POLITECHNIKA BYDGOSKA

im. Jana i Jędrzeja Śniadeckich

WYDZIAŁ TELEKOMUNIKACJI, INFORMATYKI I ELEKTROTECHNIKI

Bezpieczeństwo - Projekt

Hashowanie z wykorzystaniem drzewa BST

Michał Klamka

Sławomir Piotrzkowski

Konrad Walczak

Spis Treści

1. Cel projektu

Celem niniejszego projektu jest zbadanie wydajności oraz funkcjonalności metody przechowywania danych opartej na połączeniu haszowania z drzewem BST (Binary Search Tree) jako strukturą przechowującą dane po przekształceniu ich przy pomocy funkcji haszującej. Projekt ma charakter badawczo-eksperymentalny i obejmuje zarówno implementację systemu, jak i jego testowanie oraz analizę wyników w różnych warunkach.

Główne cele szczegółowe projektu:

1. **Implementacja struktury danych wykorzystującej połączenie haszowania z BST:**
    - Zaprojektowanie i zaimplementowanie mechanizmu, w którym funkcja haszująca przekształca dane wejściowe do kluczy liczbowych, a następnie klucze te są przechowywane w strukturze drzewa BST.
    - Obsługa podstawowych operacji: wstawianie, wyszukiwanie, usuwanie.

2. **Analiza efektywności i złożoności rozwiązania:**
    - Pomiar złożoności obliczeniowej operacji w praktyce (czas działania w funkcji liczby elementów).
    - Ocena wpływu rozkładu danych wejściowych na efektywność działania (np. dane losowe, uporządkowane, z dużą liczbą kolizji).
    - Pomiar zużycia zasobów systemowych: pamięci RAM, czasu CPU.

3. **Projekt środowiska testowego:**
    - Stworzenie zestawu danych testowych o różnych rozkładach statystycznych (jednostajny, normalny, Zipfa).
    - Ustandaryzowanie środowiska pomiarowego w celu zapewnienia porównywalności wyników (np. poprzez wyłączenie wpływu tła systemowego, wielokrotne uruchamianie testów).
    - Zapewnienie możliwości odtwarzalności testów.

4. **Porównanie z klasycznym podejściem:**
    - Zestawienie wyników uzyskanych dla BST z wynikami dla klasycznych tablic mieszających dostępnych np. wbudowanie w język Python (`dict`).
    - Analiza zalet i wad obu podejść w kontekście różnych typów danych i zastosowań.

5. **Opracowanie wniosków praktycznych:**
    - Określenie scenariuszy, w których użycie BST jako struktury wspomagającej haszowanie może mieć uzasadnienie praktyczne.
    - Identyfikacja ograniczeń rozwiązania i potencjalnych kierunków optymalizacji (np. zastosowanie AVL lub Red-Black Tree zamiast klasycznego BST).

Oczekiwane rezultaty:

- Gotowa implementacja systemu haszowania z użyciem BST w języku Python.
- Wyniki testów opisujące efektywność rozwiązania w różnych warunkach.
- Zestaw wniosków i rekomendacji dotyczących praktycznej przydatności analizowanego podejścia.

2. Wstęp Teoretyczny

2.1 Hashowanie

Hashowanie to technika wykorzystywana do szybkiego przechowywania i wyszukiwania danych na podstawie kluczy. Polega na przekształceniu klucza (np. tekstowego lub liczbowego) w indeks tablicy przy pomocy specjalnej funkcji – tzw. funkcji haszującej. Ostatecznym celem hashowania jest osiągnięcie jak najkrótszego czasu dostępu do danych – **średnio w czasie O(1)**.

W praktyce jednak funkcja haszująca może wygenerować ten sam indeks (adres) dla różnych kluczy, co prowadzi do tzw. **kolizji**. Kolizje są nieuniknione w przypadku ograniczonego rozmiaru tablicy i dużej liczby możliwych kluczy. Dlatego każda implementacja tablicy mieszającej musi uwzględniać mechanizm obsługi kolizji. Do najczęściej stosowanych metod należą:

- **Łańcuchowanie (chaining)** – każdy indeks w tablicy przechowuje listę elementów o tej samej wartości haszu.

- **Adresowanie otwarte (open addressing)** – próba znalezienia innego wolnego miejsca w tablicy przy wystąpieniu kolizji.

2.2 Drzewo BST (Binary Search Tree)

Drzewo BST (Binary Search Tree) to dynamiczna struktura danych, w której każdy węzeł posiada maksymalnie dwóch potomków – lewego i prawego. Wartości w lewym poddrzewie są mniejsze niż wartość węzła, natomiast wartości w prawym poddrzewie są większe. Ta właściwość BST umożliwia szybkie przeszukiwanie, wstawianie i usuwanie danych.

W przypadku **dobrze zrównoważonego drzewa** (np. drzew AVL lub czerwono-czarnych), podstawowe operacje mają złożoność czasową rzędu **O(log n)**. W najgorszym przypadku (np. dla silnie niesymetrycznych danych) złożoność może wzrosnąć do **O(n)**, dlatego struktura drzewa powinna być odpowiednio kontrolowana lub równoważona.

BST jest szczególnie użyteczne, gdy zachodzi potrzeba utrzymywania danych w porządku oraz wykonywania operacji zakresowych (np. wyszukiwania wartości większych/mniejszych niż zadana).

2.3 Koncepcja integracji hashowania i drzewa BST

Głównym pomysłem analizowanego w projekcie rozwiązania jest zastosowanie drzewa BST jako struktury obsługującej kolizje w tablicy haszującej. Tradycyjnie, kolizje obsługuje się za pomocą list (np. listy wiązanej) – jednak taka struktura zapewnia jedynie liniowy czas wyszukiwania (O(n)) w obrębie pojedynczego "bucketu". Zastąpienie listy przez BST może znacząco poprawić efektywność operacji, zwłaszcza w przypadku dużej liczby kolizji.

W proponowanej koncepcji:

- Tablica hash składa się z **bucketów** – miejsc, w których przechowywane są dane.
- Zamiast listy, każdy bucket zawiera drzewo BST, w którym dane są uporządkowane względem kluczy.
- W przypadku kolizji – gdy wiele elementów trafia do tego samego “bucketu”: – dane są wstawiane do drzewa, co umożliwia późniejsze **logarytmiczne przeszukiwanie**.

Zalety tego podejścia:

- Lepsza wydajność w przypadku dużej liczby kolizji w jednym bucketcie (O(log n) zamiast O(n)).
- Uporządkowana struktura danych w bucketach, co może być przydatne w aplikacjach wymagających operacji zakresowych.
- Potencjalna odporność na niekorzystne rozkłady danych wejściowych, w których tradycyjne listy mogłyby spowodować spadek wydajności.

Potencjalne ograniczenia:

- Większy narzut pamięci i bardziej złożona implementacja w porównaniu do prostych list.
- Wymagane są dodatkowe mechanizmy dbające o zrównoważenie drzewa, by uniknąć degeneracji BST do postaci listy.

2.4 Podsumowanie

Połączenie mechanizmu hashującego z dynamiczną strukturą drzewa BST stanowi ciekawe podejście hybrydowe. Takie rozwiązanie może przynieść wymierne korzyści wydajnościowe w określonych scenariuszach – szczególnie w środowiskach, gdzie kolizje występują często, a jednocześnie zależy nam na szybkim dostępie do danych. W dalszej części pracy przeprowadzona zostanie praktyczna analiza tej koncepcji, uwzględniająca zarówno aspekty algorytmiczne, jak i zużycie zasobów systemowych.

3. Opis implementacji

W tej części przedstawione zostaną szczegóły realizacji obu podejść: klasycznego drzewa BST oraz zrównoważonego drzewa AVL, używanych jako struktury danych obsługujące kolizje w tablicy haszującej.

3.1 Klasyczne drzewo BST

1. **Struktura węzła**:
    - `key` – klucz (wynik funkcji haszującej lub oryginalny klucz),
    - `value` – powiązana wartość,
    - `left`, `right` – wskaźniki na dzieci (mogą być `None`).
2. **Operacje podstawowe**:
    - `insert(node, key, value)` – rekurencyjnie wstawia nowy węzeł w miejsce liścia; w przypadku istnienia klucza nadpisuje wartość,
    - `search(node, key)` – rekurencyjnie porównuje klucz z `node.key` i schodzi w lewo lub prawo aż do znalezienia lub końca drzewa.
3. **Integracja z tablicą haszującą**:
    - Tablica o rozmiarze `size` składa się z listy obiektów BST (po jednym drzewie na bucket),
    - `hash_func(key)` zwraca indeks bucketu: `hash(key) % size`,
    - `insert(key, value)` oraz `search(key)` delegują operacje do drzewa z odpowiedniego bucketu.
4. **Złożoność i ryzyko degeneracji**:
    - Średnia złożoność operacji: O(log n) przy danych o równomiernym rozkładzie,
    - W najgorszym przypadku (np. dane uporządkowane) BST degeneruje się do listy – złożoność O(n), co znacząco pogarsza wydajność.

3.2 Zrównoważone drzewo AVL

Aby wyeliminować ryzyko degeneracji, w każdym wstawieniu dbamy o równowagę drzewa.

1. **Rozszerzona struktura węzła**:
    - `height` – wysokość poddrzewa z korzeniem w danym węźle.
2. **Obliczanie wysokości i współczynnika zrównoważenia**:
    - Metoda `_height(node)` zwraca `node.height` lub 0,
    - `_update_height(node)` aktualizuje wysokość jako `1 + max(height(left), height(right))`,
    - `_balance_factor(node)` = `height(left) - height(right)`.
3. **Rotacje**:
    - **Rotacja w prawo** (`_rotate_right`):
        - Weź lewego syna jako nowy korzeń poddrzewa,
        - Przypisz odpowiednio poddrzewa,
        - Zaktualizuj wysokości.

    - **Rotacja w lewo** (`_rotate_left`): analogicznie dla prawego syna.
4. **Rebalansowanie po wstawieniu**:
    - Po zwykłym wstawieniu węzła (podobnie jak w BST) aktualizujemy wysokość bieżącego węzła,
    - Obliczamy współczynnik zrównoważenia,
    - W zależności od jego wartości i relacji nowego klucza do klucza dziecka wykonujemy jedną z czterech rotacji:
        - LR, LL, RL, RR.
5. **Integracja z tablicą haszującą**:
    - Struktura analogiczna do BST, lecz buckety przechowują obiekty `AVLTree`,
    - Dzięki gwarancji zrównoważenia operacje wstawienia i wyszukiwania zawsze mają złożoność O(log n), nawet w przypadku niekorzystnych danych.

3.3 Podsumowanie implementacyjne

- Klasyczne BST jest proste i ma niski narzut pamięciowy, ale ryzykuje degradacją wydajności,
- AVL eliminuje ryzyko degradacji kosztem dodatkowych obliczeń przy rebalansowaniu i przechowywania wysokości w każdym węźle,
- Oba podejścia współpracują z tym samym interfejsem tablicy haszującej, co umożliwia łatwe porównanie ich wydajności w kolejnych eksperymentach.

4. Projekt Testów

W celu rzetelnej oceny wydajności zaimplementowanej struktury haszującej opartej na drzewie BST, przygotowano zestaw testów porównawczych w różnych warunkach oraz na różnych typach danych wejściowych. Celem testów było zmierzenie rzeczywistej efektywności operacji `insert` oraz `search` w porównaniu z innymi strukturami, takimi jak lista i słownik (Python `dict`), a także z AVL – zrównoważonym drzewem binarnym.

5.1 Środowisko testowe

Całość implementacji oraz testów wykonano w języku Python (wersja 3.12.10), wykorzystując środowisko Jupyter Notebook. Dodatkowo przygotowano skrypt `test.py`, umożliwiający wykonanie testów i zrzut wyników do plików `.csv` bez konieczności korzystania z notebooka. Skrypt ten nie generuje wykresów, lecz może być uruchamiany automatycznie w innych środowiskach.

Do pomiaru czasu wykonania zastosowano funkcję `time.perf_counter()`, która umożliwia precyzyjny pomiar czasu działania operacji.

W celu pomiaru szybkości wykonania użyto time.perf_counter() - pokazujacvy dokładny czas

5.2 Kalibracja środowiska

Przed rozpoczęciem testów właściwych wykonywany jest benchmark kalibracyjny, który pozwala oszacować wydajność maszyny wykonującej testy. Dzięki temu możliwe jest porównywanie wyników pomiędzy różnymi komputerami.

```
# Funkcja testu wzorcowego do standaryzacji
def calibration_benchmark():
    start = time.perf_counter()
    for i in range(100_000):
        _ = i ** 2
    return time.perf_counter() - start
```

Ten test polega na obliczeniu potęg liczb w zakresie 0–99 999 i stanowi orientacyjną miarę wydajności procesora.

5.3 Scenariusze testowe

Dla każdego zbioru danych wykonujemy dwie operacje:

- **Wstawianie wszystkich elementów** do struktury.
  Wyszukiwanie każdego z nich.

Czas wykonania tych operacji mierzony jest oddzielnie, co pozwala precyzyjnie ocenić wydajność danej implementacji.

W celu porównania struktur wykorzystano:

- **Listę** jako prymitywną strukturę par klucz–wartość (`O(n)` dla wyszukiwania).
- **Słownik (`dict`)** jako zoptymalizowaną strukturę haszującą (`O(1)` średnio).

5.3 Scenariusze testowe

Typy zbiorów danych testowych:

- **Posortowany** – `list(range(n))`: najgorszy przypadek dla BST (brak zrównoważenia).
- **Losowy unikalny** – permutacja liczb `0..n-1`: realistyczny scenariusz bez duplikatów.
- **Losowy** – liczby z dużego zakresu z możliwymi powtórzeniami: symulacja chaotycznych danych wejściowych.
- **Posortowany z szumem** – dane prawie uporządkowane z lekkimi zakłóceniami.
- **Powtarzający się** – wiele powtórzeń tych samych wartości: test wydajności w przypadku kolizji.

5.4 Pomiar zużycia pamięci – memory_profiler

Do analizy zużycia pamięci przez struktury danych wykorzystano bibliotekę **`memory_profiler`**, która umożliwia śledzenie ilości wykorzystywanej pamięci RAM przez wybrane fragmenty kodu w trakcie jego działania.

Pomiar wykonywany jest za pomocą funkcji `memory_usage`, która zwraca informacje o zapotrzebowaniu pamięci w określonym przedziale czasu.

Testy zostały umieszczone w osobnych plikach:

- `memory-test-BST.py`
- `memory-test-AVL.py`

Dla miliona elementów obie struktury wykazywały podobne zużycie pamięci – około **88 MB**. Należy jednak zaznaczyć, że narzędzie `memory_profiler` wykonuje pomiary w sposób okresowy, co może prowadzić do pewnych odchyleń. Dlatego uzyskane wyniki należy traktować jako **orientacyjne**, a nie absolutnie precyzyjne.

5.5 Instrukcja uruchomienia testów

1. **Instalacja Pythona**
   Wymagana jest instalacja języka Python, preferowana wersja: **3.12.10** (choć na nowszych wersjach testy również powinny działać poprawnie).

2. **Utworzenie i aktywacja wirtualnego środowiska**
   Zaleca się korzystanie z wirtualnego środowiska w celu izolacji zależności:
   Linux:
   ```
   python3 -m venv venv
   source venv/bin/activate
   ```
   Windows:
   ```
   python -m venv venv
   venv\Scripts\activate
   ```

3. **Instalacja zależności**
   Wszystkie niezbędne biblioteki należy zainstalować z pliku `requirements.txt`:
   ```
   pip install -r requirements.txt
   ```

4. **(Opcjonalnie) Konfiguracja środowiska Jupyter Notebook**
   W celu uruchomienia testów z pliku `test-benchmark.ipynb` formacie notebooka `.ipynb` można posłużyć się środowiskiem Jupyter Notebook. Obsługa notebooków może być zrealizowana na dwa sposoby:
   - poprzez **Visual Studio Code** z odpowiednimi rozszerzeniami (obsługa Jupyter jest dostępna bezpośrednio po zainstalowaniu rozszerzenia „Jupyter”),
   - lub poprzez klasyczne środowisko Jupyter skonfigurowane zgodnie z dokumentacją dostępną na oficjalnej stronie projektu: https://jupyter.org.

5. **Uruchomienie testów i eksport wyników do plików CSV**
   W celu uruchomienia testów bez wygenerowania wykresów:
   ```
   python3 test.py
   ```
   Skrypt ten wykona pomiary i zapisze wyniki do plików `.csv`.
6. **Testy zużycia pamięci**
   Testy pamięci znajdują się w osobnych plikach:
   - `memory-test-BST.py`
   - `memory-test-AVL.py`

Wszystkie skrypty i pliki źródłowe zostaną dołączone do projektu.

6. Wyniki testów:

Wyniki testów przedstawiono w postaci tabel oraz wykresów. Wykresy zostały znormalizowane względem wyników testów kalibracyjnych, co umożliwia ich interpretację niezależnie od sprzętu.

Dla każdego typu danych zmierzono czas operacji `insert` oraz `search` dla trzech głównych struktur:

- BST (drzewo binarne poszukiwań)
- AVL (drzewo zbalansowane)
- Dict (słownik Pythona)
- Lista (prosty wektor par)

6.1 Wyniki testów

Wyniki przedstawiające czas w sekundach oraz wartości znormalizowanej wykonania testów dla różnych scenariuszy:

Rezultaty - Dodawanie(s)

Struktura	Posortowana	Posortowana szum	Losowa unikalna	Losowa	Powtarzajaca
BST	0.066220	0.040765	0.014028	0.013598	0.005110
AVL	0.045074	0.038579	0.048713	0.048752	0.014978
List	0.001005	0.001102	0.000924	0.001125	0.001014
Dict	0.000738	0.000585	0.000664	0.000761	0.000458

Rezultaty - Dodawanie (znormalizowane)

Struktura	Posortowana	Posortowana szum	Losowa unikalna	Losowa	Powtarzajaca
BST	11.706591	7.206608	2.480006	2.403865	0.903281
AVL	7.968303	6.820104	8.611640	8.618534	2.647845
List	0.177616	0.194781	0.163437	0.198936	0.179295
Dict	0.130502	0.103366	0.117456	0.134604	0.081038

Rezultaty - Wyszukiwanie (s)

Struktura	Posortowana	Posortowana szum	Losowa unikalna	Losowa	Powtarzajaca
BST	0.059031	0.034916	0.009368	0.010554	0.004638
AVL	0.006294	0.006949	0.011729	0.010290	0.004647
List	1.335013	1.375774	1.986724	1.589953	0.289815
Dict	0.000299	0.000360	0.000333	0.000491	0.000396

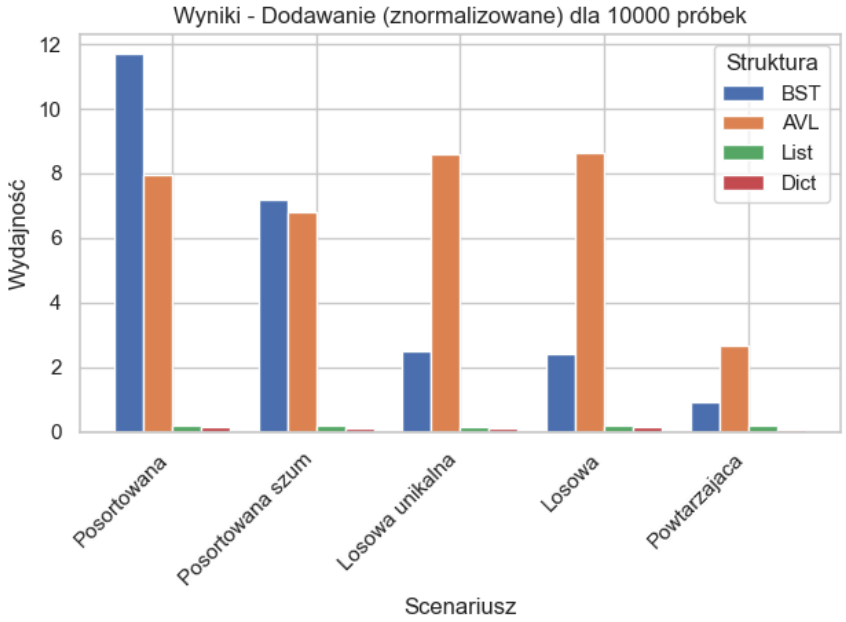
Rezultaty - Wyszukiwanie (znormalizowane)

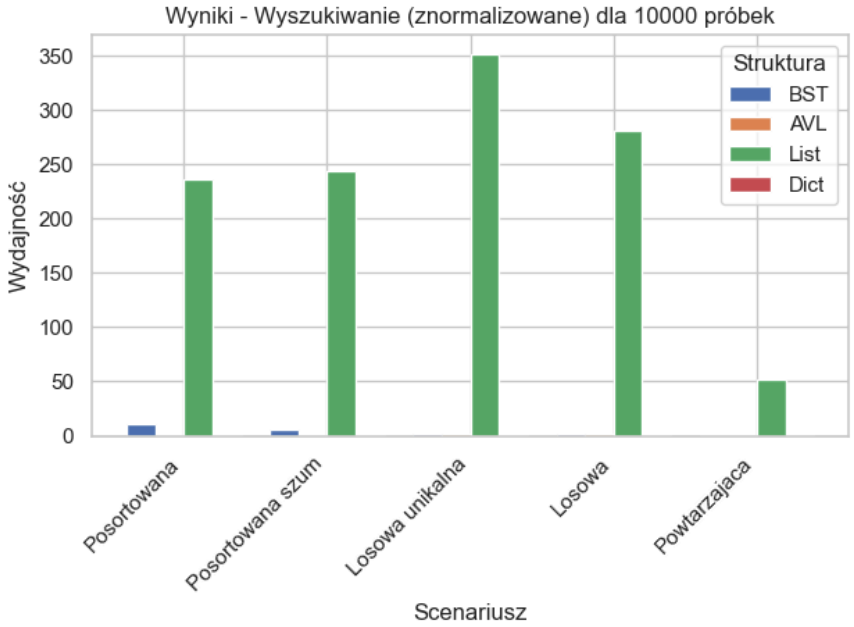
Struktura	Posortowana	Posortowana szum	Losowa unikalna	Losowa	Powtarzajaca
BST	10.435686	6.172613	1.656189	1.865750	0.819998
AVL	1.112753	1.228512	2.073542	1.819061	0.821518
List	236.009794	243.215713	351.222307	281.079253	51.234911
Dict	0.052823	0.063625	0.058834	0.086766	0.070024

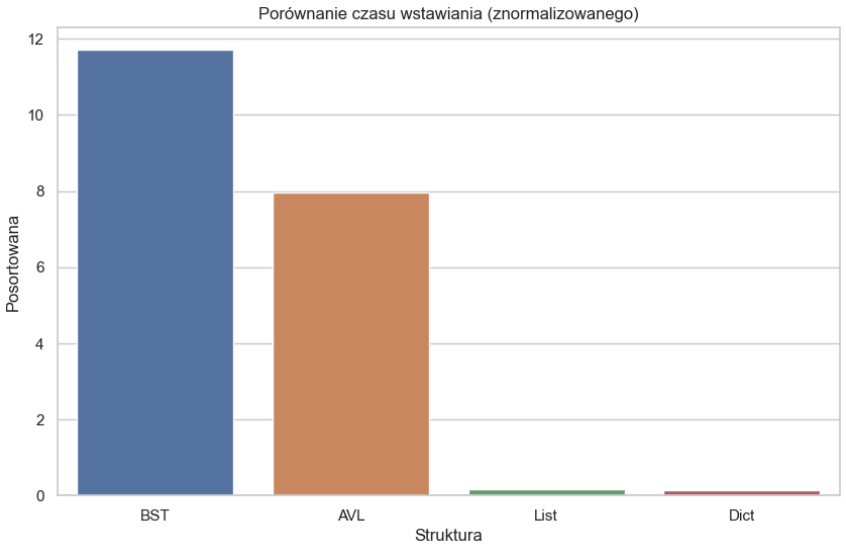
Testy pamięci:

Funkcja `memory_usage` została użyta do pomiaru średniego użycia pamięci podczas operacji. Dla miliona próbek oba drzewa wykazywały zużycie rzędu ~88 MB. Należy jednak zaznaczyć, że pomiar ten obarczony jest pewnym błędem i nie powinien być traktowany jako jednoznaczny.

6.2 Wykresy


Wyniki - Dodawanie (znormalizowane) dla 10000 próbek
Wydajność
Scenariusz
Struktura
BST
AVL
List
Dict
0
2
4
6
8
10
12
Posortowana
Posortowana szum
Losowa unikalna
Losowa
Powtarzajaca


Wyniki - Wyszukiwanie (znormalizowane) dla 10000 próbek
Wydajność
Scenariusz
Struktura
BST
AVL
List
Dict
0
50
100
150
200
250
300
350
Posortowana
Posortowana szum
Losowa unikalna
Losowa
Powtarzajaca



6.3 Obserwacje

- Wstawianie do BST dla posortowanych danych było znacznie wolniejsze niż dla danych losowych, co wynika z degeneracji struktury do listy jednokierunkowej.
- AVL zachowywał wysoką stabilność niezależnie od porządku danych, ale koszt równoważenia obniża jego wydajność przy danych losowych.
- Dict okazał się bezkonkurencyjny pod względem szybkości zarówno wstawiania, jak i wyszukiwania.
- Lista charakteryzowała się najgorszymi wynikami operacji wyszukiwania, co potwierdza jej nieefektywność dla tego typu zadań.

7. Wnioski

Na podstawie przeprowadzonych testów można sformułować następujące wnioski:

- **BST jest strukturą efektywną**, jednak tylko w warunkach, gdy dane wejściowe są losowe i niezbyt zdegenerowane. W przypadku danych posortowanych struktura ta traci swoje zalety.
- **AVL wypada lepiej niż BST w operacjach wyszukiwania**, zwłaszcza gdy dane wejściowe są losowe lub nieuporządkowane. Zbalansowanie drzewa znacząco poprawia efektywność, ale kosztem czasu wstawiania.
- **Słownik (`dict`) Pythona pozostaje bezkonkurencyjny**, zarówno pod względem wstawiania, jak i wyszukiwania. Wynika to z optymalizacji na poziomie języka i implementacji haszowania.
- **Lista** sprawdza się wyłącznie jako prosta struktura do przechowywania danych, ale nie nadaje się do wyszukiwania – operacje `search` są wyjątkowo kosztowne czasowo.
- **AVL był zaskakująco wolny przy unikalnych, losowych danych**, co może wynikać z częstego równoważenia drzewa. Pomimo lepszej złożoności w teorii, w praktyce narzut na balansowanie może być znaczący.
- **Wyszukiwanie w drzewach (BST/AVL)** wypadło znacznie lepiej niż w liście, ale zauważalnie gorzej niż w słowniku. Dla zastosowań wymagających wysokiej wydajności warto zatem rozważać `dict`, jeśli nie wymagamy struktury drzewa.