UNIWERSYTET JAGIELLOŃSKI

Michał Fiejtek

Implementacja i Wizualizacja Drzew Czerwono-Czarnych

Projekt zaliczeniowy z przedmiotu:
Python

Kraków 2025

Spis treści

1 Wstęp i opis teoretyczny algorytmu

1.1 Czym są Drzewa Czerwono-Czarne?

Drzewo Czerwono-Czarne (ang. *Red-Black Tree*, RBT) to rodzaj samo-balansującego się binarnego drzewa poszukiwań (BST). Struktura ta gwarantuje, że wysokość drzewa pozostaje rzędu $O(\log n)$, co zapewnia pesymistyczny czas wykonywania podstawowych operacji (wstawianie, usuwanie, wyszukiwanie) również na poziomie $O(\log n)$.

1.2 Własności

Aby drzewo BST mogło być uznane za Czerwono-Czarne, musi spełniać następujące warunki:

1. Każdy węzeł ma kolor czerwony lub czarny.
2. Korzeń drzewa jest zawsze czarny.
3. Każdy liść (reprezentowany przez NIL/NULL) jest czarny.
4. Jeśli węzeł jest czerwony, to oba jego dzieci muszą być czarne (brak dwóch czerwonych węzłów pod rząd).
5. Dla każdego węzła, każda ścieżka od tego węzła do liści w jego poddrzewie zawiera tę samą liczbę czarnych węzłów (tzw. czarna wysokość).

1.3 Operacje naprawcze

Podczas modyfikacji drzewa (Insert, Delete) struktura może zostać naruszona. Do przywrócenia balansu wykorzystuje się:

- **Przekolorowanie:** Zmiana koloru węzła i jego rodzica/wujka.
- **Rotacje:** Lokalne zmiany struktury drzewa (rotacja w lewo lub w prawo), które zachowują porządek BST, ale zmieniają wysokość poddrzew.

2 Opis interfejsu użytkownika

Aplikacja została wyposażona w graficzny interfejs użytkownika (GUI) stworzony przy użyciu biblioteki `tkinter`. Pozwala on na interaktywne badanie działania algorytmu.

2.1 Funkcjonalności

- **Pole tekstowe:** Umożliwia wprowadzenie liczby całkowitej.
- **Przycisk "Dodaj (Insert)":** Wstawia wpisany klucz do drzewa i automatycznie wykonuje balansowanie.
- **Przycisk "Usuń (Delete)":** Usuwa węzeł o podanym kluczu. W przypadku błędu (klucz nie istnieje) wyświetlane jest okno dialogowe.
- **Przycisk "Szukaj (Search)":** Podświetla węzeł o podanym kluczu. W przypadku błędu (klucz nie istnieje) wyświetlane jest okno dialogowe.
- **Przycisk "Losowe":** Generuje losowe drzewo składające się z 5-15 węzłów w celu szybkiego testowania.
- **Przycisk "Wyczyść":** Resetuje aplikację do stanu pustego drzewa.

2.2 Wizualizacja

Drzewo jest rysowane dynamicznie na kanwie (Canvas). Węzły czerwone i czarne są reprezentowane przez odpowiednie kolory. Algorytm rysujący rekurencyjnie oblicza pozycje węzłów, zapewniając czytelność struktury nawet przy kilku poziomach zagłębienia.

3 Uwagi na temat implementacji

Projekt został zaimplementowany w języku Python z podziałem na moduły.

3.1 Struktura projektu

- `implementation/node.py` - Definicja klasy węzła i stałych kolorów.
- `implementation/red_black_tree.py` - Główna logika (wstawianie, usuwanie, rotacje, naprawa drzewa).
- `tests/test_red_black_tree.py` - Testy jednostkowe.
- `visualization/app.py` - Interfejs graficzny.

3.2 Zastosowanie węzła wartownika (Sentinel Node)

W standardowych implementacjach drzew binarnych, brak dziecka (liść) reprezentowany jest przez wartość `None` (lub `null`). W przypadku Drzew Czerwono-Czarnych prowadzi to do komplikacji w kodzie, ponieważ algorytmy balansujące często muszą sprawdzać kolor węzła. Próba odczytania koloru z wartości `None` spowodowałaby błąd wykonania (*AttributeError*).

W prezentowanym projekcie zastosowano technikę **wartownika** (`self.TNULL`). Jest to specjalny obiekt klasy `Node`, który:

- Reprezentuje wszystkie liście w drzewie oraz rodzica korzenia.
- Zawsze posiada kolor **CZARNY** (spełnienie 3. zasady RBT).
- Pozwala na wyeliminowanie wielu instrukcji warunkowych sprawdzających, czy węzeł istnieje (np. `if node is not None`).

3.3 Wskaźniki na rodzica

Każdy węzeł w drzewie przechowuje nie tylko referencje do dzieci (`left`, `right`), ale również wskaźnik do rodzica (`parent`). Jest to kluczowe dla efektywności operacji naprawczych (`fix_insert` i `fix_delete`). Algorytm musi mieć możliwość szybkiego przemieszczania się w górę drzewa, aby sprawdzić kolor "wujka"(brata rodzica) lub dokonać rotacji wokół dziadka, bez konieczności stosowania stosu lub rekurencji do zapamiętywania ścieżki powrotnej.

3.4 Operacja przeszczepiania (Transplant)

W celu zwiększenia czytelności metody usuwania (`delete_node`), wydzielono logikę zastępowania jednego poddrzewa drugim do osobnej metody pomocniczej `rb_transplant(u, v)`. Funkcja ta obsługuje przepinanie wskaźników rodzica:

1. Jeśli `u` było korzeniem, `v` staje się nowym korzeniem.
2. Jeśli `u` było lewym dzieckiem, `v` staje się lewym dzieckiem rodzica `u`.
3. W przeciwnym razie `v` staje się prawym dzieckiem.

Dzięki temu główna logika usuwania skupia się na zachowaniu własności BST i wywołaniu naprawy kolorów, a nie na niskopoziomowym przepinaniu referencji.

3.5 Podejście iteracyjne

Większość operacji modyfikujących drzewo (`insert`, `delete`, oraz procedury naprawcze `fix`) zaimplementowano w sposób **iteracyjny** (z użyciem pętli `while`), a nie rekurencyjny. Decyzja ta wynika z ograniczeń języka Python dotyczących głębokości rekurencji (*RecursionError*). Podejście iteracyjne jest również bardziej efektywne pamięciowo, gdyż nie obciąża stosu wywołań systemowych. Jedynie operacja wyszukiwania (`search`) oraz wizualizacja zostały zaimplementowane rekurencyjnie ze względu na ich prostotę i brak wpływu na stabilność przy standardowych rozmiarach danych.

3.6 Fragment implementacji kluczowej metody

Poniżej przedstawiono implementację metody `rb_transplant`, ilustrującą obsługę wskaźników rodziców oraz wartownika:

```
def rb_transplant(self, u, v):
    """Zastepuje poddrzewo wezla u poddrzewem v."""
    if u.parent is None:
        self.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    v.parent = u.parent
```

Listing 1: Rotacja w lewo

3.7 Implementacja rotacji

Rotacje są podstawowym mechanizmem modyfikacji struktury drzewa, który nie narusza kolejności elementów (in-order traversal), ale zmienia wysokość poddrzew. Poniższy kod przedstawia rotację w lewo (`left_rotate`). Kluczowym aspektem jest tu precyzyjna aktualizacja wskaźników rodziców (`parent`) dla trzech węzłów: `x`, `y` oraz dziecka `y`. Operacja ta jest wykonywana w czasie stałym $O(1)$.

```
def left_rotate(self, x):
    y = x.right             # y staje sie nowym korzeniem poddrzewa
    x.right = y.left        # Przeniesienie lewego poddrzewa y do x

    if y.left != self.TNULL:
        y.left.parent = x

    y.parent = x.parent     # Przepiecie rodzica x do y

    if x.parent is None:    # x byl korzeniem calego drzewa
        self.root = y
    elif x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y

    y.left = x              # x staje sie lewym dzieckiem y
    x.parent = y
```

Listing 2: Rotacja w lewo

3.8 Algorytm wyszukiwania

Wyszukiwanie w Drzewie Czerwono-Czarnym działa identycznie jak w standardowym drzewie BST. Implementacja rekurencyjna wykorzystuje wartownika `TNULL` jako warunek stopu. Złożoność tej operacji jest ograniczona wysokością drzewa, co dzięki balansowaniu gwarantuje czas $O(\log n)$.

```
def _search_tree_helper(self, node, key):
    # Warunek stopu: dotarcie do liscia (TNULL) lub znalezienie klucza
    if node == self.TNULL or key == node.val:
```

```
        return node

    if key < node.val:
        return self._search_tree_helper(node.left, key)

    return self._search_tree_helper(node.right, key)
```

Listing 3: Rekurencyjne wyszukiwanie

3.9 Logika naprawy po wstawieniu (Przypadek Wujka)

Najbardziej charakterystycznym elementem Drzew Czerwono-Czarnych jest podejmowanie decyzji o naprawie na podstawie koloru "Wujka"(brata rodzica). Poniższy fragment metody `fix_insert` ilustruje ten mechanizm:

1. Jeśli Wujek jest CZERWONY: wystarczy przekolorować węzły (problem przesuwa się w górę).
2. Jeśli Wujek jest CZARNY: konieczna jest rotacja (problem rozwiązywany lokalnie).

```
# ... wewnatrz petli while ...
if u.color == RED:
    # Przypadek 1: Wujek czerwony -> Tylko przekolorowanie
    u.color = BLACK
    k.parent.color = BLACK
    k.parent.parent.color = RED
    k = k.parent.parent  # Kontynuujemy sprawdzanie dla dziadka
else:
    # Przypadek 2: Wujek czarny -> Rotacja
    if k == k.parent.left: # "Trojkat" - wymaga podwojnej rotacji
        k = k.parent
        self.right_rotate(k)

    # Przypadek 3: "Linia" - rotacja wokol dziadka
    k.parent.color = BLACK
    k.parent.parent.color = RED
    self.left_rotate(k.parent.parent)
```

Listing 4: Fragment metody fix_insert

4 Instrukcja uruchomienia

Aby poprawnie uruchomić aplikację oraz testy, należy upewnić się, że zainstalowany jest interpreter języka Python w wersji 3.x. Biblioteka graficzna `tkinter` jest zazwyczaj dołączona do standardowej instalacji Pythona.

4.1 Struktura katalogów

Wszystkie polecenia należy wykonywać z poziomu **głównego katalogu projektu** (tam, gdzie widoczne są foldery `implementation`, `tests` oraz `visualization`).

4.2 Uruchomienie wizualizacji

Aby włączyć graficzny interfejs użytkownika, w terminalu (konsoli) należy wpisać:

```
python visualization/app.py
```

W systemach Linux/macOS może być konieczne użycie polecenia `python3`.

4.3 Uruchomienie testów

Aby zweryfikować poprawność implementacji za pomocą przygotowanych testów jednostkowych, należy użyć modułu `unittest`. Pozwala to na automatyczne wykrycie testów w folderze:

```
python -m unittest discover tests
```

Alternatywnie można uruchomić konkretny plik testowy:

```
python -m unittest tests/test_red_black_tree.py
```

Uwaga: Uruchamianie plików bezpośrednio z wewnątrz podkatalogów (np. będąc w folderze `tests`) może spowodować błędy importu modułów. Należy zawsze pozostawać w katalogu głównym projektu.

5 Podsumowanie i wyniki testów

5.1 Testowanie

Do weryfikacji poprawności działania algorytmu wykorzystano moduł `unittest`. Zaimplementowano specjalną metodę `validate_tree`, która rekurencyjnie sprawdza:

- Czy nie występują dwa czerwone węzły obok siebie.
- Czy czarna wysokość jest identyczna dla każdej ścieżki.
- Czy zachowana jest relacja BST (lewe dziecko mniejsze, prawe większe).

Przeprowadzono testy brzegowe (usuwanie korzenia, puste drzewo) oraz testy obciążeniowe (losowe wstawianie i usuwanie 1000 elementów), które zakończyły się sukcesem.

5.2 Wnioski

Zaimplementowane drzewo poprawnie utrzymuje balans, co potwierdzają testy oraz wizualizacja. Złożoność czasowa operacji pozostaje logarytmiczna. Interfejs graficzny pozwala na intuicyjne zrozumienie mechaniki działania rotacji.

6 Literatura

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN.
2. Dokumentacja języka Python: `https://docs.python.org/3/`
3. Dokumentacja biblioteki Tkinter: `https://docs.python.org/3/library/tkinter.html`
4. Wprowadzenie do drzew czerwono czarnych: `https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/`
5. Implementacja drzew czerwono czarnych: `https://www.geeksforgeeks.org/python/red-black-tree-in-python/`