Sprawozdanie - Lista 4

Algorytmy i Struktury Danych

Józef Piechaczek

2019-05-19

1 Testy wydajności poszczególnych drzew

Test rozpocznę od wczytania pliku <tekstowego i umieszczeniu słów w kolejności wczytywania w *ArrayList*. W zależności od testu przetasuję wczytane słowa. Testy wykonam dla 100 prób, gdzie każda próba składa się z:

1. Wykonania *insert* dla każdego elementu z ciągu
2. Wykonania *search* dla każdego elementu z ciągu
3. Wykonania *delete* dla każdego elementu z ciągu

Wynikiem testu są informacje odnośnie:

1. Średniego czasu działania operacji *insert*, *search*, *delete* dla całego ciągu
2. Liczba porównań dla pojedynczej próby
3. Liczba modyfikacji dla pojedynczej próby

1.1 Ciąg unikatowych posortowanych elementów

W tym teście posłużę się plikiem *aspell_wordlist.txt* zawierającym 125975 unikatowych posortowanych elementów, bez przetasowania.

1.1.1 Koszty

Wyniki dla konkretnych drzew przedstawia poniższa tabela:

Test 1

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	65700	75515	6	1020490709	377925
Splay	7	31	22	9652972	6672843
RBT	62	44	46	22304203	2833929

1.1.2 Analiza wyników

Koszty dla drzewa BST, choć bardzo wysokie, wydają się poprawne. Wysoki koszt operacji insert i search, oraz wysoka liczba porównań, wynikają z faktu, iż dla każdego następnego elementu, wierzchołek umieszczany w drzewie staje się prawym synem największego wierzchołka. Drzewo budowane w ten sposób jest skrajnie niezrównoważone i sprowadzone do listy, a koszt wykonania operacji insert i search wzrasta do $O(n)$. Czas wykonania operacji delete jest bardzo niski, ponieważ dla każdego elementu, który próbujemy usunąć znajduje się on w korzeniu.

W przypadku drzew Splay oraz RBT można zauważyć znacząco niższe koszty operacji insert i search oraz znacząco mniejsza liczbę porównań. Wynika to z faktu, iż drzewa te są modyfikowane w czasie wykonywania operacji, co widoczne jest również w liczbie modyfikacji.

W powyższym teście drzewo Splay wypada najlepiej, gdyż ostatni wyszukiwany/modyfikowany element umieszczany jest w korzeniu. W przypadku danych wejściowych składających się z posortowanych elementów jest to bardzo korzystne, gdyż gwarantuje, że kolejny element będzie znajdował się blisko korzenia, co wpływa na niski koszt operacji.

1.2 Ciąg unikatowych nieposortowanych elementów

W tym teście posłużę się plikiem *aspell_wordlist.txt* zawierającym 125975 unikatowych posortowanych elementów, uprzednio wykonując przetasowanie listy zawierającej słowa.

1.2.1 Koszty

Wyniki dla konkretnych drzew przedstawia poniższa tabela:

Test 2

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	80	80	61	18620917	507991
Splay	295	277	293	79863658	59462792
RBT	95	87	96	18518252	2429433

1.2.2 Analiza wyników

W powyższym teście najmniejsze koszta uzyskaliśmy dla drzew BST oraz RBT. Wynika to z faktu, iż drzewo BST dla nieposortowanych danych, będzie miało niską średnią wysokość, co wpływa na niski czas wykonania operacji. Drzewo RBT jako samoorganizująca się struktura również ma niski koszt poszczególnych operacji, jednak w tym konkretnym przypadku wyższy niż BST, gdyż wymaga ono dodatkowo wykonywania rotacji i przekolorowywania. Drzewo Splay ma najwyższy koszt wykonywania operacji, gdyż każda z nich wymaga wykonania funkcji splay, która w wyniku rotacji umieszcza element ostatnio wyszukiwany w korzeniu. Ponieważ dane są nieposortowane, funkcja splay jest dosyć kosztowna, gdyż element, którego szukamy, może znajdować się daleko od korzenia, co powoduje wysoką liczbę koniecznych do wykonania rotacji.

1.3 Ciąg nieposortowanych elementów z powtórzeniami

W tym teście posłużę się plikiem *KJB.txt* zawierającym ciąg 821108 nieposortowanych elementów z powtórzeniami, bez przetasowania.

1.3.1 Koszty

Wyniki dla konkretnych drzew przedstawia poniższa tabela:

Test 3

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	131	131	138	67688454	55888
Splay	438	419	502	173272421	121741758
RBT	157	205	247	70605074	276405

1.3.2 Analiza wyników

W przypadku próby wykonania operacji insert lub delete dla nieistniejącego elementu drzewa BST i RBT zachowują się jednakowo - przeszukują drzewo, aż trafią na null/nil, po czym kończą operację. Drzewa Splay zachwoują się podobnie, jednak dodatkowo wykonują operację splay na ostatnim węźle drzewa odwiedzonym przy wyszukiwaniu. Zatem wyniki dla ciągu nieposortowanych elementów z powtórzeniami są zbliżone do wyników dla ciągu bez powtórzeń.

2 Przykłady optymalnych danych dla poszczególnych drzew

2.1 Splay

Optymalnymi danymi wejściowymi dla drzewa Splay może być słownik języka angielskiego dostępny w systemie Linux. Zawiera on ciąg posortowanych alfabetycznie wyrazów.

Drzewa Splay najlepiej radzą sobie w przypadku uporządkowanych elementów. Ze względu na to, iż każda operacja wykonywana na drzewie wymaga wywołania funkcji splay, ostatnio odwiedzony przy wyszukiwaniu węzeł znajduje się w korzeniu. Dzięki temu koszt wykonywania operacji na kolejnych elementach jest niski - poszukiwana wartość znajduje się w niewielkiej odległości od korzenia. W przypadku stosowania drzewa Splay dla losowych ciągów koszty operacji są większe niz dla drzew RBT.

Drzewa Splay pozwalają na wykonywanie szybkich operacji na często modyfikowanych/wyszukiwanych węzłach. Mogą zatem znaleźć zastosowanie np. w implementacji pamięci cache.

Wyniki dla podanych danych:

Test 4

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	35770	44050	4	445876232	307203
Splay	5	24	18	7846664	5424195
RBT	48	36	39	17860544	2303530

2.2 BST

Optymalnymi danymi wejściowymi dla drzewa BST może być książka *Algoruthms Unlocked* Thomasa H. Cormena. Zawiera ona ciąg wyrazów, o którym mamy pewność, że jest nieposortowany.

Drzewa BST w większości przypadków radzą sobie dobrze, jednak w pesymistycznym przypadku, dla posortowanych danych, tworzą one listę - drzewo skrajnie niezrównoważone, a koszty operacji wynoszą $O(n)$. Drzewa BST powinniśmy zatem używać gdy mamy gwarancję że dane wejściowe są w wysokim stopniu nieposortowane.

Plusem drzew BST jest fakt, iż nie musza one przechowywać wielu informacji, jak np. informacja o rodzicu, czy kolor, co wpływa na niższe użycie pamięci. Drzewa te również są łatwiejsze w implementacji.

Wyniki dla podanych danych:

Test 5

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	14	16	12	7250496	19909
Splay	41	38	49	16990137	11932416
RBT	16	19	20	6722751	97985

2.3 RBT

Optymalnymi danymi wejściowymi dla drzewa RBT może być plik połączony z 1800 posortowanych słów z słownika języka angielskiego oraz z książki *Algoruthms Unlocked* Thomasa H. Cormena.

Drzewa RBT najlepiej radzą sobie z losowymi danymi, nawet jeśli zawierają posortowane ciągi danych, która powodują znaczne opóźnienie dla drzew BST. Sa szybsze niż drzewa Splay dla losowych danych, ponieważ wymagają mniejszej ilości modyfikacji.

Drzewa RBT są zatem strukturą, którą powinno się stosować w domyślnych przypadkach. Są one jednak trudniejsze w implementacji oraz wymagają przechowywania dodatkowych informacji.

Wyniki dla podanych danych:

Test 6

	insert[ms]	search[ms]	delete[ms]	porówania	modyfikacje
BST	320	405	20	362697682	27050
Splay	59	53	59	17164026	12050549
RBT	39	29	24	8978412	148187