Sprawozdanie
AiSD lista 4

Łukasz Bratos

maj 2019

1 Wstęp

Celem listy było zaimplementowanie oraz przetestowanie następujących struktur danych:

- BST (drzewo poszukiwań binarnych)
- RBT (drzewo czerwono - czarne)
- Drzewo Splay (samoorganizujące drzewo binarne)

Struktury zostały zaimplementowane w języku Java.

2 Drzewa

2.1 BST

Drzewa poszukiwań binarnych, w skrócie BST (ang. Binary Search Trees), są strukturami na których można wykonywać różne operacje właściwe dla zbiorów dynamicznych, takie jak SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT oraz DELETE. Drzewo poszukiwań może więc być użyte zarówno jako słownik, jak i jako kolejka priorytetowa.

Podstawowe operacje na drzewach poszukiwań binanrnych wymagają czasu proporcjonalnego do wysokości drzewa. W pełnym drzewie binarnym o n węzłach takie operacje działają w najgorszym przypadku w czasie $O(\lg n)$. Jeśli jednak drzewo składa się z jednej gałęzi o długości n, to te same operacje wymagają w pesymistycznym przypadku czasu $O(n)$.

Klucze są przechowywane w drzewie BST w taki sposób, aby spełniona była własność drzewa BST:

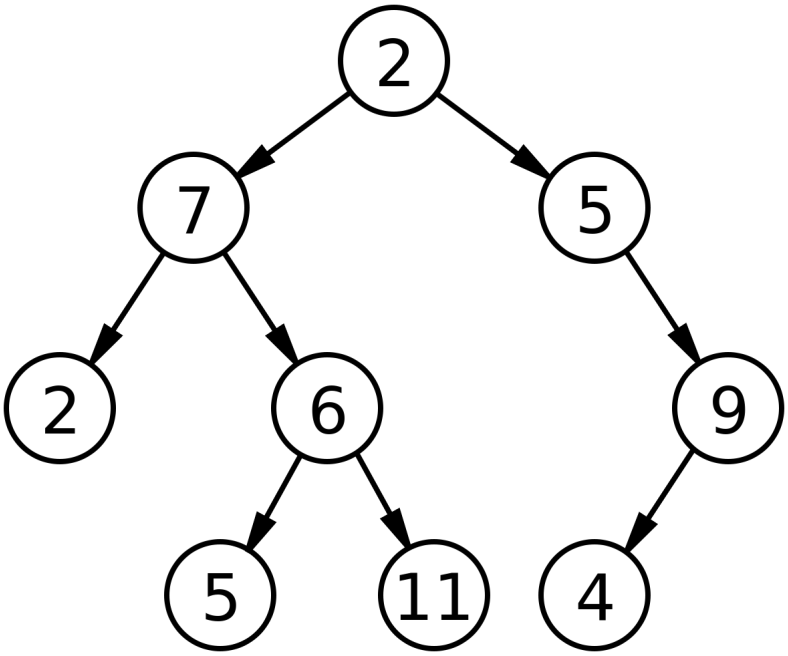
Definicja 1. Niech x będzie węzłem drzewa BST. Jeśli y jest węzłem znajdującym się w lewym poddrzewie węzła x, to $y.key \leqslant x.key$. Jeśli y jest węzłem znajdującym się w prawym poddrzewie węzła x, to $x.key \leqslant y.key$.

2.2 RBT

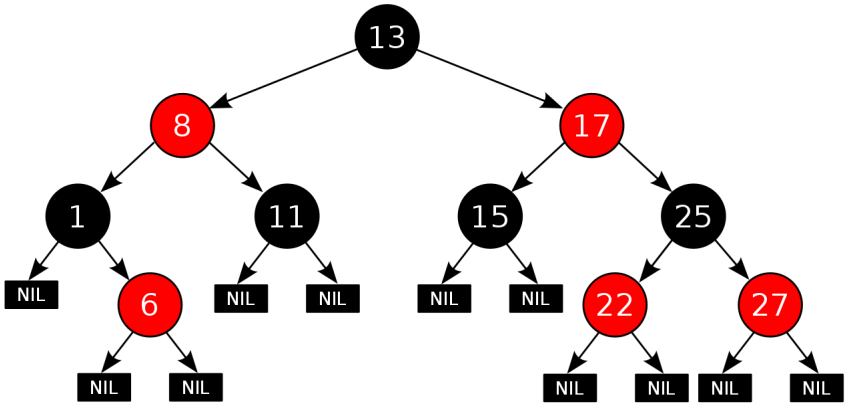
Drzewo czerwono - czarne jest drzewem poszukiwań binarnych, w którym na każdy węzeł przypada jeden dodatkowy bit informacji: jego kolor, który może być albo czerwony (RED), albo czarny (BLACK). Przez narzucenie odpowiednich warunków na możliwe ciągi kolorów węzłów leżących na dowolnej ścieżce biegnącej od korzenia do liścia drzewa czerwono - czarnego gwarantujemy, że każda ścieżka jest co najwyżej dwa razy dłuższa niż dowolna inna, dzięki czemu drzewo jest w przybliżeniu zrównoważone.

Drzewo BST jest drzewem czerwono - czarnym, jeśli ma następujące własności czerwono - czarne:

1. każdy węzeł jest albo czerwony, albo czarny;
2. korzeń jest czarny;
3. każdy liść (NIL) jest czarny;
4. jeśli węzeł jest czerwony, to obaj jego synowie są czarni;
5. każda prosta ścieżka z ustalonego węzła do liścia ma tyle samo czarnych węzłów.



Rysunek 1: Przykładowe drzewo poszukiwań binarnych



Rysunek 2: Przykładowe drzewo czerwono - czarne

2.3 Splay

Drzewo splay to struktura danych w formie samodostosowującego się drzewa poszukiwań binarnych. Wykonywanie podstawowych operacji na drzewie splay wiąże się z wykonaniem procedury $Splay(T, x)$, która powoduje taką zmianę struktury drzewa T, że węzeł x zostaje umieszczony w korzeniu przy zachowaniu porządku charakterystycznego dla drzewa BST.

W porównaniu do innych drzew BST, drzewa splay zmieniają swoją strukturę również podczas wyszukiwana kluczy (a nie tylko dodawania lub usuwania), przesuwając znaleziony węzeł w kierunku korzenia, dzięki temu często wyszukiwane węzły stają się szybsze do znalezienia. Z tego powodu drzewa splay bywają wykorzystywane w systemach typu cache. Drzewa splay nie są samorównoważące, ponieważ ich wysokość nie jest ograniczona przez $O(\lg n)$ – można np. tak wykonać operacje, że drzewo zdegeneruje się do listy.

3 Testy

3.1 Metodologia

Testy polegały na wczytaniu wszystkich słów w pliku, wyszukaniu ich, a następnie ich usunięciu. Dla każdego drzewa w każdym teście mierzone były czas działania, liczba porównań oraz liczba modyfikacji węzłów dla operacji *insert, search, delete*. Testy był wykonywane 100 razy dla każdego pliku. Wynik to średnia arytmetyczna ze wszystkich 100 testów.

Testy były przeprowadzone na następujących plikach:

- lotr.txt - plik z powtórzeniami
- aspell.txt - ciąg znaków w porządku leksykograficznym
- KJB.txt - duży plik z powtórzeniami
- permutation.txt - dla każdego testu losowa permutacja pliku lotr.txt

3.2 Wyniki

Tabela 1: Wyniki dla pliku lotr.txt

	Operacja	BST	RBT	Splay
Czas [s]	Insert	5,793	0,105	0,108
	Search	0,085	0,0522	0,078
	Delete	0,069	0,088	0,140
Porównania	Insert	317 576 165	10 880 514	28 878 615
	Search	12 573 838	5 575 015	20 186 563
	Delete	9 509 763	9 794 607	26 171 327
Modyfikacje	Insert	3 379 275	3 193 751	14 580 045
	Search	0	0	9 162 265
	Delete	967 705	2 772 111	10 115 982

Tabela 2: Wyniki dla pliku aspell.txt

	Operacja	BST	RBT	Splay
Czas [s]	Insert	179,241	0,056	0,035
	Search	51,943	0,055	0,074
	Delete	0,039	0,051	0,06
Porównania	Insert	15 869 952 574	8 918 531	1 637 666
	Search	233 804 613 925	6 059 926	6 135 476
	Delete	755 850	7 199 810	5 615 808
Modyfikacje	Insert	251 949	2 078 210	755 845
	Search	0	0	2 492 963
	Delete	125 974	1 007 622	2 105 212

4 Wnioski

Na podstawie testów możemy wywnioskować jak zachowują się poszczególne struktury oraz skonfrontować to z teorią. Pierwszą obserwacją jest to jak operacja insert w BST odstaje od pozostałych struktur. Może to być spowodowane brakiem jakiejkolwiek samoorganizacji, przez co złożoność czasowa, która jest proporcjonalna do wysokości drzewa, w pesymistycznym przypadku jest liniowa. Widać to dobrze w przypadku posortowanego pliku gdzie liczba porównań dla operacji insert oraz search gwałtownie wzrasta.

RBT wypada bardzo dobrze w każdych testach niezależnie od pliku.

Samoorganizacja drzewa splay znacząco przyspiesza operację insert w stosunku do BST. Operacja search w przypadku gdy dane są posortowane również przyspiesza. To wszystko niestety odbywa się kosztem operacji delete, która w każdym z testowanych przez nas przypadków działa wolniej niż w BST.

Tabela 3: Wyniki dla pliku KJB.txt

	Operacja	BST	RBT	Splay
Czas [s]	Insert	268,383	0,560	0,486
	Search	0,387	0,242	0,326
	Delete	0,322	0,355	3,922
Porównania	Insert	12 628 705 658	582 023 342	116 327 704
	Search	51 918 408	24 224 577	710 933 137
	Delete	386 339 751	45 245 189	531 003 874
Modyfikacje	Insert	1 704 605	145 633 628	58 495 699
	Search	0	0	31 892 857
	Delete	4 343 619	12 718 738	36 817 366

Tabela 4: Wyniki dla losowych permutacji pliku lotr.txt

	Operacja	BST	RBT	Splay
Czas [s]	Insert	4,902	0,106	0,115
	Search	0,058	0,046	0,074
	Delete	0,058	0,078	0,132
Porównania	Insert	312 359 747	10 780 871	29 772 722
	Search	7 815 644	5 603 440	21 602 571
	Delete	8 292 971	9 787 674	27 802 732
Modyfikacje	Insert	378 066	3 148 830	15 052 097
	Search	0	0	9 836 913
	Delete	991 138	2 752 475	10 794 134

4.1 Preferowane rodzaje danych

4.1.1 BST

BST najgorzej współgra z danymi posortowanymi - drzewo jest wtedy degenerowane do listy. Aby temu zapobiec dane powinny być podawane w takiej kolejności aby wynikowe drzewo było jak najbardziej zbalansowane. Pozwoli to uzyskać rzeczywistą złożoność logarytmiczną. Dane takie można uzyskać sczytując je wiersz po wierszu z drzewa RBT.

Tabela 5: Czasy [s] dla najlepszych danych dla BST

Operacja	BST	RBT	Splay
Insert	0,070	0,065	0,022
Search	0,034	0,039	0,036
Delete	0,032	0,035	0,024

4.1.2 RBT

Z testów wynika, że RBT spisuje się świetnie dla każdych danych. Najmniej porównań oraz modyfikacji wykonuje jednak na danych posortowanych.

4.1.3 Splay

Ze względu na specyfikę funkcji splay, która jest wywoływana po każdej operacji i przesuwa dane o które pytaliśmy w górę drzewa, możemy spodziewać się drzewo splay najlepiej będzie się zachowywać w przypadku częstego pytania o te same dane.

5 Bibliografia

1. *Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*

2. *pl.wikipedia.org/wiki/Drzewo_splay*