. Dodatkowo w pracy można zobaczyć wynik reprezentacji graficznej dla tego przykładu w postaci figury 4.1.

2.9. Wartość 9 odpowiada przykładowi o indeksie 9.

- i. Kodowanie znaków jest zgodne z kodowaniem znaków w wirtualnej maszynie języka *Java*.
- ii. Strategia pobierania kluczy definiująca czym jest znak jest równa *SINGLE*.
- iii. Nie zawiera pliku zawierającego koniunkcyjną postać normalną.
- iv. Zawartością pliku źródłowego obiektu klasy *PatriciaTree* mu odpowiadającego jest:

```
„THIS IS THE TEST FILE NUMBER 1 FOR DELETION ONE OF
NODES. LONGESTWORDINTREExTODELETE. MORE NODES AFTER
NODE TO DELETE;“
```

- v. Dodatkowo w pracy można zobaczyć wynik reprezentacji graficznej dla tego przykładu w postaci figury 4.2.

- 3. Trzecim argumentem jest opcjonalny argument decydujący o tym czy przed stworzeniem obiektu klasy *PatriciaTree* chcielibyśmy wykonać konwersję pliku *CNF* na źródłowy dla naszego obiektu – *OPTIONAL_DO_CONVERT_CNF_FILE*. Przyjmuje wartość *true* lub *false*.

Pliki źródłowe obiektów klasy *PatriciaTree* dla wszystkich przykładów są stworzone przed uruchomieniem programu. Nie należy ich kasować. Można wykasować ich zawartość. Są one nadpisywane w przypadku, gdy argument ten przyjmuje wartość *true*. Gdy nie zostanie podany ten parametr, domyślną przyjmowaną wartością jest *false* wewnątrz programu.

4. Podsumowanie

4.1. Osiągnięcia

1. Bazując na książce „Sztuka programowania“ Donalda E. Knuth’a [1] w wersji angielskiej oraz posiłkując się wersją polską, przedstawienie – w przystępnej formie – głównych konceptów wykraczających poza zakres rozdziału „Przeszukiwanie cyfrowe“ i ćwiczeń go dotyczących.
2. Dokładna analiza tematów nie poruszanych przez Donalda E. Knuth’a, wykorzystująca ponad 30 źródeł w języku polskim, bądź angielskim.
3. Implementacja parametryzowanej klasy *MixMachine* symulującej reprezentację binarną i kodowanie znaków maszyny *MIX*.
4. Implementacja drzewa *Patricia*, według algorytmów opisanych przez Donalda Knuth’a [1] w postaci parametryzowanej klasy *PatriciaTree*.
 - (a) Implementacja operacji *look-up* oraz *search* w kontekście prefiksów oraz *insert* na klasie *PatriciaTree*. Wykonanie na podstawie opisu Knuth’a.
5. Implementacja drzewa *Patricia*, wykorzystującej odmienny sposób reprezentacji kluczy od opisanego przez Knuth’a, za pomocą wzorca projektowego strategia w postaci sparametryzowanej klasy *FileOps*.
 - (a) Implementacja operacji *look-up* oraz *search* w kontekście kluczy oraz *insertAll* na klasie *PatriciaTree*. Wykonanie na podstawie wcześniej zaimplementowanych operacji dotyczących prefiksów.
6. Implementacja konwertera plików *DIMACS CNF* – w postaci parametryzowanej klasy *CNFConverter* – na format plików źródłowych potrzebnych do utworzenia obiektów klasy *PatriciaTree*.
7. Implementacja ponad 570 złożonych testów jednostkowych sprawdzających poprawność tworzonych obiektów klas *PatriciaTree*, *CNFConverter* oraz innych klas przez nie wykorzystywanych.
8. Implementacja reprezentacji graficznej stworzonej struktury drzewa *Patricia*, w postaci klasy *PatriciaTreeVisualization*.

9. Implementacja głównej klasy programu *App* obsługującej argumenty uruchomienia pliku wykonywalnego, oraz automatyzacja przygotowania pliku wykonywalnego za pomocą narzędzia *Maven*.
10. Testy manualne ponad 9 złożonych przykładów reprezentacji graficznej oraz głównej klasy programu *App* na 2 różnych platformach systemowych (*Unix* oraz *Windows*).

4.2. Rezultaty implementacji oraz analiza pracy

Implementacje uważamy za udaną. Udało się nam zaimplementować algorytm opisany z myślą o innym, nisko poziomowym języku, maszynie posiadającej kompletnie inną tablicę znaków oraz inną reprezentację binarną. Dodatkowo rozszerzyliśmy ten algorytm dla przypadków ogólnych – w postaci strategii pojedynczego słowa. Wykorzystanie implementacji w rozwiązaniu problemu koniunkcyjnej postaci normalnej przebiega zgodnie z postawionymi, podstawowymi założeniami. Reprezentacja graficzna spełnia swoją rolę obrazowania struktury stworzonego drzewa *Patricia*. Testy jednostkowe potwierdzają poprawność wprowadzanych zmian do implementacji.

Mając to wszystko na uwadze, oczywiście nie obyło się bez swojego rodzaju założeń upraszczających. Zakres programu projektowego ze względu na jego możliwości jest bardzo szeroki i można odnieść wrażenie, że zaczyna on wiele różnych tematów nie doprowadzając ich do zwięźczenia. W związku z tym w wielu miejscach pojawiają się koncepcje upraszczające, bądź porzucenie pewnych funkcjonalności na rzecz rozwinięcia innych. Oczywiście można też w tym miejscu zauważyć okazję do kontynuowania tego tematu, poszerzania wglądu i swojego wkładu do niego.

4.2.1. Implementacja drzewa *Patricia* według Knuth’a

Zgodnie z tym co opisaliśmy w poprzedniej sekcji cała zaimplementowana funkcjonalność klasy *PatriciaTree* jest udana pod względem poprawności i prędkości przetwarzania dużej ilości danych.

Oczywiście, aby w pełni stwierdzić czy implementacja jest optymalna należałoby porównać ją do innych implementacji przy użyciu testów wydajnościowych pamięciowych i czasu wykonywania. Na pewno również w kontekście specyfikacji samego języka *Java* dałoby się zoptymalizować pewnie działania.

W autorskiej implementacji drzewa *Patricia* brakuje kilku operacji opisanych w części teoretycznej. Takimi operacjami są: usuń (ang. *delete*), poprzednik (ang. *predecessor*), następca (ang. *successor*), minimum oraz maksimum.

Do samej budowy drzewa oraz jego przeszukiwania w kontekście odnajdywania prefiksów i kluczy, na których skupiamy się w niniejszej pracy, wymienione wyżej operacje są niepotrzebne.

Z powodu zaplanowanego zakresu pracy inżynierskiej pominęliśmy ich implementację. Plan pracy nigdy nie przewidywał implementacji wszystkich możliwych do przeprowadzenia działań na drzewie. Dodatkowo Knuth nie opisuje nawet w swoich ćwiczeniach oraz rozwiązaniach algorytmu tych operacji. Nie pojawiają się one nawet w formie nawiązania czy wspomnienia o nich.

Szczególnie skomplikowaną do zaimplementowania metodą okazuje się metoda usuwania klucza. W obecnej implementacji należałoby rozróżnić dwa możliwe rozwiązania algorytmu usuwania klucza z drzewa.

Pierwszym rozwiązaniem jest usuwanie jedynie węzła ze struktury drzewa, pozostawiając plik nienaruszony. Wiązałoby by się to jednak z przebudową całego poddrzewa węzła usuniętego, a potencjalnie mogłoby by to wymagać modyfikacji również struktury węzłów wychodzący poza to pod-drzewo – ze względu na dowiązania węzłów do ich przodków.

Drugim rozwiązaniem jest usuwanie klucza z pliku, z czym wiązałoby się przesunięcie całej zawartości pliku w lewo o d znaków (w przedstawionej implementacji bajtów odpowiadających d znakom) od miejsca w którym zaczynał się usunięty klucz. Konsekwencją takiego zabiegu byłaby zmiana pozycji startowej klucza zawarta we wszystkich węzłach reprezentujące klucze właśnie zaczynające się po pozycji startowej wykasowanego klucza. Struktura drzewa *Patricia* nie pozwala na łatwe znalezienie węzłów zaczynających się po określonej pozycji, a więc musielibyśmy sprawdzić wszystkich węzły lub wyłuskiwać każdy węzeł ze względu na jego klucz – co byłoby skomplikowane ze względu na fakt, że bylibyśmy w trakcie modyfikacji drzewa.

Rysunki 4.1 i 4.2 przedstawiają różnicę między drzewami *Patricia*, z których drzewo 4.1 zawiera węzeł z dodatkowym kluczem, a drzewo 4.2 nie zawiera tego węzła. Ma to na celu obrazować uproszczony wpływ wykasowania węzła ze struktury drzewa. Efekt taki uzyskany został poprzez stworzenie dwóch źródłowych plików dla obiektów klasy *PatriciaTree*, które różnią się jedynie fragmentem określającym istnienie dodatkowego klucza lub jego brak. Fragment ten ma postać w przypadku drzewa z rysunku 4.1 „LONGESTWORDINTREE TODELETE. “, a w przypadku drzewa z rysunku 4.2 „LONGESTWORDINTREEXTODELETE. “. Różnica w tych dwóch fragmentach jest subtelna, ale znacząca. Poprzez zmianę znaku spacji na znak 'x' dwa klucze z rysunku 4.1 stały się jednym kluczem na rysunku 4.2. Dzięki tym ilustracją możemy zaobserwować jak duży wpływ na strukturę drzewa może mieć operacja usunięcia jednego węzła – nawet przyjmując upraszczające założenia.

Zgodnie z opisem w sekcji 3.4, modyfikacja algorytmów Donalda Knuth’a dotyczących drzewa *Patricia* powiodła się. Dzięki dodaniu parametrów typu *char* konstruktora klasy *PatriciaTree*, określającego znak końca klucza modyfikacja ograniczała się w praktyce do stworzenia klasy implementującej oddzielny algorytm odczytywania klucza z pliku.

Sprawdzenie rezultatów poprawności algorytmów wykonaliśmy za pomocą testów jednostkowych, opisanych w sekcji 3.7.

Jednym z dowodów na brak znacznych modyfikacji względem algorytmów Knuth’a oraz struktury drzewa jest fakt, iż obie interpretacje tego czym jest klucz są dostępne w jednej klasie *PatriciaTree* – a wybór ich odbywa się za pomocą wzorca projektowego strategia, który dostarcza jedną z klas dziedziczących po klasie *FileOpsStrategy*.

Drugim dowodem na nieznaczny wpływ modyfikacji interpretacji tego czym jest klucz jest reprezentacja graficzna. Przykładowymi rysunkami przedstawiającymi przykład reprezentacji graficznej struktury drzewa dla takiej samej zawartości plików, ale różnej strategii pobierania klucza z pliku (interpretacji tego czym jest klucz) są rysunki 3.4 i 3.5.

Dodatkową własnością, którą można zaobserwować na tych rysunkach jest fakt – iż mimo oprócz różnych strategii te dwa obiekty korzystają z różnych kodowań znaków i reprezentacji binarnych – to struktura drzewa nie zmienia się w sposób znaczący. Różnicę w strukturze drzewa spowodowaną przez kodowanie znaków można zaobserwować dopiero na znakach wymagający do ich reprezentacji różnej ilości bajtów. Przykładami takich znaków są: θ , Φ , Π .

4.2.3. Koniunkcyjna postać normalna – wykorzystanie autorskiej implementacji w rozwiązaniu problemu praktycznego

Implementacja konwertera plików w formacie *DIMCAS CNF* wykorzystuje klasę *RandomAccessFile*, której obsługa została już napisana i wykorzystana wcześniej w programie projektowym. Niestety prędkość konwertowania pliku nie jest zadowalająca i może być to spowodowane właśnie wyborem klasy obsługującej pracę na plikach. Możliwe, że użycie kombinacji klas *File*, *FileReader*, *BufferedReader* przez klasę *CNFReader* (a przez klasę *CNFWriter*: *File*, *FileWriter*, *BufferedWriter*) skutkowałoby szybszym przetwarzaniem plików.

Przykłady struktury klasy *PatriciaTree* z klasy *agh.jo.Examples* mają możliwość nie konwertowania plików poprzez ustawienie finalnego, statycznego pola *DEFAULT_IS_DO_CONVERT_CNF_SOURCE_FILE_TO_PATRICIA_SOURCE_FILE* klasy *App* na wartość *false* dla przypadków kiedy nie nastąpiła zmiana pliku źródłowego *CNF* i parametrów konstruktów klas *PatriciaTree* i *CNFConverter*.

Jednym z wcześniej wspomnianych założeń upraszczających było założenie przyjęte przy konwertowaniu plików w formacie *DIMACS CNF* dotyczące tego, że nie mogą się w nich pojawić dwie identyczne klauzule z punktu widzenia logiki Bool'a. Dzięki temu konwerter nie musi sprawdzać, czy klauzula wstawiana do pliku źródłowego drzewa już w nim nie występuje. W efekcie implementacja drzewa *Patricia* wyrzuca wyjątek mówiący o tym, że do struktury nie może zostać wstawiony klucz będący prefiksem już istniejącego klucza.

Kolejnym założeniem uproszczającym jest to, że każda klauzula w pliku *DIMACS CNF* kończy się znakiem (lub kombinacją znaków) końca linii. Specyfikacja plików w tym formacie mówi o przypadkach, w których może się zdarzyć, że dwie klauzule są w tej samej linii, a rozdziela je ciąg znaków *klauzula1* + " 0 " + *klauzula2*. Specyfikacja mówi również o nieregularnych przypadkach, gdzie ostatnia klauzula w pliku nie jest zakończona kombinacją znaków *ostatnia_klauzula* + " 0 ". Dodatkowo dopuszczane są przypadki, w których linie komentarzy przeplatają się z liniami klauzul.

Następnym założeniem uproszczającym, niezwiązanym już ze specyfikacją plików w formacie *DIMACS CNF*, jest wymóg klasy *CNFConverter* istnienia pliku wejściowego i wyjściowego konwertera. Podobnie klasa *PatriciaTree* wymaga istnienia pliku wejściowego, który jest plikiem wyjściowym konwertera. Zapobiega to sytuacji, w której użytkownik mógłby popełnić błąd podając inne ścieżki czy nazwy plików dla tych dwóch klas i mógłby stanąć on przed zagadką „Dlaczego skoro konwerter poprawnie działa to klasa drzewa *Patricia* wyrzuca wyjątek związany z brakiem pliku, który przecież konwerter *NA PEWNO* tworzy? “. Oczywiście nigdy się nie zdarzyło tak, żeby użytkownikiem w takim scenariuszu byli autorzy tej pracy inżynierskiej – czyli my.

Z kolei wnioskiem wynikającym z zastosowania struktury jaką jest drzewo *Patricia* do problemu plików zawierających koniunkcyjną postać normalną jest fakt, że nie jest ona kompatybilna z problemem. Drzewa typu *Trie* korzystają z własności wynikających z kolejności ciągów znaków, podczas gdy klauzula, która jest alternatywą literałów – nie dba o kolejność literałów. Dlatego do problemu, w którym kolejność wartości nie ma znaczenia lepszym wyborem były by *hashowe*-funkcje (ang. *hash-functions*). Inspiracją dla tego rozwiązania było przedstawienie ich praktycznego wykorzystania w wideo-artykule na przykładzie aplikacji *Shazam* (aplikacji znajdującej nazwę utworu muzycznego na podstawie jego fragmentu rozpoznanego przez mikrofon telefonu) przez kanał *RealEngineering* na *YouTube* [30].

Dodatkowym problem, który dla ułatwienia ignorujemy jest fakt, że z logicznego punktu widzenia drzewa typu *Trie* są alternatywą kluczy w nim zawartych, podczas gdy koniunkcyjna postać normalna jest koniunkcją klauzul. Bardziej nadającym się do tego rozwiązania – z tego punktu widzenia – problemem jest problem *DNF* (dysjunkcyjna postać normalna), który jest alternatywą koniunkcji literałów.

Nie rozwiązuję to jednak problemu jakim jest dowolność kolejności literałów w klauzulach alternatyw czy koniunkcji literałów.

4.2.4. Reprezentacja graficzna stworzonego drzewa

Dodanie reprezentacji graficznej jako dodatkowego elementu edukacyjnego oraz testowego zakończyło się sukcesem. Niekonwencjonalna budowa drzewa *Patricia* – a dokładniej – orientacja węzłów względem siebie i gałęzi obrazujących relację tych węzłów – wymagała autorskiej implementacji tworzenia wizualnej struktury drzewa. Niestety wymagało to dodania dodatkowych „ukrytych” węzłów na gałęziach drzewa, aby było zrozumiałe z którego węzła wychodzi dana krawędź i do którego wchodzi. Spowodowało to, że obecnie zaimplementowana funkcjonalność przesuwania węzłów drzewa *Patricia* w czasie działania programu nie spełnia swojej funkcji, gdyż gałęzie zostają w pozycji startowej i niestety zaburza ona czytelność grafu. Decyzję czy warto reorganizować położenie węzłów drzewa *Patricia* na grafie pozostawiam użytkownikowi.

Jedynym ograniczeniem wizualizacji struktury klasy *PatriciaTree* jest ilość krawędzi w drzewie – bo na tym etapie przetwarzania program nie daje sobie rady z ilością wymaganej pamięci. Klasa *PatriciaTreeVisualization* nie jest w stanie sobie poradzić ze strukturami klasy *PatriciaTree* stworzonej na podstawie dwóch dużych plików o rozmiarze 1.5 i 8.4 Mb. Jest to dosyć ciekawy problem, którego rozwiązaniem nie może być po prostu zwiększenie ilości pamięci RAM przydzielonej programowi, gdyż zawsze może istnieć większy przykład drzewa *Patricia*, wymagający większej ilości pamięci RAM. Rozwiązaniem tego problemu mogłoby być dynamiczne tworzenie i kasowanie obiektów odpowiednio widocznych i nie widocznych obecnie na ekranie oraz ograniczenie obszaru jaki może widzieć użytkownik korzystając z funkcjonalności oddalenia widoku. Inspiracją tego rozwiązania jest rozwiązanie popularnie stosowane w dzisiejszych grach, gdzie jedynie obiekty widoczne w następnej klatce są renderowane, a stare obiekty są niszczone na rzecz wolnej pamięci.

4.3. Możliwości rozwoju

Implementacja wykonana w zakresie tego projektu inżynierskiego w formie klasy *PatriciaTree* jest uniwersalna i nadaje się do zastosowania przy problemach dużych zbiorów, których elementy mają określoną kolejność – są ciągami liczb czy innych znaków.

Możliwością rozwoju samej implementacji postaci klasy *PatriciaTree* jest implementacja operacji pominiętych w obecnej wersji (usuń, następcą, poprzednik, minimum, maksimum). Dodatkowo można rozwinąć projekt o algorytmy umożliwiające modyfikację

pliku źródłowego drzewa *Patricia*. Takim algorytmem może być na przykład algorytm dodawania lub usuwania - do drzewa i jednocześnie pliku – nowego klucza. Dokładniej opisany został ten problem w sekcji 4.2.1.

Rozwojem rozważań tematu dotyczącego koniunkcyjnej postaci normalnej może być próba zastosowania *hash*-funkcji do szybszego przeszukiwania zbioru klauzul. Dokładniej problem ten opisałem w sekcji 4.2.3.

Kolejną okazją do rozwoju tego projektu jest modyfikacja implementacji reprezentacji graficznej tak, aby niezależnie od rozmiaru drzewa *Patricia*, generacja wizualnej warstwy projektu zawsze kończyła się sukcesem. Opis propozycji rozwiązania tego problemu opisałem w sekcji 4.2.4.

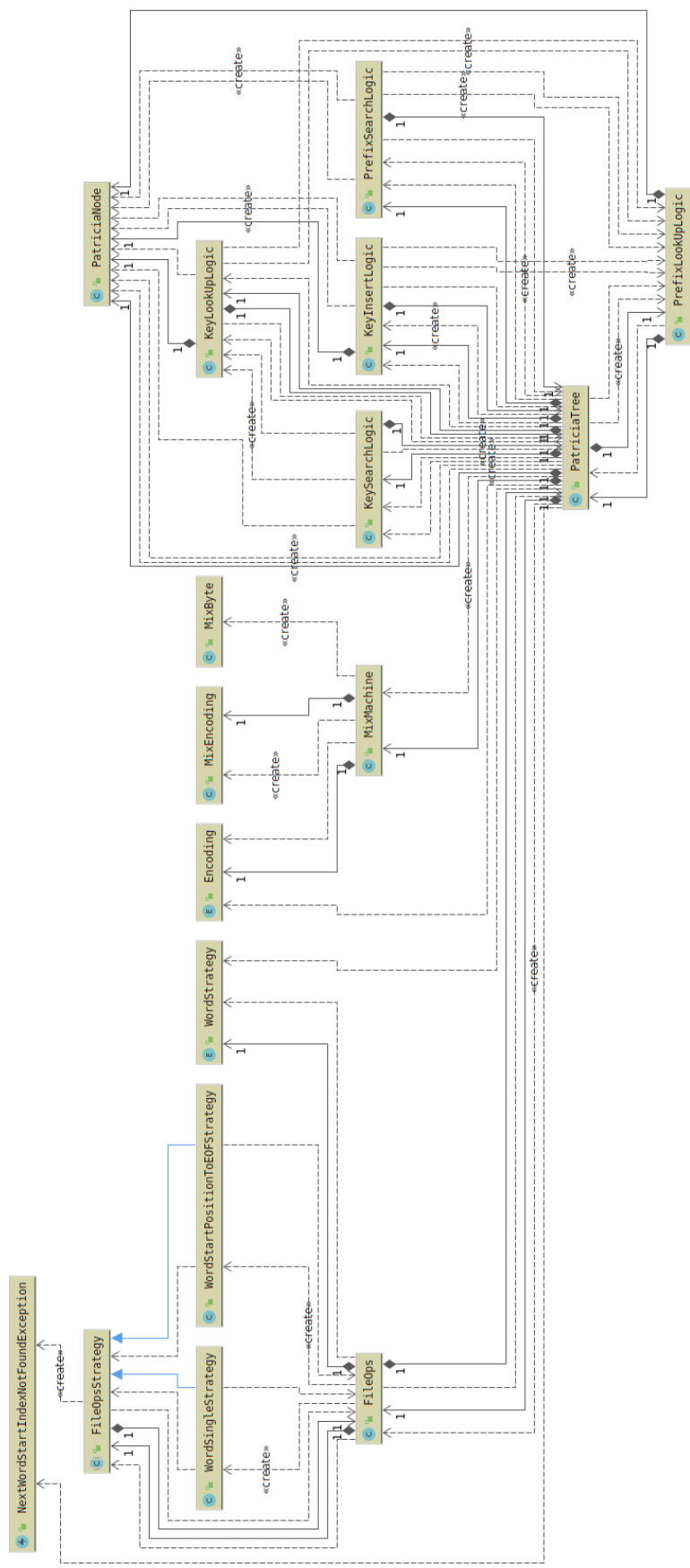
Bibliografia

- [1] Donald E. Knuth: *The Art of Computer Programming*, vol. 3, <https://linuxnasm.be/media/pdf/donald-knuth/taocp/volume-3/taocp-vol3-sorting-searching.pdf>, Boston, MA 02116, USA: Mathematical Sciences Publishers (MSP), June 2014, chap. 6.3. Digital Searching, pp. 624–629
- [2] Venkateswara Rao Sanaka: “Trie | (Insert and Search)”, in: <https://www.geeksforgeeks.org>
- [3] Sakshi Tiwari aka. vero: “Advantages of Trie Data Structure”, in: <https://www.geeksforgeeks.org>
- [4] Fatos Morina: “The Trie Data Structure in Java”, in: July 2019, <https://www.baeldung.com>
- [5] Amogh Avadhani: “How to build a Trie Tree in Java”, in: May 2018, <https://medium.com>
- [6] Vaidehi Joshi: “Trying to Understand Tries”, in: July 2017, <https://medium.com>
- [7] Thomas H. Cormen & Charles E. Leiserson & Ronald L. Rivest & Clifford Stein, 3rd ed., Cambridge, Massachusetts, USA | London, England: The MIT Press, 2009, chap. 20-2, p. 558
- [8] Axel Thue: *Skifter udgivne af Videnskabs-Selskabet i Christiania, MathematiskNaturvidenskabelig Klasse (1912), No. 1*, Oslo, 1977, pp. 413–477
- [9] René de la Briandais: “File searching using variable length keys”, in: *Proc. Western Joint Computer Conference* 15, 1959, pp. 295–298
- [10] Edward Fredkin: „Trie Memory”, w: *Commun. ACM* 3.9, wrz. 1960, s. 490–499, ISSN: 0001-0782, DOI: 10.1145/367390.367400
- [11] Wikipedia: “M-ary tree”, in: Sept. 2019, <https://www.en.wikipedia.org>
- [12] Donald E. Knuth: *The Art of Computer Programming*, vol. 1, Boston, MA 02116, USA: Addison Wesley Longman, May 1997, chap. 2.3. Trees, p. 308

- [13] Erik Rosenthal Andrew Matusiewicz Neil V. Murray: „Tri-Based Set Operations and Selective Computation of Prime Implicates”, w: *Foundations of Intelligent Systems. ISMIS 2011. Lecture Notes in Computer Science* 6804, czer. 2011, s. 203–213
- [14] Donald R. Morrison: „Patricia - Practical Algorithm To Retrieve Information Coded in Alphanumeric”, w: *Journal of the Association for Computer Machinery* 15.4, paź. 1968, s. 514–534, ISSN: 0001-0782
- [15] Hemant Tiwari et al. S. M. Meena Vanraj Vala: „File Based Trie for Embedded Devices”, w: *9th International Conference on Advanced Computing (IACC)* 9, sty. 2019, s. 163–170, ISSN: 2473-3571
- [16] Till Knollmann i Christian Scheideler: *A Self-Stabilizing Hashed Patricia Trie*, 2018, arXiv: 1809.04923 [cs.DC]
- [17] Keith Schwarz: “CS 166: Data Structures, x-Fast and y-Fast Tries”, in: <http://web.stanford.edu/class/archive/cs/cs166/cs166.1146/>
- [18] Esolang: “MIX (Knuth)”, in: Nov. 2019, <https://esolang.org>
- [19] Interviewcake.com: “Bit Shifting (Java)”, in: <https://interviewcake.com>
- [20] Wikipedia: „Koniunkcyjna postać normalna”, w: kw. 2019, <https://en.wikipedia.org>
- [21] Wikipedia: “Conjunctive normal form”, in: Dec. 2019, <https://en.wikipedia.org>
- [22] Wikipedia: “Canonical normal form”, in: Dec. 2019, <https://en.wikipedia.org>
- [23] Wikipedia: „Klauzula (Matematyka)”, w: czer. 2019, <https://en.wikipedia.org>
- [24] Wikipedia: “Clause (Logic)”, in: Dec. 2019, <https://en.wikipedia.org>
- [25] Wikipedia: „Formuła atomowa”, w: sierp. 2019, <https://en.wikipedia.org>
- [26] Wikipedia: „Dysjunkcyjna postać normalna”, w: paź. 2018, <https://en.wikipedia.org>
- [27] John Burkardt: “CNF Files”, in: June 2008, <https://www.sc.fsu.edu/>
- [28] “All about CNF Files”, in: <https://filext.com>
- [29] “DIMACS CNF (.cnf) Format”, in: <https://www.maplesoft.com>
- [30] Real Engineering: “How Shazam Works - Hash Functions”, in: Dec. 2018, <https://www.youtube.com>

Załączniki

Rysunek A.1. Diagram klas paczki agh.jo.knuth



Powered by yFiles