Paweł Drakoniewicz

Analiza drzew Splay i BST z zastosowaniem różnych rozkładów prawdopodobieństwa

Celem projektu jest eksperymentalne porównanie wydajności różnych typów drzew binarnych: **drzew Splay oraz zwykłych nierównoważonych drzew BST (Binary Search Tree).** Struktury te są szeroko stosowane w informatyce do efektywnego przechowywania i wyszukiwania danych, a ich wydajność może znacząco wpływać na szybkość działania aplikacji, w których są wykorzystywane.

Drzewa Splay to rodzaj drzew samoorganizujących się, które dynamicznie przekształcają swoją strukturę w odpowiedzi na operacje wyszukiwania. Kluczową cechą drzew Splay jest to, że po każdej operacji wyszukiwania, wstawienia lub usunięcia, drzewo przekształca się tak, aby często wyszukiwane elementy znajdowały się bliżej korzenia. Taka strategia ma na celu zoptymalizowanie dostępu do często używanych kluczy, co teoretycznie powinno prowadzić do zwiększenia wydajności w przypadkach, gdy niektóre klucze są wyszukiwane częściej niż inne.

Zwykłe nierównoważone drzewa BST (Binary Search Tree) to proste struktury danych, w których każdy węzeł ma co najwyżej dwoje dzieci, a wartości w lewym poddrzewie są mniejsze niż wartość w węźle, a wartości w prawym poddrzewie są większe. Drzewa BST zapewniają średnią złożoność operacji O(log n) dla wyszukiwania, wstawiania i usuwania, jednak w przypadku degeneracji (np. gdy wstawiane elementy są już posortowane), złożoność może wzrosnąć do O(n).

Plan projektu:

1. Implementacja **drzewa Splay** oraz **zwykłego drzewa BST.**
2. Porównanie czasów tworzenia drzew.
3. Przeprowadzenie eksperymentów polegających na wielokrotnym wyszukiwaniu kluczy w tych drzewach i porównanie czasów.
4. Przeanalizowanie wpływu różnych rozkładów prawdopodobieństwa na wydajność tych struktur.

Oto kod **drzewa Splay**:

```
class Node:
    def __init__(self, key):
        # Inicjalizacja węzła
        self.key = key  # Klucz węzła
        self.left = None  # Lewe poddrzewo
        self.right = None  # Prawe poddrzewo
class SplayTree:
    def __init__(self):
        # Inicjalizacja drzewa Splay
        self.root = None  # Korzeń drzewa
    def rotate_right(self, x):
        # Rotacja w prawo wokół węzła x
        y = x.left
        x.left = y.right
        y.right = x
        return y
    def rotate_left(self, x):
        # Rotacja w lewo wokół węzła x
        y = x.right
        x.right = y.left
        y.left = x
        return y
    def splay(self, key):
        # Operacja splay na węźle o kluczu key
        if self.root is None or self.root.key == key:
            return self.root
        dummy = Node(None)  # Węzeł pomocniczy
        left = right = dummy  # Wskaźniki pomocnicze
        cur = self.root  # Aktualny węzeł
        while True:
```

```
    if key < cur.key:
        # Wyszukiwanie klucza w lewym poddrzewie
        if cur.left is None:
            break
        if key < cur.left.key:
            # Rotacja w prawo
            cur = self.rotate_right(cur)
            if cur.left is None:
                break
        right.left = cur
        right = cur
        cur = cur.left
        right.left = None
    elif key > cur.key:
        # Wyszukiwanie klucza w prawym poddrzewie
        if cur.right is None:
            break
        if key > cur.right.key:
            # Rotacja w lewo
            cur = self.rotate_left(cur)
            if cur.right is None:
                break
        left.right = cur
        left = cur
        cur = cur.right
        left.right = None
    else:
        break
# Przepięcie poddrzew
left.right = cur.left
right.left = cur.right
```

```python
        cur.left = dummy.right
        cur.right = dummy.left
        self.root = cur
    def insert(self, key):
        # Wstawianie nowego klucza do drzewa
        if self.root is None:
            # Drzewo jest puste
            self.root = Node(key)
            return
        # Operacja splay dla węzła o kluczu key
        self.splay(key)
        if key < self.root.key:
            # Wstawianie klucza do lewego poddrzewa
            node = Node(key)
            node.left = self.root.left
            node.right = self.root
            self.root.left = None
            self.root = node
        elif key > self.root.key:
            # Wstawianie klucza do prawego poddrzewa
            node = Node(key)
            node.right = self.root.right
            node.left = self.root
            self.root.right = None
            self.root = node
    def search(self, key):
        # Wyszukiwanie klucza w drzewie
        self.splay(key)
        if self.root.key == key:
            return True
        return False
```

Kod **zwykłego nierównoważonego drzewa BST:**

```
import math
class Node2:
    def __init__(self, data = None, par = None):
        self.data = data
        self.left = self.right = None
        self.parent = par
class Tree:
    def __init__(self):
        self.dummy = Node('u')
        self.root = self.dummy.right
        # do korzenia dodajemy sztucznego rodzica
    def find(self, node, value):
        if node is None:
            return None, False
        if value == node.data:
            return node, True
        if value < node.data:
            if node.left:
                return self.find(node.left, value)
        if value > node.data:
            if node.right:
                return self.find(node.right, value)
        return node, False
    def append(self, obj):
        if self.root is None:
            self.root = obj
            self.root.parent = self.dummy
        s, fl_find = self.find(self.root, obj.data)
        if not fl_find and s:
            if obj.data < s.data:
                s.left = obj
```

```
            obj.parent = s
        else:
            s.right = obj
            obj.parent = s
def show_tree(self, node):
    if node is None:
        return
    self.show_tree(node.left)
    print(node.data, end = ",")
    self.show_tree(node.right)
def suma(self, node):
    if node is None:
        return 0
    return self.suma(node.left) + node.data + self.suma(node.right)
def count(self, node):
    if node is None:
        return 0
    return self.count(node.left) + 1 + self.count(node.right)
def height(self, node):
    if node is None:
        return 0
    l = self.height(node.left)
    r = self.height(node.right)
    if (l>r):
        return l+1
    # ja + lewa noga
    return r+1
    # ja + prawa noga
def mini(self, node):
    while node is None:
        return node
    while node.left:
        node =  node.left
    return node
```

```
def myk(self, node, depth = 0):
    # wyświetla drzewo tak, żeby wyglądało w miarę jak drzewo
    if node is None:
        return
    if node.right:
        self.myk(node.right, depth + 1)
    for _ in range(0, depth):
        print("    ", end = "")
    print(node.data)
    if node.left:
def del_leaf(self, node):
    if node.parent.left == node:
        node.parent.left = None
    if node.parent.right == node:
        node.parent.right = None
def del_one_child(self, node):
    if node.parent.right == node:
        if node.right:
            node.parent.right = node.right
        if node.left:
            node.parent.left = node.left
    elif node.parent.left == node:
        if node.left:
            node.parent.left = node.left
        if node.right:
            node.parent.right = node.right
def del_node(self, key):
    s, pl_find = self.find(self.root, key)
    if not pl_find:
        return
    if s.left is None and s.right is None:
        self.del_leaf(s)
    elif s.left is None or s.right is None:
        self.del_one_child(s)
```

```
    else:
        nd = self.mini(s.right)
        s.data = nd.data
        self.del_one_child(nd)
def Rotate(self,B):
    if (B == self.dummy or B == None or B == self.root):
        return
    A = B.parent
    P = A.parent
    #rotacja w prawo
    if A.left == B:
        B.parent = p
        if (P.right == A):
            P.right = B
        else:
            P.left = B
        beta = B.right
        B.parent = B
        B.right = A
        A.left = beta
        if beta:
            beta.append = A
        if self.root.parent != self.dummy:
            self.root = self.dummy.right
```

Funkcje, którymi zmierzymy czas tworzenia się drzew:

Funkcja **measure_BST_creation_time** mierzy czas potrzebny na utworzenie drzewa BST (Binary Search Tree) z liczb naturalnych w przedziale od 1 do 10000, dodawanych w losowej kolejności. Po utworzeniu drzewa wyświetla czas potrzebny na wykonanie tej operacji.

```
import random

def measure_BST_creation_time():
    # Tworzenie drzewa BST
    t = Tree()
    czas_poczatkowy = time.time()
    numbers = list(range(1, 10001))
    random.shuffle(numbers)
    for number in numbers:
        t.append(Node2(number))
    czas_koncowy = time.time()
    czas_tworzenia = czas_koncowy - czas_poczatkowy
    # Wyświetlenie czasu utworzenia drzewa
    print(f'Utworzono nierównoważone drzewo BST w {czas_tworzenia} sekund')
```

Funkcja measure_splay_creation_time mierzy czas potrzebny na utworzenie drzewa Splay z liczb naturalnych w przedziale od 1 do 10000, dodawanych w losowej kolejności. Po utworzeniu drzewa wyświetla czas potrzebny na wykonanie tej operacji.

```python
def measure_splay_creation_time():
    # Tworzenie drzewa Splay
    splay_tree = SplayTree()
    czas_poczatkowy = time.time()
    liczby = list(range(1, 10001))
    random.shuffle(liczby)
    for liczba in liczby:
        splay_tree.insert(liczba)
    czas_koncowy = time.time()
    czas_tworzenia = czas_koncowy - czas_poczatkowy
    # Wyświetlenie czasu utworzenia drzewa
    print(f'Utworzono drzewo Splay w {czas_tworzenia} sekund')
```

Funkcje, którymi zmierzymy czas wyszukiwania w drzewie losowej liczby z różnymi prawdopodobieństwami:

```
import random
def measure_BST_search_time(tree,rozklad_prawdopodobienstwa=None):
    zakres = range(1, 10001)
    if rozklad_prawdopodobienstwa:
        random_numbers = np.random.choice(zakres, 50000, p=rozklad_prawdopodobienstwa)
    else:
        random_numbers = np.random.randint(1, 10001, 50000)
    czas_poczatkowy_szukania = time.time()
    for random_number in random_numbers:
        result = tree.find(tree.root, random_number)
    czas_koncowy_szukania = time.time()
    # Obliczanie czasu wykonania testów
    czas_szukania = czas_koncowy_szukania - czas_poczatkowy_szukania
    # Wyświetlenie wyników wyszukiwania
    nazwa = nazwa_rozkładu(rozklad_prawdopodobienstwa)
    print(f"Dla rozkładu {nazwa}")
    print(f"Wyszukanie 50000 razy w drzewie losowej liczby zajeło {czas_szukania} sekund")
```

measure_splay_search_time(tree,rozkład_prawdopodobienstwa=None): mierzy czas wyszukiwania liczb w drzewie Splay przy użyciu określonego rozkładu prawdopodobieństwa.

```
def measure_splay_search_time(tree, rozklad_prawdopodobienstwa=None):
    zakres = range(1, 10001)
    if rozklad_prawdopodobienstwa:
        random_numbers = np.random.choice(zakres, 50000, p=rozklad_prawdopodobienstwa)
    else:
        random_numbers = np.random.randint(1, 10001, 50000)
    czas_poczatkowy_szukania = time.time()
    for random_number in random_numbers:
        result = tree.search(random_number)
    czas_koncowy_szukania  = time.time()
    # Obliczanie czasu wykonania testów
    czas_szukania = czas_koncowy_szukania - czas_poczatkowy_szukania
    # Wyświetlenie wyników wyszukiwania
    nazwa = nazwa_rozkładu(rozklad_prawdopodobienstwa)
    print(f"Dla rozkładu {nazwa}")
    print(f"Wyszukanie 50000 razy w drzewie losowej liczby zajeło {czas_szukania} sekund")
```

nazwa_rozkładu(rozklad): przyjmuje argument rozklad, który określa rozkład prawdopodobieństwa. Na podstawie przekazanego rozkładu zwraca odpowiednią nazwę w formie tekstowej*:*

```
def nazwa_rozkładu(rozklad):
    if rozklad is None:
        return "jednostajnego"
    elif rozklad == Rozkład_normalny:
        return "normalnego"
    elif rozklad == Rozkład_odwrotny:
        return "odwrotnego"
    elif rozklad == Rozkład_liniowy:
        return "liniowego"
    elif rozklad == Rozkład_kwadratowy:
        return "kwadratowego"
    elif rozklad == Rozkład_potęgowy:
        return "potęgowy"
```

Testowanie rozpoczynamy od wywoływania funkcji **measure_BST_creation_time()** oraz **measure_splay_creation_time(),** które mierzą czas tworzenia odpowiednio drzewa BST oraz drzewa Splay i wyświetlają wyniki tych pomiarów.

```
print('Zakres liczb: (1, 10000)')
measure_BST_creation_time()
measure_splay_creation_time()
```

Wyniki 5 prób:

Zakres liczb: (1, 10000)

Numer próby	Czas tworzenia drzewa Splay (sekundy)	Czas tworzenia nierównoważonego drzewa BST (sekundy)
1	0.11990	0.07680
2	0.11214	0.07837
3	0.11673	0.07319
4	0.11386	0.07642
5	0.11209	0.07319

Następnie definiujemy różne rozkłady prawdopodobieństw:

- Dla **rozkładu normalnego** ustawiamy średnią (mu = 5000) oraz odchylenie standardowe (sigma = 2000). Rozkład ten generowany jest przez listę **Rozkład_normalny** z wartościami wylosowanymi za pomocą funkcji **random.normalvariate(mu, sigma).** Następnie wartości te są ograniczane do przedziału [1, 10000], a po tym normalizowane, aby suma wszystkich prawdopodobieństw wynosiła 1.

  mu, sigma = 5000, 2000 # średnia i odchylenie standardowe
  Rozkład_normalny = [random.normalvariate(mu, sigma) for _ in range(10000)]
  Rozkład_normalny = [min(max(x, 1), 10000) for x in Rozkład_normalny] # Ograniczenie wartości do przedziału [1, 10000]
  Rozkład_normalny = [x / sum(Rozkład_normalny) for x in Rozkład_normalny] # normalizacja

- **Rozkład odwrotnie proporcjonalny do wartości** tworzony jest przez listę **Rozkład_odwrotny**, która zawiera wartości odwrotności liczb z przedziału [1, 10000]. Po obliczeniu tych wartości, są one również normalizowane.

  Rozkład_odwrotny = [1/i for i in range(1, 10001)]
  Rozkład_odwrotny = [x / sum(Rozkład_odwrotny) for x in Rozkład_odwrotny] # normalizacja

- **Rozkład liniowy** generowany jest przez listę **Rozkład_liniowy**, która zawiera liczby od 1 do 10000. Wartości te są następnie normalizowane.

  Rozkład_liniowy = [i for i in range(1, 10001)]
  Rozkład_liniowy = [x / sum(Rozkład_liniowy) for x in Rozkład_liniowy] # normalizacja

- **Rozkład kwadratowy** tworzony jest przez listę **Rozkład_kwadratowy**, zawierającą kwadraty liczb z przedziału [1, 10000], które również są normalizowane.

```
Rozkład_kwadratowy = [i**2 for i in range(1, 10001)]
Rozkład_kwadratowy = [x / sum(Rozkład_kwadratowy) for x in Rozkład_kwadratowy]  # normalizacja
```

- **Rozkład potęgowy** o stałym wykładniku k, gdzie k = 6, generowany jest przez listę **Rozkład_potęgowy.** Lista ta zawiera wartości odwrotności kolejnych liczb z przedziału [1, 10000], podniesionych do potęgi k. Po obliczeniu tych wartości, są one również normalizowane, aby sumować się do jedności.

```
k = 6
Rozkład_potęgowy = [1/i**k for i in range(1, 10001)]
Rozkład_potęgowy = [x / sum(Rozkład_potęgowy) for x in Rozkład_potęgowy]  # normalizacja
```

Tworzymy liste różnych rozkładów prawdopodobieństwa:

```
rozklad_prawdopodobienstwa = [
    None,  # Jednostajny rozkład prawdopodobieństwa
    Rozkład_normalny,
    Rozkład_odwrotny,
    Rozkład_liniowy,
    Rozkład_kwadratowy,
    Rozkład_potęgowy
]
```

Tworzymy najpierw nowe drzewo BST. Lista liczb od 1 do 10000 jest losowo tasowana, a następnie każda liczba dodawana jest do drzewa za pomocą metody append(). Funkcja **measure_BST_search_time(t, rozklad)** mierzy czas wyszukiwania 50000 losowych liczb w drzewie BST i wyświetla wyniki tych testów:

```
print('Dla drzewaBST:')
 t = Tree()
numbers = list(range(1,10001))
random.shuffle(numbers)
 for number in numbers:
   t.append(Node2(number))
for rozklad in rozklad_prawdopodobienstwa:
   measure_BST_search_time(t,rozklad)
   print('\n')
```

Po przeprowadzeniu testów dla drzewa BST, tworzymy drzewo Splay. Lista liczb od 1 do 10000 jest losowo tasowana, a każda liczba dodawana jest do drzewa Splay za pomocą funkcji insert(). Następnie dla każdego rozkładu prawdopodobieństwa, funkcja **measure_splay_search_time(root, rozklad)** mierzy czas wyszukiwania 50000 losowych liczb w drzewie Splay i wyświetla wyniki tych testów.

```
root_splay = None
liczby = list(range(1, 10001))
random.shuffle(liczby)
splay_tree = SplayTree()
for liczba in liczby:
   splay_tree.insert(liczba)
print('Dla drzewa splay:')
for rozklad in rozklad_prawdopodobienstwa:
   measure_splay_search_time(splay_tree, rozklad)
   print('\n')
```

Wyniki dla 3 prób:

1.

Rozkład	**Drzewo Splay (sekundy)**	**Drzewo BST (sekundy)**
Jednostajny	0.45986	0.31968
Normalny	0.47468	0.31984
Odwrotny	0.33407	0.25300
Liniowy	0.44540	0.32115
Kwadratowy	0.43029	0.32239
Potęgowy	0.03562	0.18907

2.

Rozkład	**Drzewo Splay (sekundy)**	**Drzewo BST (sekundy)**
Jednostajny	0.54344	0.37989
Normalny	0.52260	0.37670
Odwrotny	0.38899	0.27639
Liniowy	0.53030	0.38996
Kwadratowy	0.52655	0.37721
Potęgowy	0.03771	0.15520

3.

Rozkład	**Drzewo Splay (sekundy)**	**Drzewo BST (sekundy)**
Jednostajny	0.49915	0.36920
Normalny	0.49118	0.37030
Odwrotny	0.36491	0.24094
Liniowy	0.48900	0.37693
Kwadratowy	0.46985	0.36635
Potęgowy	0.03601	0.19647

Wnioski:

- **Czas tworzenia drzewa BST i Splay:**

  Czas tworzenia drzewa Splay jest dłuższy niż czas tworzenia drzewa BST. Wynika to z faktu, że drzewo Splay wymaga dodatkowych operacji rotacji, które są dla niego charakterystyczne i mają na celu zapewnienie jego samoorganizującej się natury. Prowadzi to do wydłużenia czasu konstrukcji drzewa, co może mieć znaczenie w przypadku dużych zbiorów danych.

- **Czas wyszukiwania:**

  Drzewo Splay:
  Dla wszystkich rozkładów prawdopodobieństwa (oprócz potęgowego), czas wyszukiwania w drzewie Splay jest dłuższy niż w drzewie BST. Wynika to z faktu, że drzewo Splay automatycznie dostosowuje się do często wyszukiwanych elementów poprzez przenoszenie ich na górne poziomy drzewa, co może prowadzić do dłuższych czasów wyszukiwania.

  Drzewo BST:
  Dla większości rozkładów, czas wyszukiwania w drzewie BST jest krótszy niż w drzewie Splay. Jest to spowodowane brakiem automatycznego dostosowywania się drzewa BST do często wyszukiwanych elementów.

- **Podsumowanie:**
  Drzewo Splay jest bardziej skuteczne w sytuacjach, gdy niektóre klucze są wyszukiwane znacznie częściej niż inne (**rozkład potęgowy**), ponieważ dostosowuje się ono do często wyszukiwanych elementów poprzez operacje rotacji. Jednakże, dla ogólnych przypadków, drzewo BST może być bardziej efektywne ze względu na krótszy czas wyszukiwania.