Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wioletta Wajdlich

Nr albumu: 1062732

Implementacja drzew trie w języku Python

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2018

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Pragnę serdecznie podziękować Promotorowi niniejszej pracy, Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za ogromne zaangażowanie, poświęcony czas, rady oraz wsparcie w powstawaniu mojej pracy licencjackiej.

Streszczenie

Drzewo trie jest to drzewo poszukiwań, które w węzłach przechowuje fragmenty kluczy. Wszyscy potomkowie węzła mają wspólny prefix klucza związanego z danym węzłem. Drzewa trie wspierają następujące operacje: wstawianie, usuwanie, wyszukiwanie klucza, wyszukiwanie wszystkich kluczy o podanym prefiksie. W wielu zastosowaniach drzewa trie mogą zastępować tablice hashowalne lub słowniki.

W pracy przedstawiono implementację drzew trie w języku Python. Klucze przechowywane w drzewie trie są stringami, a wartości mogą być dowolnymi obiektami pythonowymi. W kodzie drzewo trie występuje w reprezentacji na lewo syn, na prawo brat, która konwertuje drzewo z wieloma potomkami do drzewa binarnego. Jest to rozwiązanie wydajne pamięciowo z wolniejszym wyszukiwaniem potomka danego wezła.

W pracy zaimplementowano także skompresowane drzewa trie, zwane drzewami Patricia. Są to drzewa trie o zredukowanym rozmiarze, w których węzły będące jedynymi dziećmi zostają złączone ze swoim rodzicem. Przez to węzły muszą przechowywać fragmenty kluczy o różnej długości, ale ścieżki w drzewie są skrócone, co zmniejsza liczbę koniecznych porównań.

Przygotowane moduły posiadają dokumentację zgodną z regułami Pythona i stylem pakietu NumPy. Testy jednostkowe stworzono przy użyciu modułu unittest. Klasy dla drzew trie dziedziczą z abstrakcyjnych klas bazowych dla zmiennych odwzorowań, dlatego kod może zastępować słowniki pythonowe, o ile klucze są stringami.

Słowa kluczowe: drzewo trie, drzewo Patricia, drzewo pozycyjne, abstrakcyjna klasa bazowa, iterator

English title: Python implementation of tries

Abstract

A trie or a prefix tree is a search tree where the tree nodes store parts of keys. All the descendants of a node have a common prefix of the key associated with that node. Tries support insertion, deletion, searching, finding all keys with common prefix. Tries can replace hash tables or dictionaries in many applications.

Python implementation of tries is presented. Keys stored in a trie are strings, values can be any Python objects. Left-child right-sibling representation is used which converts a multi-way trie in a binary tree. This is a memory efficient solution with a slower looking up a node's children.

Compressed tries or Patricia trees are also implemented. They are tries reduced in size because each node that is the only child is merged with its parent. As a result nodes store parts of keys with variable lengths, tree paths are shorter, and the number of necessary comparisons is lower.

Prepared modules are documented according to Python rules and the NumPy package style. Unit tests are created using the unittest module. Trie classes inherit from the abstract base class for mutable mappings and that is why the code can replace Python dictionaries if keys are strings.

Keywords: trie, prefix tree, radix tree, Patricia tree, abstract base class, iterator

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy są *drzewa trie* [1]. Nazwa drzew pochodzi od angielskiego słowa *retrieval* (odczyt), co odróżnia je od zwykłych drzew (ang. *tree*). Drzewo trie jest to drzewo poszukiwań przechowujące w węzłach fragmenty kluczy, co pozwala przyspieszyć wyszukiwanie. Zwykłe drzewa przeszukiwań (oparte na drzewach BST) przechowują w węzłach całe klucze. Klucze są ciągami znaków z pewnego skończonego alfabetu. Mogą to być znaki zwykłego alfabetu, cyfry, ciągi bitów, albo ciągi dowolnych elementów z porządkiem leksykograficznym.

Drzewa trie pozwalają wykonać następujące zadania [1].

— Sprawdzenie w czasie $O(k)$, czy klucz o długości k jest w drzewie.
— Znalezienie w czasie $O(m)$ najdłuższego prefiksu występującego w drzewie dla podanego klucza, gdzie m to długość prefiksu.
— Wyszukanie wszystkich kluczy o podanym prefiksie.

Drzewa trie reprezentuje się jako drzewa łączone z korzeniem, przy czym węzły mogą mieć różną liczbę dzieci. Z korzeniem jest związany pusty klucz. W praktyce stosuje się kilka sposobów organizowania łączy.

— Każdy węzeł może mieć ustaloną liczbę łączy do dzieci, równą liczbie znaków skończonego alfabetu. Wtedy wiele łączy jest pustych.
— Każdy węzeł może mieć tablicę łączy do dzieci o zmiennej długości. Wtedy trzeba w węźle przechowywać jeszcze długość tablicy.
— Każdy węzeł może mieć dokładnie dwa łącza: pierwsze do lewego dziecka, drugie łącze do prawego brata. Wtedy efektywnie uzyskujemy drzewo binarne i wydajne wykorzystanie pamięci. Jest to reprezentacja *na lewo syn, na prawo brat* (ang. *left-child right-sibling representation*).

Modyfikacją drzewa trie jest *skompresowane drzewo trie* lub inaczej *drzewo Patricia* (ang. *Patricia tree*, *radix tree*). Nazwa drzewa pochodzi od skrótu PATRICIA utworzonego przez Morrisona (1968): *Practical Algorithm To Retrieve Information Coded in Alphanumeric* (Praktyczny algorytm wyszukiwania informacji zakodowanych w formie alfanumerycznej) [2]. W drzewie Patricia unika się sytuacji, w której węzeł ma dokładnie jedno dziecko, bo wtedy jedyne dziecko jest łączone ze swoim rodzicem. Węzły drzewa mogą przechowywać łańcuchy znaków, a nie tylko pojedyńcze znaki. Dzięki temu skracane są ścieżki w drzewie i zmniejsza się liczba porównań przy wyszukiwaniu klucza.

Drzewa trie mają szereg zastosowań.

— Drzewa trie mogą zastępować tablice hashowalne. Zaletą jest brak kolizji kluczy, nie jest potrzebna funkcja hashująca. Wyszukiwanie danych

w najgorszym przypadku jest szybsze, w porównaniu do tablicy hashującej z kolizjami.
— Drzewa trie są stosowane do przechowywania słowników wyrazów w telefonach komórkowych do autouzupełniania wyrazów (ang. *word completion*) lub przewidywania tekstu (ang. *predictive text*).
— Drzewo trie przechowuje indeks wyrazów w drzewie dyskryminacyjnym (ang. *discrimination tree*).

1.1. Cele pracy

Celem pracy jest implementacja drzew trie i drzew Patricia w języku Python [3]. Pomysł tematu pracy pojawił się przy szukaniu zastosowań drzew o dowolnym stopniu rozgałęzień. W książce Cormena i in. [4] można znaleźć wzmiankę o reprezentacji *na lewo syn, na prawo brat*, a także zadania wskazujące na różne możliwości organizacji informacji w węzłach. Nie ma tam jednak pseudokodów opisujących operacje na drzewie.

Naszym celem jest przygotowanie kodu ilustrującego operacje na drzewie z dowolnym stopniem rozgałęzień, na bazie ważnego zastosowania takich drzew. Język Python jest sprawdzonym narzędziem w takiej sytuacji, ponieważ był już wykorzystany do implementacji różnych rodzajów drzew binarnych (binarne drzewa poszukiwań, drzewa czerwono-czarne, drzewa AVL, drzewa splay) [5].

1.2. Organizacja pracy

Praca jest zorganizowana następująco. Rozdział 1 podaje cele pracy. Rozdział 2 zawiera krótki opis języka Python. W rozdziale 3 omówiono zasady testowania oprogramowania i przygotowywania testów jednostkowych. Rozdział 4 zawiera podstawowe definicje dotyczące grafów, natomiast rozdział 5 omawia drzewa, w tym drzewa trie i drzewa Patricia. Opis implementacji drzew znajduje się w rozdziale 6. Rozdział 7 stanowi podsumowanie pracy.

2. Python

Python jest językiem skryptowym wysokiego poziomu rozwijanym jako projekt typu Open Source. Został stworzony w 1990 roku przez Guido van Rossuma. Obecnie własność intelektualna jest własnością niedochodowej organizacji Python Software Foundation. Nazwa języka pochodzi od popularnego serialu komediowego Latający cyrk Monty Pythona, który był emitowany w latach siedemdziesiątych przez telewizję BBC.

Python jest w pełni multiplatformowy – interpretery języka są dostępne na wszystkie wiodące platformy, w tym systemy wbudowane. Python jest językiem dynamicznie typowanym. Posiada zautomatyzowany system zarządzania pamięcią oparty o *garbage collection*. Python nie jest skoncentrowany na żadnym paradygmacie programowania i umożliwia pisanie w różnych stylach takich jak:
— programowanie obiektowe,
— programowanie imperatywne,
— programowanie funkcyjne,
— programowanie proceduralne.

2.1. Dokumenty PEP

Python jest rozwijany za pomocą dokumentów PEP (ang. *Python Enhancement Proposal*). To kluczowe dokumenty w ekosystemie języka Python [10]. W zależności od sytuacji pełnią funkcję:
— informacyjną - zawierają informacje potrzebne do pracy z interpreterem oraz plany dotyczące nowych wersji języka,
— standaryzacyjną - zawierają wytyczne dotyczące stylu kodu i dokumentacji oraz opisują proces wytwarzania oprogramowania w języku Python,
— projektową - zawierają opisy zmian oraz nowych funkcjonalności w Pythonie.

Każdy PEP jest szczegółowo oceniany przez społeczność programistów języka Python. Do niedawna każdy PEP był również recenzowany przez autora języka, aczkolwiek w lipcu 2018 roku postanowił odizolować się od procesu decyzyjnego związanego z Pythonem. Guido van Rossum nie wyznaczył swojego następcy i dalszy rozwój języka pozostawił w rękach społeczności.

Wśród wszystkich PEP-ów można wyróżnić kilka szczególnych. PEP 0 to zbiór wszystkich zgłoszonych dokumentów PEP. Można w nim znaleźć wszystkie oficjalnie zaakceptowane propozycje oraz dokumenty, które będą w przyszłości rozpatrywane. PEP 8 opisuje konwencje oraz wytyczne dotyczące stylu, które powinny być przestrzegane przez programistów. W dokumencie PEP 404 oficjalnie zamknięto pracę nad Pythonem w wersji 2. Ostatnia wydana wersja w tej gałęzi to Python 2.7.

W grudniu 2008 roku został oficjalnie wydany Python 3.0 i od tego momentu wyłącznie ta wersja jest rozwijana. Python 3 nie bierze pod uwagę przy projektowaniu wstecznej kompatybilności z Pythonem 2, zatem skrypty tworzone w wersji 2 mogą mieć poważne problemy z uruchomieniem w najnowszej gałęzi języka. Mimo wszystko, jeśli aplikacja nie jest skomplikowana, można utrzymywać kod w taki sposób, aby był on kompatybilny z obiema gałęziami bez korzystania z dodatkowych narzędzi i technik.

Kluczowe zmiany występujące w Pythonie 3, które łamią wsteczną kompatybilność języka:

— Zmiany składni, które usuwają lub modyfikują istniejące elementy języka lub dodają kompletnie nowe elementy.
— Zmiany w standardowej bibliotece.
— Zmiany, które tyczą się wbudowanych struktur danych i typów.

3. Testowanie oprogramowania

Testowanie oprogramowania polega na weryfikacji poprawności działania oprogramowania i dostarczaniu informacji na jego temat. Jest częścią procesu zapewnienia jakości i można je potraktować jako jedną z wielu czynności w całej serii aktywności mających na celu dostarczenie oprogramowania wysokiej jakości.

Testowanie jest efektywniejsze, jeśli wykonywane jest na wielu poziomach w całym cyklu wytwarzania oprogramowania. W zależności od stosowanej metody testowania, może być ono wdrożone w dowolnym momencie wytwarzania oprogramowania.

3.1. Testowanie jednostkowe

Podstawowym rodzajem testów są testy jednostkowe (ang. *unit tests)*, nazywane także testowaniem programistycznym. Jest to sposób testowania programu, w którym jednostka pracy jest testowana w izolacji od reszty systemu [11]. Pisanie testów jednostkowych pozwala wykryć błędy w najwcześniejszej możliwej fazie wytwarzania oprogramowania, czyli w trakcie pisania kodu programu.

Przy tworzeniu oprogramowania testy pozwalają one upewnić się, że poszczególne części programu działają poprawnie. Aby weryfikacja była wiarygodna, testy jednostkowe powinny testować nie tylko oczekiwany rezultat, ale także nieoczekiwane elementy, takie jak niepoprawne dane lub ich brak.

Testy jednostkowe są szybkie, zatem należy je uruchamiać po każdej zmianie w kodzie źródłowym, co sprawia, że kod testowy jest zawsze zsynchronizowany z kodem produkcyjnym. Dobra struktura testów pozwala na zorientowanie się nie tylko w architekturze i działaniu systemu, ale także intencjach autora danego testu.

Dodatkowo testy jednostkowe ewoluują wraz z kodem aplikacji, więc z czasem tworzą coraz dokładniejszą i coraz lepszą prezentację dla produkcyjnych instrukcji. Testy mogą w dużej mierze zastąpić komentarze w kodzie. Gdy kod jest niezrozumiały, testy mogą stanowić uzupełnienie uzasadniające decyzje podjęte na etapie programowania.

Pokrycie kodu: Termin *pokrycie kodu* (*code coverage*) jest często spotykany w testach jednostkowych. Jest to miara określająca ile procent kodu źródłowego jest wykonywanych podczas wszystkich testów. Optymalną wartością pokrycia jest 70-85%, w zależności od złożoności, logiki biznesowej, technologii itp. Pokrycie poniżej 40% może oznaczać, że dużo błędów występujących w kodzie programu nie zostało wykrytych, co przekłada się w późniejszym czasie na wzrost kosztów naprawy.

3.2. Moduł unittest

Standardowa biblioteka języka Python posiada moduł wspomagający proces testowania [3]. Obecnie nazywa się *unittest*, ale zanim znalazł się w standardowej bibliotece Pythona, był nazywany *PyUnit*. Jego twórcą jest Steve Purcell. Do standardowej biblioteki Pythona po raz pierwszy został dodany w wersji 2.1 i od tego momentu cieszy się dużą popularnością. Jest on podstawowym modułem do tworzenia testów, który bazuje na bibliotece *Junit* dla języka Java [10].

Moduł *unittest* posiada klasę bazową *TestCase*. Zawiera ona zbiór asercji, które pomagają przy weryfikacji wartości wyrażeń i rezultatów wywołań funkcji. Został on stworzony głównie dla testów jednostkowych jednak nie ma przeszkód aby stosować go w implementacji testów np. integracyjnych czy funkcjonalnych.

Aby napisać test przy użyciu modułu *unittest* trzeba wpierw utworzyć klasę rozszerzającą *TestCase*. Moduł *unittest* wymusza konwencje dotyczące nazewnictwa klas oraz metod, podczas tworzenia testów. W nazwach metod klasy testującej należy używać przedrostka "test" przed nazwą jednostki pracy (**def** test_nazwa_metody_testowanej). Klasa testowa powinna zawierać słowo "Tests" oraz nazwę testowanego modułu w odwrotnej kolejności (**class** NazwaModuluTests).

Pomimo tego, że *unittest* jest szeroko wykorzystywany przez społeczność Pythona, to jednak zawiera pewne ograniczenia. Moduł najczęściej jest krytykowany za wymuszenie na programistach rozszerzania klasy *TestCase* oraz stosowania nazw zgodnych z konwencją. Ponadto, sugerowanym rozwiązaniem do wykonywania asercji są metody dziedziczone z klasy bazowej, które nie wyczerpują wszystkich przypadków użycia.

3.3. TDD

Test Driven Development, czyli programowanie sterowane testami, jest to proces wytwarzania oprogramowania, w którym zasadniczą rolę odgrywają zautomatyzowane testy pozwalające na regularne weryfikowanie zachowania kodu. Sposób tego programowania polega na tym, iż najpierw przygotowywany jest test funkcjonalności, a dopiero później należy napisać kod, który ma spełniać założenia testu [9].

Ogólny scenariusz postępowania w przypadku programowania sterowanego testami wygląda następująco:

1. Należy przygotować automatyczny test weryfikujący zachowanie kodu.
2. Test kończy się niepowodzeniem z powodu braku kodu, który ma zostać poddany testowaniu, co sygnalizowane jest przez kolor czerwony w odpowiednim środowisku programistycznym. Stworzony test na tym etapie nawet się nie kompiluje - kompilator informuje o braku poszczególnych elementów testowanego kodu (np. klasy, konstruktora, metod, zmiennych itp.).

3. W tym kroku zostaje uzupełniony brakujący kod aplikacji, która będzie testowana. Kod należy pisać do momentu, w którym test zakończy się prawidłowym wynikiem, symbolizowanym zielonym kolorem.
4. Ostatnim etapem TDD jest refaktoryzacja. Umożliwia ona pozbycie się z kodu duplikatów, takich jak identyczne wyrażenie, które jest obliczane w kilku rożnych miejscach. Podczas tego kroku należy ulepszać strukturę kodu w celu zwiększenia czytelności oraz wydajności, jednak w taki sposób, aby nie została zmieniona jego funkcjonalność. Każda zmiana kodu powinna zostać zweryfikowana poprzez uruchomienie wszystkich automatycznych testów, w celu sprawdzenia czy nie doszło do zmian w jego funkcjonalności.

Filarem TDD są zasady, które pozwalają na pisanie wiarygodnych i łatwych w utrzymaniu testów. Każdy test powinien testować tylko jedną funkcjonalność danego kodu. Dzięki takiej granulacji możliwa jest drobnoziarnista kontrola wszelkich zachowań testowanej funkcjonalności, co pozwala na radykalne ograniczenie liczby błędów.

Po każdej zmianie wprowadzonej do kodu należy uruchamiać wszystkie testy w celu weryfikacji, czy modyfikacja bazy kodu nie spowodowała wprowadzania defektów do wcześniej działającego projektu. Niespełnienie tego warunku, szczególnie podczas procesu refaktoryzacji, może wprowadzać błędy do wcześniej poprawnie działającego oprogramowania.

Nowe testy mogą zostać dodane wtedy i tylko wtedy, gdy wszystkie istniejące testy kończą się sukcesem. Każdy problem z napisaniem testu powinien sugerować rozbicie testu na mniejsze kroki.

Regularnie powinien zostać przeprowadzany całościowy przegląd projektu. Należy sprawdzać, czy wszystkie nazwy i koncepcje pasują do siebie. Nazewnictwo metod, zmiennych, klas i modułów powinno jak najdokładniej określać ich przeznaczenie, dzięki czemu kod pozostaje czytelny i łatwy do utrzymania.

W programowaniu sterowanym testami kardynalną kwestią jest uniezależnienie testu od testowanego kodu aplikacji, czyli napisanie testu w ten sposób, aby nie było konieczne dokonywanie ingerencji w kod, który będzie testowany. Także poszczególne testy powinny być niezależne od siebie. Każde sprzężenie pomiędzy testami może spowodować niedeterministyczne wyniki testów. Klasycznym błędem popełnianym przez programistów jest wymuszenie kolejności testów. W takiej sytuacji jeden test inicjuje dane dla innego testu. Jeżeli zostaną one uruchomione w odwrotnej kolejności spowoduje to, że drugi test zostanie wykonany na niezainicjowanych zmiennych, co prawdopodobnie zakończy się niepowodzeniem. Jest to niepożądana sytuacja, ponieważ testy będą zwracały różne wyniki, co podważy zaufanie do nich.

Podobnie nie należy zbytnio optymalizować kodu testowego poprzez np. współdzielenie zasobów. Porażka jednego testu i nie zwolnienie zasobu może powodować kaskadowe załamanie pozostałych testów.

4. Grafy

Przed wprowadzeniem pojęcia drzewa, należy najpierw zaprezentować grafy. Graf to struktura danych, na którą składają się dwa zbiory: zbiór *wierzchołków* i zbiór *krawędzi* [4]. Graf G zapisywany jest jako uporządkowana para $G = (V, E)$, gdzie $V = \{v_1, v_2, ..., v_n\}$ to zbiór n wierzchołków, a $E = \{e_1, e_2, ..., e_m\}$ to zbiór m krawędzi.

Dane przechowywane są w wierzchołkach, natomiast sposób przemieszczania się po grafie określają krawędzie. Z jednego wierzchołka do innego można przejść wyłącznie wtedy, gdy istnieje krawędź, która je łączy. Jeżeli dwa wierzchołki połączone są ze sobą więcej niż jedną krawędzią, to takie krawędzie nazywane są *krawędziami wielokrotnymi*. Gdy wierzchołek posiada krawędź, która łączy go z samym sobą, to tworzy się wówczas *pętla*. Graf, który zawiera krawędzie wielokrotne lub pętle, nazywany jest *multigrafem*. Graf, który ich nie posiada, jest *grafem prostym*.

Krawędź, którą można przemieszczać się wyłącznie w jednym kierunku, nazywa się *krawędzią skierowaną*, natomiast graf, który zawiera takie krawędzie – *grafem skierowanym* lub *digrafem*. Graf, który nie zawiera krawędzi skierowanych to *graf nieskierowany*.

Graf, którego zbiór krawędzi jest pusty, nazywany jest *grafem pustym*. Nie wyklucza to posiadania wierzchołków. Wierzchołek, który nie jest połączony żadną krawędzią nazywa się *wierzchołkiem izolowanym*.

Gdy z krawędziami grafu są związane dodatkowe wartości liczbowe, to wartości takie nazywane są *wagami*, natomiast graf – *grafem ważonym*. Taki graf zawiera zbiór krawędzi zbudowany z uporządkowanych trójek, gdzie dwa pierwsze elementy są wierzchołkami połączone krawędzią, natomiast trzeci element mówi o wadze tej krawędzi.

Graf planarny to graf, w którym żadne krawędzie nie przecinają się. Graf, w którym każda para wierzchołków łączy się krawędzią, nazywa się *grafem pełnym*.

Wybrane pojęcia dotyczące grafów:

— *Rząd grafu* jest to liczba wierzchołków występujących w grafie.

— *Rozmiar grafu* jest to liczba krawędzi występujących w grafie.

— *Stopień wierzchołka* jest liczba krawędzi, które łączą się z danym wierzchołkiem. Dla grafów skierowanych występuje *stopień wchodzący* – to liczba krawędzi które, wchodzą do wierzchołka, oraz *stopień wychodzący* – liczba krawędzi wychodzących z danego wierzchołka. Stopień wierzchołka izolowanego jest równy 0.

— *Ścieżka* jest to uporządkowany ciąg kolejnych krawędzi (bądź wierzchołków), po których można dotrzeć z wierzchołka startowego do wierzchołka końcowego. Pomiędzy dwoma wierzchołkami może istnieć wiele różnych ścieżek. Długość ścieżki to liczba krawędzi, przez które przechodzi. Naj-

krótsza ścieżka pomiędzy wierzchołkami to ścieżka posiadająca minimalną liczbę wierzchołków pomiędzy tymi dwoma wierzchołkami. Jeśli przez krawędź/wierzchołek przechodzi się tylko raz, wtedy taka ścieżka nazywa się *ścieżką prostą*. Ścieżka prosta, która zawiera wszystkie wierzchołki danego grafu, nazywana jest ścieżką Hamiltona. Ścieżka prosta, która przechodzi przez wszystkie krawędzie grafu, nazywa się ścieżką Eulera.

— *Cykl* jest to ścieżka zamknięta, która zaczyna się i kończy w tym samym wierzchołku. Cykl nazywany jest prostym, jeśli przez każdą krawędź przechodzi się wyłącznie raz. Cykl prosty, który zawiera wszystkie wierzchołki danego grafu nazywa się cyklem Hamiltona. Graf nieposiadający cykli nazywa się grafem acyklicznym.

5. Drzewa

Drzewo jest strukturą danych, która jest spójnym i acyklicznym grafem nieskierowanym [4]. Drzewo, podobnie jak wcześniej opisane grafy, składa się z wierzchołków (w kontekście drzew nazywanych często węzłami), oraz łączących je krawędzi. W odróżnieniu od grafów, jeden z wierzchołków jest wyróżniony i nazywany *korzeniem* drzewa. Każde niepuste drzewo zawiera dokładnie jeden korzeń. Ciąg krawędzi łączących węzły nazywa się *ścieżką*. Drzewo jest grafem spójnym i acyklicznym, zatem istnieje dokładnie jedna ścieżka, która łączy korzeń z każdym innym wierzchołkiem. Ogólniej, dwa dowolne wierzchołki drzewa są połączone dokładnie jedną ścieżką.

Poziomem węzła nazywana jest liczba krawędzi na ścieżce łączącej wskazany wierzchołek z korzeniem. Korzeń drzewa ma zawsze poziom 0. Każdy wierzchołek znajdujący się na niższym poziomie, który jest połączony z wierzchołkiem na wyższym poziomie, nazywamy *potomkiem* lub *dzieckiem*, zaś wierzchołek na wyższym poziomie jest w tej relacji nazywany *przodkiem* lub *rodzicem*. Węzeł może mieć dowolną liczbę dzieci. Węzeł nie posiadający żadnych dzieci nazywany jest *liściem* bądź *węzłem zewnętrznym*. Natomiast każdy węzeł, z wyjątkiem korzenia, posiada dokładnie jednego rodzica. Korzeń jako wierzchołek najwyższego poziomu nie posiada rodzica. Węzły, które posiadają tego samego rodzica, nazywane są *braćmi*.

5.1. Drzewa binarne

Szczególnym rodzajem drzewa jest *drzewo binarne*. Drzewem binarnym (z korzeniem) jest każde drzewo, w którym każdy węzeł posiada co najwyżej dwójkę dzieci. Wyróżnia się lewe i prawe dziecko danego węzła. *Binarne drzewo poszukiwań* (ang. *Binary Search Tree (BST)*) to drzewo binarne, w którym lewe (prawe) poddrzewo każdego węzła zawiera elementy mniejsze (większe lub równe) od elementu w danym węźle [6].

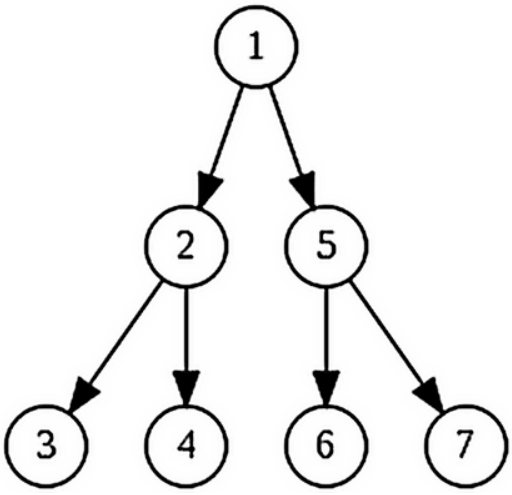
W drzewie BST nowe elementy dodawane są w odpowiednim kierunku (na lewo bądź na prawo) w zależności od wartości dodawanego elementu. Dodawanie elementu rozpoczyna się od korzenia. Gdy wartość dodawanego elementu jest mniejsza, niż wartość znajdująca się w wierzchołku, nowy węzeł dodawany jest do lewego poddrzewa. W przeciwnym wypadku wartość umieszczana jest w prawym poodrzewie. Jeżeli poddrzewo jest puste (brak potomka danego węzła), to dodawany element tworzy nowy liść. Każdy następny dodawany element przechodzi tą samą kierunkową operację i jest umieszczany na końcu drzewa jako nowy liść.

5.2. Operacje na drzewach

Interfejs drzewa umożliwia wykonywanie typowych operacji dla struktur danych, takich jak:
— wyszukiwanie elementu w drzewie,
— dodawanie elementu do drzewa,
— usuwanie elementu z drzewa,
— pobranie wszystkich elementów znajdujących się w drzewie.

Przechodzenie drzewa jest procesem odwiedzenia każdego wierzchołka znajdującego się w drzewie. Węzły drzewa mogą być odwiedzane na co najmniej trzy sposoby [7].

— Metoda *pre-order*, prefiksowa, przechodzenie wzdłużne. W pierwszej kolejności odwiedzony zostaje rodzic, a następnie jego dzieci.



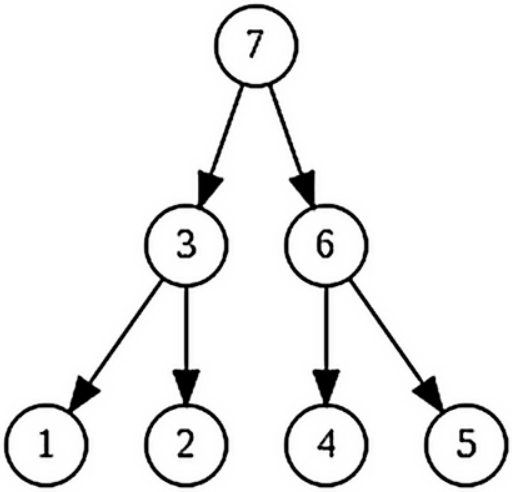
Rysunek 5.1: Przechodzenie drzewa w porządku pre-order.

— Metoda *post-order*, postfiksowa, przechodzenie wsteczne. Jest to sposób odwrotny do przechodzenia prefiksowego – najpierw odwiedzone zostają wszystkie dzieci, a w dalszej kolejności rodzic.
— Metoda *in-order*, infiksowa, przejście poprzeczne. W taki sposób można przechodzić jedynie drzewo binarne. W pierwszej kolejności odwiedza się lewe dziecko, następnie rodzica, a na końcu prawe dziecko.

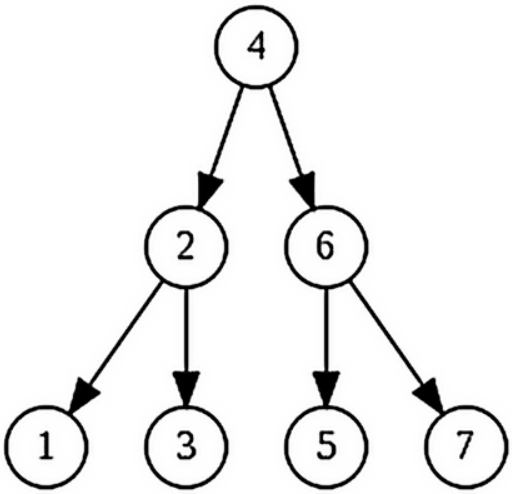
5.3. Reprezentacja na lewo syn, na prawo brat

Drzewa o dowolnym stopniu rozgałęzień można wydajnie reprezentować jako drzewo binarne LCRS, *na lewo syn, na prawo brat* (ang. *left-child, right-sibling binary tree*) [8]. Każdy węzeł drzewa LCRS zawiera dwa łącza, jedno do pierwszego dziecka, drugie do następnego brata. Dzieci danego węzła tworzą listę pojedyńczą.

Reprezentacja LCRS jest bardziej wydajna niż zwykłe m-drzewo, ale kosztem wolniejszego przeszukiwania dzieci danego węzła. Zaleca się korzystać z reprezentacji LCRS, jeżeli (1) chcemy oszczędzać pamieć, (2) nie jest wymagany swobodny dostęp do dzieci danego węzła. Drzewa LCRS stosuje się



Rysunek 5.2: Przechodzenie drzewa w porządku post-order.



Rysunek 5.3: Przechodzenie drzewa w porządku in-order.

np. w strukturach danych działających na stertach (kopiec dwumianowy, kopiec Fibonacciego).

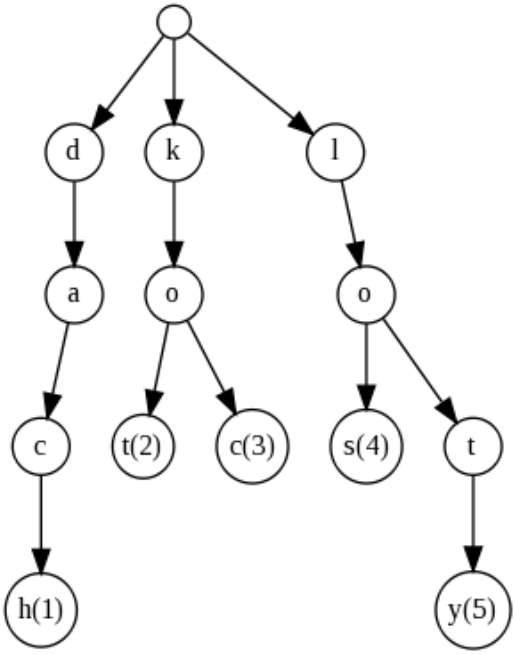
Listing 5.1: Szukanie k-tego dziecka danego węzła w drzewie LCRS.

```
def kth_child(top, k):
    """Szukanie k-tego dziecka danego wezla."""
    if top is None:
        return None
    child = top.left
    while k != 0 and child is not None:
        child = child.right
        k -= 1
    return child    # moze byc None
```

5.4. Drzewa trie

W poprzednim rozdziale zostało opisane drzewo jako abstrakcyjna struktura danych. W praktyce różne implementacje drzew są wykorzystywane w różnych zastosowaniach. Najpopularniejszym rodzajem drzew są binarne drzewa poszukiwań, takie jak drzewa czerwono-czarne, drzewa AVL, czy drzewa splay. Jednakże, jeżeli dodane zostanie założenie o tym, że klucze w drzewie są łańcuchami znaków, to można stworzyć wydajniejsze rozwiązanie korzystające z własności łańcuchów znaków.

Dedykowaną strukturą drzewiastą do obsługi łańcuchów znaków jest drzewo trie [12]. Wspomniane wcześniej drzewa poszukiwań przechowują w węzłach całe klucza. Drzewo trie przechowuje w wierzchołkach jedynie fragmenty kluczy, co pozwala zoptymalizować procedurę wyszukiwania. Ponadto taki sposób reprezentacji kluczy pozwala na wydajne zaimplementowanie dodatkowych metod, takich jak wyszukiwanie wszystkich kluczy z zadanym przedrostkiem.



Rysunek 5.4: Przykładowe drzewo trie, zawierające pary (klucz, wartość): ("dach", 1), ("kot", 2), ("koc", 3), ("los", 4), ("loty", 5).

5.4.1. Reprezentacja węzłów

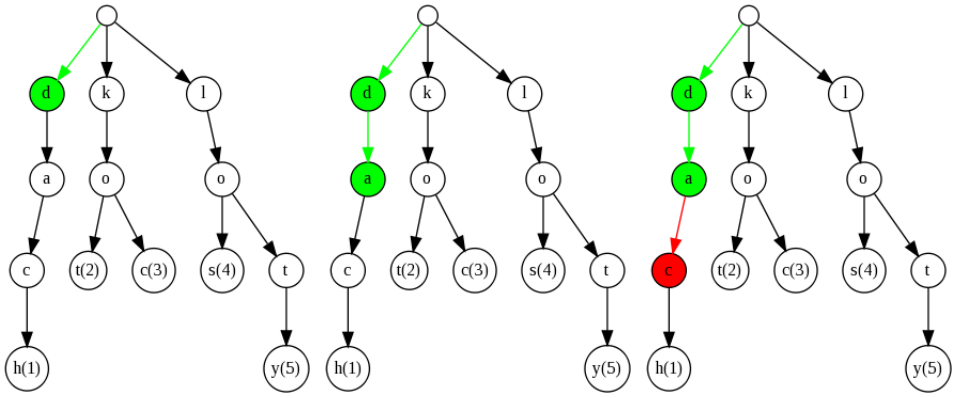
Tak jak zostało to opisane w pierwszym podrozdziale, podstawową cechą odróżniającą drzewo trie od innych drzew jest przechowywanie w wierzchołku jedynie fragmentu klucza [12]. Często implementuje się węzeł drzewa trie nie jako fragment klucza, ale jako tablicę zawierającą wszystkie obsługiwane znaki. Taka implementacja pozwala na bardzo wydajne wyszukiwanie, ale sprawia, że w drzewie można przechowywać jedynie klucze składające się ze znaków predefiniowanych przez autora.

5.4.2. Przeszukiwanie drzewa trie

Podstawową operacją w drzewie trie jest wyszukanie wartości pasującej do zadanego klucza. W klasycznych drzewach poszukiwań binarnych, na każdym etapie procesu, poszukiwany klucz jest porównywany z kluczem przypisanym do tego węzła, a następnie analogiczna procedura wykonywana jest w lewym bądź w prawym dziecku, w zależności od wyniku operacji porównania. Kolejne kroki są wykonywane do momentu odnalezienia poszukiwanego węzła, bądź do osiągnięcia pustego wskaźnika symbolizującego, że wierzchołek z zadanym kluczem nie znajduje się w drzewie. Odnalezienie klucza wymaga zatem $O(h)$ operacji porównania, gdzie h oznacza głębokość drzewa.

Porównanie łańcuchów znaków może być szybkie, jeżeli porównywane łańcuchy nie mają wspólnego przedrostka, aczkolwiek może być również kosztowne, jeżeli taki przedrostek istnieje. Złożoność takiej operacji wynosi $O(s)$, gdzie s jest długością poszukiwanego klucza. Zatem w najgorszym przypadku procedura wyszukiwania może zająć czas $O(sh)$, a w najlepszym czas $O(h)$.

W drzewie trie proces poszukiwania w najgorszym przypadku przebiega znacznie szybciej. Na każdym etapie porównywany jest jedynie fragment zadanego klucza z fragmentem klucza umieszczonego w kolejnych węzłach. Jest to operacja mniej kosztowna, niż porównywanie całego łańcucha. W najgorszym przypadku takich porównań należy wykonać tyle, ile znaków posiada zadany łańcuch [czas $O(s)$], w najlepszym proces zostanie zakończony po pierwszym porównaniu [czas $O(1)$].



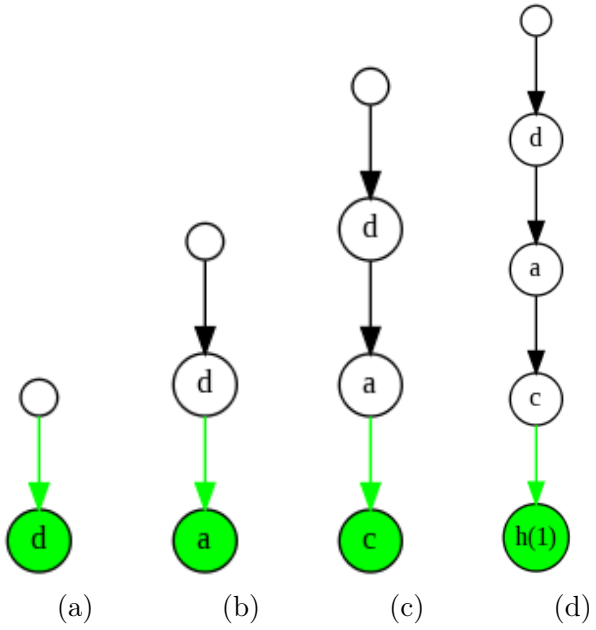
Rysunek 5.5: Szukanie klucza "dar" w drzewie trie.

5.4.3. Wstawianie wartości

Wstawianie wartości do drzewa trie poprzedzone jest procedurą wyszukiwania najdłuższego przedrostka wstawianego klucza już znajdującego się w drzewie. Procedura zakończy się po p porównaniach (długość najdłuższego przedrostka wstawianego klucza znajdującego się w drzewie), gdzie $0 \leqslant p \leqslant s$. Następnie należy dodać $s - p$ brakujących węzłów.

W typowej sytuacji wyszukiwany zostaje najdłuższy wspólny przedrostek, a następnie zostanie dodany brakujący węzeł. W skrajnych przypadkach, je-

żeli długość najdłuższego przedrostka wynosi 0, to wtedy należy dodać s węzłów. Tą sytuację obrazuje poniższa sekwencja rysunków. Podczas wstawiania klucza do pustego drzewa należy tworzyć kolejne węzły zawierające fragmenty klucza, które zazwyczaj są pojedynczymi literami dodawanego klucza. Jeżeli klucz już istnieje w drzewie, należy wykonać s operacji porównania, aczkolwiek dodawanie nowych węzłów nie będzie konieczne. Zatem procedura wstawiania do drzewa również jest liniowa $O(s)$.

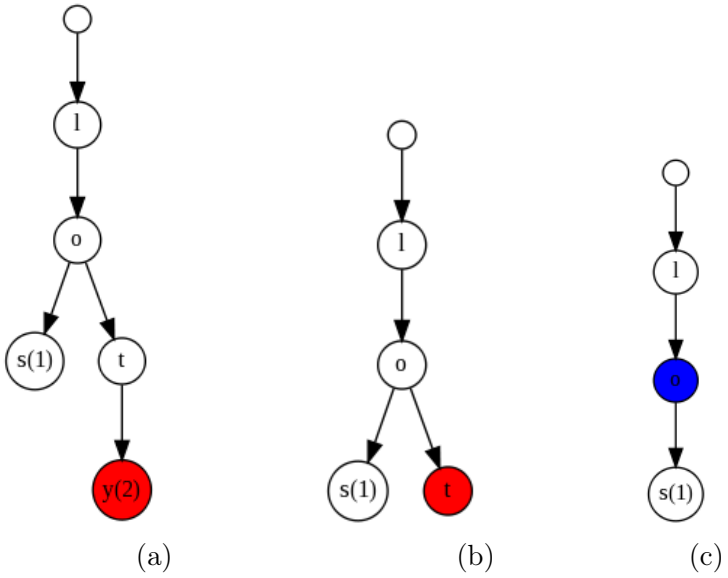


Rysunek 5.6: Dodawanie pary ("dach", 1) do pustego drzewa trie.

5.4.4. Usuwanie wartości

Usuwanie wartości z drzewa trie również poprzedzone jest procedurą wyszukiwania. W najprostszym przypadku, jeżeli klucz nie znajduje się w drzewie, nic nie zostanie usunięte. Implementacja tej operacji zakładająca idempotentność zakończy metodę sukcesem. Inna implementacja może rzucić wyjątek sygnalizujący, że klucz nie został odnaleziony.

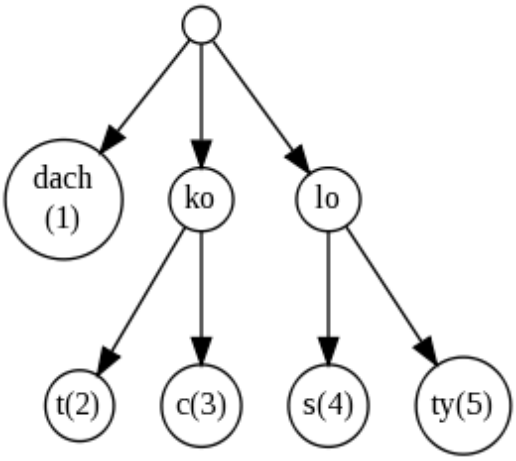
Jeżeli klucz znajduje się w drzewie to należy go usunąć. W najprostszym przypadku, po odnalezieniu ostatniego węzła, należy z niego usunąć wartość. Takie rozwiązanie posiada złożoność liniową $O(s)$. W celu oszczędzania pamięci można usunąć zbędne węzły. Jeżeli węzeł nie posiada potomków oraz braci, to taki węzeł może zostać usunięty. Po jego usunięciu może się okazać, że taka sama sytuacja występuje u rodzica ostatniego węzła, zatem operację można powtarzać dopóki węzły spełniają warunek nie posiadania potomka oraz rodzeństwa. Podczas tej operacji zostanie usuniętych co najwyżej tyle węzłów, ile znaków posiadał usuwany klucz, więc asymptotyczna złożoność operacji pozostaje liniowa $O(s)$.



Rysunek 5.7: Usuwanie klucza "loty" z drzewa trie.

5.5. Drzewa Patricia

Drzewo Patricia to sposób implementacji drzew trie. W literaturze zwany jest także jako skompresowane drzewo trie. Nazwa tego drzewa to akronim od *Practical Algorithm to Retrieve Information Coded in Alphanumeric*, co można przetłumaczyć jako praktyczny algorytm pozyskiwania informacji zakodowanych alfanumerycznie. Od tradycyjnego drzewa trie rożni się informacjami przechowywanymi w węźle. Klasyczne drzewo trie w każdym wierzchołku przechowuje jedynie jeden znak klucza. Drzewo Patricia w węźle może przechowywać więcej niż jeden znak. Pozwala to na zredukowanie liczby węzłów, o ile nie posiadają one wartości lub braci.



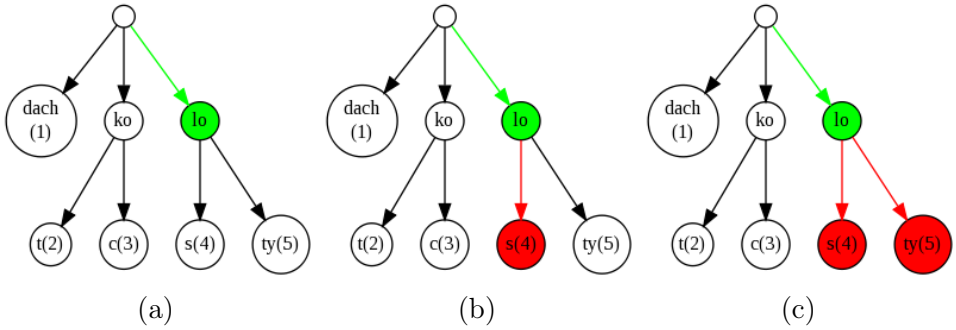
Rysunek 5.8: Przykładowe drzewo Patricia.

5.5.1. Reprezentacja węzła

Węzeł w drzewie Patricia reprezentowany jest przez podobną strukturę, jak w zwykłym drzewie trie. Jedyna różnica polega na przechowywaniu większego fragmentu klucza. W szczególności możliwe jest przechowywanie całego klucza w jednym węźle, jeżeli dany klucz nie posiada wspólnego przedrostka z żadnym innym kluczem przechowywanym w drzewie. Możliwa jest również sytuacja odwrotna. Jeżeli wstawione klucze stworzą strukturę, w której każdy węzeł będzie posiadał wartość lub brata, to takie drzewo Patricia będzie takie samo jak nieskompresowane drzewo trie. Stąd można wywnioskować, że drzewo Patricia warto stosować jeżeli liczba przechowywanych kluczy jest o wiele mniejsza od przestrzeni wszystkich możliwych kluczy.

5.5.2. Przeszukiwanie drzewa Patricia

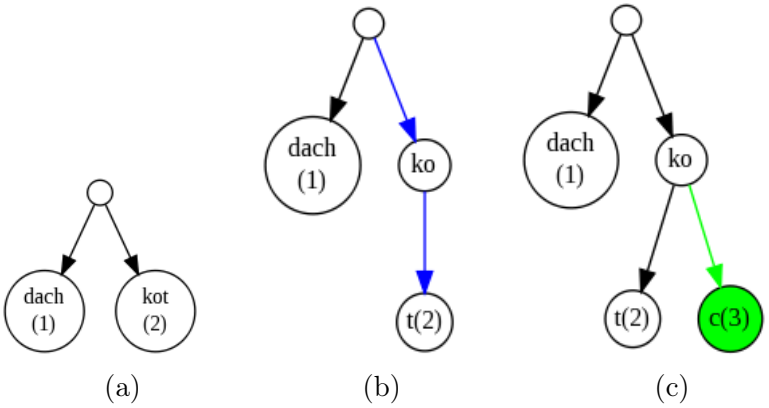
Wyszukiwanie klucza w drzewie Patricia przebiega w podobny sposób jak w nieskompresowanym drzewie trie. Co najwyżej w ramach jednego węzła może zostać porównany dłuższy fragment klucza. W dalszym ciągu zachowana jest asymptotyczna złożoność liniowa oraz stała w optymistycznym przypadku, ponieważ w ramach pojedyńczego wyszukiwania klucza liczba porównań znaków będzie co najwyżej równa liczbie znaków w poszukiwanym kluczu.



Rysunek 5.9: Szukanie klucza "lok" w drzewie Patricia.

5.5.3. Wstawianie wartości

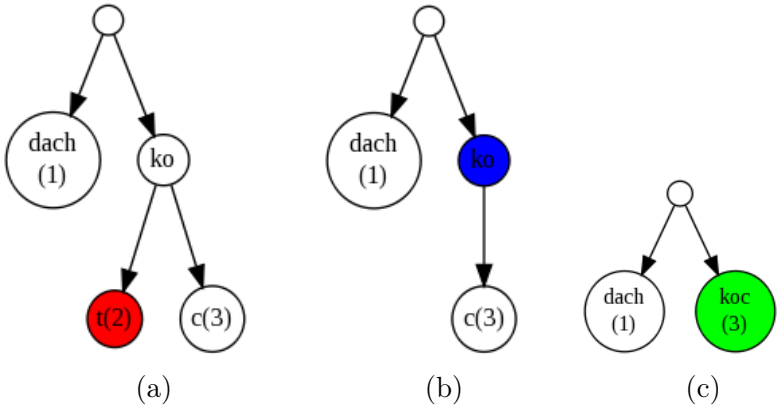
Wstawianie wartości w drzewie Patricia jest procedurą mniej złożoną niż analogiczna procedura w nieskompresowanym drzewie trie. Po wyszukaniu najdłuższego przedrostka wstawianego klucza istniejącego w drzewie należy stworzyć nowy węzeł potomny zawierający pozostałą część wstawianego klucza. Jeżeli fragment klucza ostatniego węzła zawierającego przedrostek wstawianego klucza jest dłuższy niż odnaleziony najdłuższy wspólny przedrostek, to należy dokonać procedury podziału tego węzła na dwa węzły z krótszymi fragmentami klucza. Pierwszy z nich powinien zawierać wspólną część, a drugi pozostałą część fragmentu klucza podzielonego węzła.



Rysunek 5.10: Dodanie pary ("koc", 3) do drzewa Patricia.

5.5.4. Usuwanie wartości

Procedura usuwania klucz w drzewie Patricia również jest prostsza niż taka operacja w zwykłym drzewie trie. Po odnalezieniu węzła zawierającego klucz, wystarczy usunąć jego wartość, a jeżeli nie posiada potomków można usunąć cały węzeł. Po przeprowadzeniu operacji może okazać się, że rodzic usuwanego węzła może zostać scalony ze swoim dzieckiem. Powoduje to skompresowanie drzewa i zaoszczędzenie pamięci. Z definicji węzła w drzewie Patricia wynika, że operacja scalenie będzie mogła zostać wykonana jedynie na bezpośrednim rodzicu usuwanego węzła. Zatem złożoność usuwania klucza, podobnie jak w nieskompresowanym drzewie trie, można oszacować na $O(s)$.



Rysunek 5.11: Usuwanie klucza "kot" z drzewa Patricia.

6. Implementacja drzew

W rozdziale opisano implementację drzew trie i drzew Patricia w języku Python.

6.1. Implementacja drzewa trie

Drzewo trie w załączonym do tej pracy kodzie źródłowym zostało zaimplementowane w języku Python. W standardowej bibliotece tego języka istnieje słownik, który jest strukturą przechowującą pary (klucz, wartość). Język Python rozpoznaje typ za pomocą refleksji, determinując go na podstawie interfejsu obiektu. Zatem każdy obiekt posiadający ten sam interfejs co słownik może być z nim używany wymiennie. Implementacja opisana w tej pracy korzysta z tej właściwości.

6.1.1. Klasa MutableMapping

Moduł collection języka Python zawiera zestaw abstrakcyjnych klas bazowych, które ułatwiają utrzymanie spójnego interfejsu ze słownikiem. Tabela 6.1 zestawia abstrakcyjne klasy użyteczne podczas implementowania słownika [3].

Rekomendowanym sposobem implementacji słownika jest rozszerzanie klasy MutableMapping z modułu collections . Zaimplementowanie pięciu abstrakcyjnych metod pozwala otrzymać typ będący słownikiem. Pierwszym krokiem jest więc przygotowanie klasy podrzędnej (listing 6.1).

Listing 6.1: Przykładowa klasa implementująca MutableMapping.

```
from collections import MutableMapping

class MutableMappingImplementation(MutableMapping):

    def __setitem__(self, key, value): pass

    def __delitem__(self, key): pass

    def __getitem__(self, key): pass

    def __len__(self): pass

    def __iter__(self): pass
```

Tabela 6.1: Wybrane klasy abstrakcyjne.

Klasa	Klasy nadrzędne	Abstrakcyjne metody	Dostarczane metody
Container		__contains__	
Iterable		__iter__	
Iterator	Iterable	**next**	__iter__
Sized		__len__	
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	pop, popitem, clear, update, setdefault

6.1.2. Węzeł drzewa trie

Drzewa korzystające z reprezentacji *na lewo syn, na prawo brat* składają się z węzłów, zatem przed podjęciem dalszych kroków należy przygotować strukturę reprezentującą węzeł. Tak jak zostało to opisane w części teoretycznej powinien on zawierać fragment klucza, wartość, oraz referencje do syna, brata i rodzica (listing 6.2).

Listing 6.2: Klasa Node reprezentująca węzeł drzewa.

```
class Node(object):

    def __init__(self, partial_key, parent=None, right=None,
                 left=None, value=None):
        self.partial_key = partial_key
        self.parent = parent
        self.right = right
        self.left = left
        self.value = value
```

Przydatne mogą być również proste metody pozwalające stworzyć czytelniejszy interfejs (listing 6.3).

Listing 6.3: Pomocnicze metody klasy Node.

```
    def has_child(self):
        return self.left is not None

    def has_brother(self):
        return self.right is not None

    def has_value(self):
```

```
        return self.value is not None

    def is_root(self):
        return self.parent is None
```

Węzeł przechowuje jedynie fragment klucza, zatem należy zaimplementować metodę, która umożliwi zrekonstruowanie całego klucza. Fragmenty klucza tworzą klucz jeżeli węzeł przechowuje wartość, w przeciwnym wypadku kolejne wartości są jedynie przedrostkiem jakiegoś klucza. W związku z tym, jeżeli węzeł nie posiada wartości, to nie posiada klucza, co zostało zaimplementowane za pomocą zwracania wartości None. Sam proces konstrukcji klucza polega na odwiedzaniu kolejnych wierzchołków aż do osiągnięcia korzenia i połączenia ze sobą wszystkich fragmentów kluczy w pełen klucz.

Listing 6.4: Rekonstrukcja klucza.

```
    def get_key(self):
        if self.has_value():
            node = self.parent
            key = self.partial_key
            while not node.is_root():
                key = node.partial_key + key
                node = node.parent
            return key
        else:
            return None
```

Ostatnie dwie metody węzła umożliwiają pobranie brata i dziecka o zadanym fragmencie klucza. Wyszukiwanie brata jest abstrakcyjną metodą, ponieważ tą operację można przeprowadzić na kilka sposobów opisanych w dalszej części tego rozdziału. Pobieranie dziecka niezależnie od implementacji wygląda tak samo. Jeżeli węzeł w ogóle posiada potomka, to może być to węzeł bezpośrednio połączony z rodzicem lub któryś z jego braci.

Listing 6.5: Pobieranie dziecka węzła.

```
    @abstractmethod
    def get_brother(self, partial_key):
        pass

    def get_child(self, partial_key):
        if not self.has_child():
            return None
        elif self.left.partial_key == partial_key:
            return self.left
        else:
            return self.left.get_brother(partial_key)
```

6.1.3. Iterator dla drzewa trie

Rozszerzanie klasy MutableMapping wymusza stworzenie iteratora. Zgodnie z dokumentacją każdy iterator powinien implementować metodę __iter__() oraz **next**(). Implementacja pierwszej z nich jest typowa.

Iterowanie po drzewie trie może przypominać przeszukiwanie grafu wszerz. Podobnie jak w algorytmie BFS używana jest kolejka do której wstawiane są wszystkie dzieci wierzchołka. Konstrukcja drzewa sprawia, że nie ma potrzeby oznaczania odwiedzanych wierzchołków. Żaden węzeł nie zostanie dodany do kolejki dwa razy, ponieważ wierzchołek nie może być dzieckiem więcej niż jednego rodzica.

Słownik w Pythonie umożliwia iterowanie po kluczach, zatem metoda **next**() powinna zwracać jedynie węzły, które zawierają ostatni fragment klucza, czyli te, które posiadają wartość. Kontrakt pythonowego iteratora wymusza, aby metoda **next**() rzucała wyjątek StopIteration w sytuacji, w której iterator odwiedził już wszystkie wierzchołki. Analogicznie jak podczas przeszukiwania grafu wszerz taka sytuacja następuje, gdy kolejka nie zawiera już więcej węzłów.

Listing 6.6: Iterator dla drzewa trie.

```
class TrieIterator(object):

    def __init__(self, node):
        self.queue = Queue()
        if node is not None:
            self.queue.put(node)

    def __iter__(self):
        return self

    def __next__(self):
        if self.queue.empty():
            raise StopIteration
        else:
            node = self._get_node_and_populate_children()
            while not node.has_value():
                node = self._get_node_and_populate_children()
            return node.get_key()

    next = __next__    # kompatybilnosc z Pythonem 3

    def _get_node_and_populate_children(self):
        node = self.queue.get()
        if node.has_child():
            child = node.left
            self.queue.put(child)
            while child.has_brother():
                child = child.right
                self.queue.put(child)
        return node
```

6.1.4. Abstrakcyjne drzewo trie

Struktura rozszerzająca MutableMapping musi między innymi zwracać liczbę kluczy umieszczonych w kolekcji. Można osiągnąć stałą złożoność tej operacji przechowując liczbę dodanych kluczy (atrybut size). Następnie podczas

dodawania i usuwania elementów należy odpowiednio inkrementować i dekrementować licznik.

Listing 6.7: Klasa Trie.

```
class Trie(collections.MutableMapping):

    def __init__(self):
        self.root = self._make_node('')
        self.size = 0

    def __len__(self):
        return self.size
```

Operacja wstawiania zostanie zaimplementowana w konkretnych implementacjach. Należy zauważyć, że wstawianie do drzewa nowej pary (klucz, wartość) nie powoduje zwiększenia liczby kluczy w drzewie, jeżeli jest to tylko aktualizacja wartości związanej z kluczem.

Listing 6.8: Dodawanie nowego elementu do drzewa trie.

```
    def __setitem__(self, key, value):
        is_new_key_added = self._put(self.root, key, value)
        if is_new_key_added:
            self.size += 1
```

Wyszukiwanie elementu zgodnie z pythonową konwencją powinno rzucać wyjątkiem KeyError w przypadku porażki, czyli braku klucza. To wymaganie zostało umieszczone w metodzie _get_node_by_key(), aczkolwiek sama procedura wyszukiwania zostanie oddelegowana do konkretnej implementacji.

Listing 6.9: Wyszukiwanie klucza w drzewie trie.

```
    def __getitem__(self, key):
        matched_node = self._get_node_by_key(self.root, key)
        return matched_node.value

    def _get_node_by_key(self, node, key):
        matched_node = self._get_node_by_prefix(node, key)
        if matched_node is None or not matched_node.has_value():
            raise KeyError(key)
        else:
            return matched_node
```

Usuwanie klucza z drzewa trie poprzedzone jest wyszukaniem elementu do usunięcia. Wykorzystana do tego celu została metoda służąca do pobierania klucza, zatem w przypadku jej porażki również zostanie rzucony wyjątek KeyError. Sama procedura usuwania może przebiegać dwojako. Jeżeli klucz posiada dziecko wystarczy tylko usunąć wartość z węzła. W przeciwnym wypadku należy usunąć cały węzeł z drzewa. Procedura jest tożsama z usuwaniem elementu z listy wiązanej. Na koniec w celu oszczędzania pamięci można przeprowadzić procedurę pozbywania się z drzewa zbędnych węzłów.

Można usunąć każdy węzeł na drodze do korzenia, który nie posiada dzieci i nie przechowuje wartości.

Listing 6.10: Usuwanie klucza z drzewa trie.

```
    def __delitem__(self, key):
        node_to_remove = self._get_node_by_key(self.root, key)
        if node_to_remove.has_child():
            node_to_remove.value = None
        else:
            self._remove_orphaned_nodes(node_to_remove)
        self.size -= 1

    def _remove_orphaned_nodes(self, node_to_remove):
        parent = node_to_remove.parent
        self._remove_node(node_to_remove)

        if (not parent.is_root() and not parent.has_child()
                and not parent.has_value()):
            self._remove_orphaned_nodes(parent)

    @staticmethod
    def _remove_node(node_to_remove):
        parent = node_to_remove.parent
        if parent.left == node_to_remove:
            parent.left = node_to_remove.right
        else:
            node = parent.left
            while node.right != node_to_remove:
                node = node.right
            node.right = node_to_remove.right
```

Ostatnia metoda, która jest wymagana przez kontrakt klasy MutableMapping pozwala na pobranie iteratora. Cała jego logika została umieszczona w odrębnej klasie, zatem wystarczy jedynie utworzyć nowy obiekt, którego pierwszym węzłem zostanie korzeń.

Listing 6.11: Pobieranie iteratora drzewa trie.

```
    def __iter__(self):
        return TrieIterator(self.root)
```

Mając przygotowane wszystkie te metody bardzo łatwo można przygotować metodę, która zwróci wszystkie klucze rozpoczynające się zadanym przedrostkiem. Wystarczy jedynie znaleźć odpowiedni węzeł i skorzystać z iteratora rozpoczynając przechodzenie drzewa od wierzchołka z ostatnim znakiem przedrostka.

Listing 6.12: Pobieranie kluczy z zadanym przedrostkiem.

```
    def keys_with_prefix(self, prefix):
        node_with_prefix = self._get_node_by_prefix(
            self.root, prefix)
        return TrieIterator(node_with_prefix)
```

Ostatnia dostarczana metoda pozwalająca wyszukać najdłuższy klucz będący przedrostkiem zadanego klucza będzie różnie implementowana w zależności od konkretnej klasy. W abstrakcyjnej klasie można jedynie zająć się obsługą pustej wartości.

Listing 6.13: Pobieranie najdłuższego klucza będącego przedrostkiem zadanego klucza.

```
    def longest_prefix_of(self, key):
        longest_prefix_of_key = self._get_longest_prefix_of(
            self.root, key)
        if longest_prefix_of_key is None:
            return ''
        else:
            return longest_prefix_of_key
```

6.1.5. Implementacja rekurencyjna

Abstrakcyjny węzeł zaprezentowany w poprzednim rozdziale posiadał jedną metodę abstrakcyjną służącą do wyszukiwania brata wierzchołka z określonym fragmentem klucza. Metoda w wersji rekurencyjnej zwraca None, jeżeli węzeł w ogóle nie posiada brata, zwraca bezpośrednio połączonego brata, jeżeli zawiera on poszukiwany fragment klucza, albo wywołuje się rekurencyjnie przechodząc po kolejnych braciach.

Listing 6.14: Rekurencyjny węzeł drzewa trie.

```
    def get_brother(self, partial_key):
        if not self.has_brother():
            return None
        elif self.right.partial_key == partial_key:
            return self.right
        else:
            return self.right.get_brother(partial_key)
```

Dodawanie węzła do rekurencyjnego drzewa rozpoczyna się od rekurencyjnego znalezienia odpowiedniego węzła. Na koniec przesuwając się po pozostałych znakach klucza rekurencyjnie tworzone są kolejne węzły.

Listing 6.15: Rekurencyjne dodawanie węzła.

```
    def _put(self, node, key, value, index=0):
        if len(key) == index:
            is_update = node.has_value()
            node.value = value
            return not is_update
        else:
            partial_key = key[index]
            matched_child = node.get_child(partial_key)
            if matched_child is None:
                matched_child = self._make_node(
                    partial_key, parent=node, right=node.left)
```

```
            node.left = matched_child
        next_index = index + 1
        return self._put(matched_child, key, value, next_index)
```

Rekurencyjne wyszukiwanie węzła przechodzi kolejne wierzchołki poszukując potomka zawierającego kolejne fragmenty klucza. Procedura jest powtarzana do momentu, w którym dziecko z poszukiwanym fragmentem klucza nie istnieje lub do momentu odnalezienia poszukiwanego węzła.

Listing 6.16: Rekurencyjne wyszukiwanie węzła.

```
def _get_node_by_prefix(self, node, prefix, index=0):
    if len(prefix) == index:
        return node
    partial_key = prefix[index]
    child = node.get_child(partial_key)
    next_index = index + 1
    if child is None:
        return None
    else:
        return self._get_node_by_prefix(
            child, prefix, next_index)
```

Ostatnia metoda interfejsu drzewa trie umożliwia odnalezienie najdłuższego klucza będący przedrostkiem zadanego klucza. Jej rekurencyjna wersja poszukuje coraz dłuższy fragmentów klucza. Jeżeli po zadanym fragmentem znajduje się klucz to metoda zwraca tą wartość, w przeciwnym wypadku zwracana jest wartość None. Implementacje upraszcza kontrakt metody get_key(), która zwraca klucz wtedy i tylko wtedy, gdy węzeł posiada wartość.

Listing 6.17: Rekurencyjne wyszukiwanie węzła.

```
def _get_longest_prefix_of(self, node, key, index=0):
    if len(key) == index:
        return node.get_key()
    partial_key = key[index]
    child = node.get_child(partial_key)
    next_index = index + 1
    if child is not None:
        longest_prefix_of_key = self._get_longest_prefix_of(
            child, key, next_index)
        if longest_prefix_of_key is not None:
            return longest_prefix_of_key
    return node.get_key()
```

6.1.6. Implementacja iteracyjna

Każdą z powyższych rekurencji można również zapisać w postaci iteracyjnej. Kod zapisany w formie iteracji zamiast rekurencji wykorzystuje pętle, co pozwala na zapisanie kodu w bardziej zwięzły sposób.

Listing 6.18: Iteracyjny węzeł drzewa trie.

```
    def get_brother(self, partial_key):
        node = self
        while node.has_brother():
            node = node.right
            if node.partial_key == partial_key:
                return node
        return None
```

Listing 6.19: Iteracyjne drzewo trie.

```
class IterativeTrie(Trie):

    def _put(self, node, key, value):
        for partial_key in key:
            next_node = node.get_child(partial_key)
            if next_node is None:
                next_node = self._make_node(
                    partial_key, parent=node, right=node.left)
                node.left = next_node
            node = next_node
        is_update = node.has_value()
        node.value = value
        return not is_update

    def _get_node_by_prefix(self, node, key):
        for partial_key in key:
            node = node.get_child(partial_key)
            if node is None:
                return None
        return node

    def _get_longest_prefix_of(self, node, key):
        longest_prefix_of_key = ''
        for partial_key in key:
            node = node.get_child(partial_key)
            if node is None:
                return longest_prefix_of_key
            if node.has_value():
                longest_prefix_of_key = node.get_key()
        return longest_prefix_of_key
```

6.2. Implementacja drzewa Patricia

Drzewo Patricia różni się od dotychczas opisanych implementacji już na poziomie węzła. Obok metod do pobierania dziecka i brata węzeł drzewa Patricia posiada także metody je pobierające na podstawie przedrostka fragmentu klucza, którym zazwyczaj jest jedna litera. Pomocna jest tu metoda stringów **str**.startswith().

Listing 6.20: Węzeł drzewa Patricia.

```
    def get_child_starts_with(self, partial_key):
        if not self.has_child():
            return None
        elif self.left.partial_key.startswith(partial_key):
            return self.left
        else:
            return self.left.get_brother_starts_with(partial_key)

    def get_brother_starts_with(self, partial_key):
        if not self.has_brother():
            return None
        elif self.right.partial_key.startswith(partial_key):
            return self.right
        else:
            return self.right.get_brother_starts_with(partial_key)
```

Pierwszym krokiem operacji wstawiania jest wyszukanie dziecka korzenia, którego fragment klucza rozpoczyna się od pierwszego znaku wstawianego klucza. Jeżeli okaże się, że klucz dziecka jest równy ze wstawianym kluczem, to należy jedynie zaktualizować wartość. Następnie należy obliczyć najdłuższy wspólny przedrostek. Jeżeli obliczony przedrostek jest równy fragmentowi klucza dopasowanego dziecka, to należy na nim rekurencyjnie wywołać procedurę wstawiania z indeksem przesuniętym o długość przedrostka. Przedrostek krótszy niż fragment klucza potomka oznacza konieczność podziału potomka na dwa mniejsze węzły. Operację podziału rozpoczyna tymczasowe usunięcie dzielonego węzła z drzewa. Następnie w jego miejsce wstawiany jest nowy węzeł, który zawiera jedynie wspólną część fragmentu klucza, a wcześniej usunięty węzeł, po usunięciu z fragmentu klucza wspólnej części, zostaje jego dzieckiem. Następnie, jeżeli pozostała część klucza jest równa fragmentowi klucza w nowym węźle powstałym podczas podziału, należy przypisać temu węzłowi wstawianą wartość. W przeciwnym wypadku do wspomnianego węzła należy dodać kolejny, który będzie zawierał pozostałą część nowo wstawianego klucza. W najprostszym wariancie, jeżeli dziecko zawierające pierwszy znak nie istnieje w drzewie to wystarczy jedynie dodać węzeł zawierający klucz.

Listing 6.21: Dodawanie węzła do drzewa Patricia.

```
    def _put(self, node, key, value, index=0):
        partial_key = key[index]
        remaining_part_of_key = key[index:]
        matched_child = node.get_child_starts_with(partial_key)

        if matched_child is None:
            node.left = self._make_node(remaining_part_of_key,
                parent=node, value=value, right=node.left)
        else:
            if matched_child.partial_key == remaining_part_of_key:
                is_update = matched_child.has_value()
                matched_child.value = value
                return not is_update
```

```
            longest_common_prefix = os.path.commonprefix(
                [remaining_part_of_key, matched_child.partial_key])

            if longest_common_prefix == matched_child.partial_key:
                next_index = index + len(longest_common_prefix)
                self._put(matched_child, key, value, next_index)
            else:
                self._split_node(matched_child, longest_common_prefix,
                                 remaining_part_of_key, value)
        return True

    def _split_node(self, node_to_be_split, longest_common_prefix,
                    remaining_part_of_key, value):
        self._remove_node(node_to_be_split)
        parent = node_to_be_split.parent

        node_with_longest_common_prefix = self._make_node(
            longest_common_prefix,
            parent=parent,
            left=node_to_be_split,
            right=parent.left)
        parent.left = node_with_longest_common_prefix

        common_prefix_length = len(longest_common_prefix)
        node_to_be_split.parent = node_with_longest_common_prefix
        node_to_be_split.partial_key = node_to_be_split.partial_key[
                                       common_prefix_length:]

        if common_prefix_length == len(remaining_part_of_key):
            node_with_longest_common_prefix.value = value
        else:
            node_to_be_split.right = self._make_node(
                remaining_part_of_key[common_prefix_length:],
                parent=node_with_longest_common_prefix,
                value=value)
```

Procedura wyszukiwania klucza oraz wyszukiwania najdłuższego klucza będącego przedrostkiem zadanego klucza przebiega w sposób analogiczny do wyszukiwania węzła podczas dodawania nowego klucza.

Listing 6.22: Wyszukiwanie klucza w drzewie Patricia.

```
    def _get_node_by_prefix(self, node, prefix, index=0):
        if len(prefix) == index:
            return node
        partial_key = prefix[index]
        matched_child = node.get_child_starts_with(partial_key)
        if matched_child is None:
            return None
        else:
            remaining_part_of_key = prefix[index:]
            longest_common_prefix = os.path.commonprefix(
                [remaining_part_of_key, matched_child.partial_key])
            if longest_common_prefix == prefix:
                return matched_child
            elif longest_common_prefix == matched_child.partial_key:
```

```
                next_index = index + len(longest_common_prefix)
                return self._get_node_by_prefix(
                    matched_child, prefix, next_index)
            else:
                return None

    def _get_longest_prefix_of(self, node, key, index=0):
        if len(key) == index:
            return node.get_key()
        partial_key = key[index]
        matched_child = node.get_child_starts_with(partial_key)
        if matched_child is not None:
            remaining_part_of_key = key[index:]
            longest_common_prefix = os.path.commonprefix(
                [remaining_part_of_key, matched_child.partial_key])
            if longest_common_prefix == key:
                return matched_child.get_key()
            if longest_common_prefix == matched_child.partial_key:
                next_index = index + len(longest_common_prefix)
                longest_prefix_of_key = self._get_longest_prefix_of(
                    matched_child, key, next_index)
                if longest_prefix_of_key is not None:
                    return longest_prefix_of_key
        return node.get_key()
```

W drzewie Patricia czyszczenia drzewa różni się od sposobu zaimplementowanego w abstrakcyjnym drzewie trie. W tym rodzaju drzewa może istnieć konieczność połączenia węzłów, jeżeli ich dalszy podział nie jest potrzebny. Taka sytuacja występuje, jeżeli usuwany węzeł ma dokładnie jednego brata, a jego rodzic nie posiada wartości.

Listing 6.23: Usuwanie zbędnych węzłów z drzewa Patricia.

```
    def __delitem__(self, key):
        node_to_remove = self._get_node_by_key(self.root, key)

        if node_to_remove.has_child():
            node_to_remove.value = None
            self._merge_with_child_if_possible(node_to_remove)
        else:
            self._remove_node(node_to_remove)
            self._merge_with_child_if_possible(node_to_remove.parent)
        self.size -= 1

    def _merge_with_child_if_possible(self, node):
        directly_connected_child = node.left

        if (not node.has_value()
                and directly_connected_child is not None
                and not directly_connected_child.has_brother()):
            self._remove_node(node)
            parent = node.parent
            directly_connected_child.parent = parent
            directly_connected_child.partial_key = (
                node.partial_key + directly_connected_child.partial_key)
            directly_connected_child.right = parent.left
```

```
parent.left = directly_connected_child
```

6.3. Przykłady użycia drzew trie

Przykładowa sesja interaktywna z listingu 6.24 przedstawia korzystanie z kodu drzew trie. Kod działa dla Pythona 2 i dla Pythona 3, choć mogą pojawić się różnice w zwracanych obiektach. W przypadku słowników Python 3 korzysta z widoków, Python 2 zwraca listy, więc zachowanie kodu dla drzew trie będzie analogiczne.

Listing 6.24: Sesja interaktywna.

```
>>> from recursive_trie import RecursiveTrie as Trie
# Podobnie dla
# from iterative_trie import IterativeTrie as Trie
# from recursive_patricia_trie import RecursivePatriciaTrie as Trie
>>> trie = Trie()
>>> trie['dach'] = 1
>>> trie['dach']
1
>>> trie['dach'] = 2
>>> trie['dach']
2
>>> del trie['dach']
>>> trie['dach']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "trie.py", line 78, in __getitem__
    matched_node = self._get_node_by_key(self.root, key)
  File "trie.py", line 147, in _get_node_by_key
    raise KeyError(key)
KeyError: 'dach'
>>> trie['dach'] = 2
>>> trie['kot'] = 5
>>> trie['koc'] = 3
>>> trie['los'] = -3
>>> trie['lot'] = 0
>>> len(trie)
5
>>> list(trie)
['lot', 'los', 'koc', 'kot', 'dach']
>>> trie.keys()   # wynik dla Pythona 3
KeysView(<recursive_trie.RecursiveTrie object at 0x7f8145b26b38>)
>>> list(trie.keys())
['lot', 'los', 'koc', 'kot', 'dach']
>>> trie.values()   # wynik dla Pythona 3
ValuesView(<recursive_trie.RecursiveTrie object at 0x7f8145b26b38>)
>>> list(trie.values())
[0, -3, 3, 5, 2]
>>> trie.items()   # wynik dla Pythona 3
ItemsView(<recursive_trie.RecursiveTrie object at 0x7f8145b26b38>)
>>> list(trie.items())
[('lot', 0), ('los', -3), ('koc', 3), ('kot', 5), ('dach', 2)]
>>> trie.keys_with_prefix('ko')
```

```
<trieiterator.TrieIterator object at 0x7f81451a8358>
>>> list(trie.keys_with_prefix('ko'))
['koc', 'kot']
>>> trie.longest_prefix_of('kotlet')
'kot'
```

---

7. Podsumowanie

Niniejsza praca prezentuje sposób implementacji drzew trie i drzew Patricia w języku Python.

W pierwszej części pracy przedstawiono teoretyczne podstawy opisywanych zagadnień. Scharakteryzowano język Python użyty w implementacji i opisano dokumenty PEP, które dokumentują kierunki rozwoju języka. Dalej przestawiono wprowadzenie do testowania jednostkowego ze szczególnym uwzględnieniem testowania w języku Python. Następnie zostały opisane struktury danych - grafy oraz drzewa, które są podstawą drzew trie i drzew Patricia. Omówiono reprezentację węzłów dla drzew, proces przeszukiwania drzew, wstawianie i usuwanie elementów.

Praktyczna część pracy przedstawia implementację drzew trie w języku Python. Opisane zostały abstrakcyjne klasy węzła i drzewa trie, oraz rekurencyjna i iteracyjna implementacja oparta o reprezentację *na lewo syn, na prawo brat*. Na koniec zaprezentowane zostało skompresowane drzewo Patricia, które pozwala ograniczyć zapotrzebowanie struktury danych na pamięć. Drzewa trie przygotowane w ramach tej pracy mają interfejs pythonowego słownika i mogą z nim być stosowane zamiennie, jeżeli klucze są stringami. Wszystkie przedstawione implementacje zostały dostarczone z testami jednostkowymi wykorzystującymi pythonowy moduł unittest oraz z dokumentacją wykorzystującą format biblioteki numpy.

Wyżej wymienione implementacje mogą być podstawą bardziej złożonych systemów służących do przechowywania słowników wyrazów, czy przewidywania tekstu. Czytelna składnia języka Python sprawia, że stworzony kod może być wykorzystany do nauki prezentowanych struktur danych. Można próbować uogólnić kod tak, aby kluczami zamiast stringów mogły być dowolne obiekty dające dostęp do swoich fragmentów. Przykładami takich obiektów są liczby całkowite nieujemne (fragmenty to cyfry), adresy IP, ciągi bajtów (fragmenty to pojedyńcze bajty lub bity).

Bibliografia

[1] Wikipedia, Trie, 2018, `https://en.wikipedia.org/wiki/Trie`.

[2] Wikipedia, Radix tree, 2018, `https://en.wikipedia.org/wiki/Radix_tree`.

[3] Python Programming Language - Official Website, `https://www.python.org/`.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmow*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[5] Ewelina Matusiewicz, *Implementacja drzew binarnych w języku Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków 2017.

[6] Wikipedia, Binary search tree, 2018, `https://en.wikipedia.org/wiki/Binary_search_tree`.

[7] Wikipedia, Przechodzenie drzewa, 2018, `https://pl.wikipedia.org/wiki/Przechodzenie_drzewa`.

[8] Wikipedia, Left-child right-sibling binary tree, 2018, `https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree`.

[9] Kent Beck, *TDD. Sztuka tworzenia dobrego kodu*, Wydawnictwo Helion, 2014.

[10] Michał Jaworski, Tarek Ziade, *Profesjonalne programowanie w Pythonie*, Wydawnictwo Helion, 2016.

[11] Roy Osherove, *Testy jednostkowe. Świat niezawodnych aplikacji*, Wydawnictwo Helion, 2014.

[12] Robert Sedgewick, Kevin Wayne, *Algorytmy*, Wydawnictwo Helion, 2017.