

# Dokumentacja Implementacji Drzewa Czerwono-Czarnego (Red-Black Tree)

**Autor:** Marian Wachała

**Data:** 7 stycznia 2025

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Drzewo czerwono-czarne . . . . .	2
1.2	Założenia drzewa czerwono-czarnego . . . . .	2
1.3	Podstawowe pojęcia w drzewie czerwono-czarnym . . . . .	2
<b>2</b>	<b>Opis Implementacji</b>	<b>2</b>
2.1	Rola NIL jako strażnika . . . . .	2
2.2	Podstawowe metody . . . . .	3
2.2.1	Wstawianie elementu . . . . .	3
2.2.2	Naprawa drzewa po wstawieniu ( <b>fixInsert</b> ) . . . . .	3
2.2.3	Usuwanie elementu . . . . .	4
2.2.4	naprawa drzewa po usunięciu elementu ( <b>fixDelete</b> ) . . . . .	4
2.2.5	Wyszukiwanie elementu . . . . .	5
2.3	Metody pomocnicze . . . . .	5
<b>3</b>	<b>Podsumowanie</b>	<b>5</b>
3.1	Złożoność operacji dodawania i usuwania elementów z drzewa czerwono-czarnego . . . . .	5

# 1 Wstęp

## 1.1 Drzewo czerwono-czarne

**Drzewo czerwono-czarne** (ang. *Red-Black Tree*) to samobalansujące się drzewo BST (ang. *Binary Search Tree*), które zapewnia średni czas wstawiania, usuwania i wyszukiwania na poziomie  $\mathcal{O}(\log n)$ .

Zachowanie równowagi jest możliwe dzięki specjalnym regułom związanym z kolorowaniem węzłów na kolor czerwony (RED) lub czarny (BLACK) oraz dzięki odpowiednim operacjom rotacji (lewo- i prawostronnej).

## 1.2 Założenia drzewa czerwono-czarnego

Drzewa czerwono-czarne opierają się na następujących regułach:

1. Każdy węzeł jest **czerwony** lub **czarny**.
2. **Strażnik NIL** jest zawsze czarny i działa jako końcowy węzeł (liść) w każdej ścieżce.
3. Korzeń drzewa jest zawsze **czarny**.
4. Każda ścieżka od węzła do liścia zawiera taką samą liczbę **czarnych węzłów**.
5. Czerwony węzeł nie może mieć czerwonych rodziców ani dzieci (własność **Black Depth**).

## 1.3 Podstawowe pojęcia w drzewie czerwono-czarnym

Podczas opisywania operacji na drzewie czerwono-czarnym używane są następujące terminy:

- **Ojciec (Parent)** – bezpośredni przodek danego węzła w drzewie.
- **Dziadek (Grandparent)** – przodek węzła będący ojcem jego ojca.
- **Wujek (Uncle)** – węzeł będący drugim dzieckiem dziadka, który nie jest ojcem danego węzła.
- **Brat (Sibling)** – inne dziecko tego samego ojca co dany węzeł.

# 2 Opis Implementacji

## 2.1 Rola NIL jako strażnika

W implementacji zastosowano specjalny węzeł NIL, który:

- Działa jako wskaźnik na wszystkie puste liście w drzewie.
- Upraszcza implementację, eliminując konieczność sprawdzania, czy wskaźniki są `nullptr`.
- Wspiera zachowanie spójności drzewa podczas operacji naprawczych (`fixInsert` i `fixDelete`).

## 2.2 Podstawowe metody

### 2.2.1 Wstawianie elementu

Proces wstawiania polega na dodaniu nowego węzła do drzewa zgodnie z zasadami BST, a następnie naprawie potencjalnych naruszeń zasad drzewa czerwono-czarnego za pomocą metody `fixInsert`.

```
1 void insert(const T& key);
```

Metoda wykonuje następujące kroki:

1. Tworzy nowy węzeł o wartości `key` z kolorem RED.
2. Znajduje odpowiednie miejsce w drzewie dla nowego węzła.
3. Wywołuje `fixInsert` w celu przywrócenia właściwości drzewa czerwono-czarnego.

### 2.2.2 Naprawa drzewa po wstawieniu (`fixInsert`)

Gdy nowy węzeł zostanie wstawiony, może dojść do naruszenia reguł drzewa czerwono-czarnego. Naprawa odbywa się w następujących krokach:

**Przypadek A: Ojciec węzła jest lewym dzieckiem dziadka.**

- 1. Wujek jest czerwony:
  - Zmieniamy kolor dziadka na czerwony.
  - Zmieniamy kolor ojca na czarny.
  - Zmieniamy kolor wujka na czarny.
  - Przechodzimy do dalszego przetwarzania od dziadka.
- 2. Wujek jest czarny:
  - a. Węzeł jest lewym dzieckiem ojca:
    - \* Wykonujemy rotację w lewo na ojcu.
    - \* Aktualizujemy odniesienia do węzła i jego rodzica.
  - b. Węzeł jest prawym dzieckiem ojca:
    - \* Wykonujemy rotację w prawo na dziadku.
    - \* Zamieniamy kolory ojca i dziadka.
    - \* Przechodzimy do dalszego przetwarzania od ojca.

**Przypadek B:** Ojciec węzła jest prawym dzieckiem dziadka.

Wykonujemy to samo, tylko, że podmieniamy `left` z `right`.

### 2.2.3 Usuwanie elementu

Proces usuwania obejmuje:

```
1 bool remove(const T& key);
```

1. Wyszukanie węzła o wartości `key`.
2. Przypadek dla węzła z jednym dzieckiem lub bez dzieci:
  - Usunięcie węzła i podłączenie jego dziecka.
3. Przypadek dla węzła z dwoma dziećmi:
  - Znalezienie najmniejszego węzła w prawym poddrzewie (`minValueNode`).
  - Zastąpienie wartości usuwanego węzła wartością znalezionego węzła.
  - Usunięcie znalezionego węzła.
4. Wywołanie `fixDelete` w celu przywrócenia właściwości drzewa.

### 2.2.4 naprawa drzewa po usunięciu elementu (`fixDelete`)

Podobnie jak w przypadku wstawiania, po usunięciu elementu może dojść do naruszenia własności drzewa czerwono-czarnego. Naprawa po usunięciu elementu odbywa się w następujących krokach:

**Przypadek A: Węzeł jest lewym dzieckiem rodzica.**

- **1. Brat węzła jest czerwony:**
  - Zmieniamy kolor brata na czarny.
  - Zmieniamy kolor rodzica na czerwony.
  - Wykonujemy rotację w lewo na rodzicu.
  - Aktualizujemy odniesienie do brata (teraz jest to prawy syn rodzica po rotacji).
- **2. Brat węzła jest czarny i ma dwóch czarnych synów:**
  - Zmieniamy kolor brata na czerwony.
  - Przechodzimy do dalszego przetwarzania od rodzica.
- **3. Brat węzła jest czarny i jego prawy syn jest czarny, ale lewy syn jest czerwony:**
  - Zmieniamy kolor lewego syna brata na czarny.
  - Zmieniamy kolor brata na czerwony.
  - Wykonujemy rotację w prawo na bracie.
  - Aktualizujemy odniesienie do brata (teraz jest to prawy syn rodzica po rotacji).

- **4. Brat węzła jest czarny i jego prawy syn jest czerwony:**

- Zmieniamy kolor brata na kolor rodzica.
- Zmieniamy kolor rodzica na czarny.
- Zmieniamy kolor prawego syna brata na czarny.
- Wykonujemy rotację w lewo na rodzicu.
- Przerywamy pętlę (ustawiamy węzeł na korzeń).

**Przypadek B: Węzeł jest prawym dzieckiem rodzica.**

Postępujemy analogicznie jak w przypadku A, zamieniając kierunki rotacji oraz role lewego i prawego syna.

### 2.2.5 Wyszukiwanie elementu

Metoda wyszukuje element w drzewie:

```
1 Node<T>* search(Node<T>* root, const T& key);
```

Wyszukiwanie odbywa się rekurencyjnie w oparciu o reguły BST:

- Jeśli wartość **key** jest równa wartości bieżącego węzła, zwracany jest wskaźnik do węzła.
- Jeśli wartość **key** jest mniejsza, wyszukiwanie kontynuowane jest w lewym poddrzewie.
- Jeśli wartość **key** jest większa, wyszukiwanie kontynuowane jest w prawym poddrzewie.

## 2.3 Metody pomocnicze

- `rotateLeft(Node<T>* node)` – Rotacja w lewo względem węzła `node`.
- `rotateRight(Node<T>* node)` – Rotacja w prawo względem węzła `node`.
- `minValueNode(Node<T>* node)` – Znajduje węzeł o najmniejszej wartości w poddrzewie.
- `transplant(Node<T>* u, Node<T>* v)` – Zamienia węzeł `u` z węzłem `v`.

## 3 Podsumowanie

### 3.1 Złożoność operacji dodawania i usuwania elementów z drzewa czerwono-czarnego

Drzewa czerwono-czarne są niezwykle efektywnymi strukturami danych, które dzięki swoim właściwościom zapewniają, że operacje wstawiania i usuwania są wykonywane w czasie  $\mathcal{O}(\log n)$ . Utrzymanie zrównoważenia drzewa poprzez odpowiednie rotacje i zmiany kolorów węzłów gwarantuje, że nawet w najgorszym przypadku wysokość drzewa pozostaje na poziomie logarytmicznym względem liczby jego elementów.