Autor: Kamil Król
Numer indeksu: 244949

Sprawozdanie

Drzewa przeszukiwań binarnych

1. Wprowadzenie

Zostały przeprowadzone eksperymenty polegające na wstawianiu, wyszukiwaniu i usuwaniu elementów dla trzech typów drzew przeszukiwań binarnych. Są to następujące drzewa:

- binary search tree (BST),
- splay tree (tzw. drzewo rozchylane),
- red black tree (RBT).

Badane były takie rzeczy jak ilość poszczególnych operacji, ilość porównań kluczy, czasy działania etc. Wszystkie eksperymenty zostały przeprowadzone dla różnych zestawów danych tzn. zarówno dla dużych jak i małych, posortowanych i losowych zestawów danych.
Celem sprawozdania jest przedstawienie i analiza wyników przeprowadzonych eksperymentów.

2. Złożoności

2.1. Binary search tree

Dla tego drzewa w optymistycznym przypadku wysokość drzewa wynosi log(n) gdzie n jest liczbą elementów. W związku z tym w optymistycznym przypadku złożoność operacji insert, search i delete wynosi O(log(n)). Jednak dla BST łatwo jest znaleźć złośliwy przypadek, który zwiększy złożoność. Drzewa BST można łatwo zdegradować do listy wstawiając posortowane elementy. Wtedy złożoność wszystkich operacji to O(n) – jest to pesymistyczna złożoność.

2.2. Splay tree

Splay tree jest modyfikacją drzewa BST. Nazywane jest również drzewem rozchylanym. Strukturę tą nadal można zdegradować do listy, a więc pesymistyczna złożoność wszystkich operacji wynosi nadal O(n). Różnica polega na tym, że koszt zamortyzowany dla wszystkich operacji wynosi O(log(n)).

2.3. Red black tree

Drzewo czerwono czarne jako jedyne spośród wymienionych jest drzewem samorównoważącym się. Równoważenie polega na tym, że odległość z korzenia do najbardziej oddalonego liścia nigdy nie jest większa niż dwukrotność odległości z korzenia do najmniej oddalonego liścia. Stąd złożoności wszystkich operacji wynoszą O(log(n)).

3. Eksperymenty

Eksperymenty polegały na ładowaniu, wyszukiwaniu, a następnie usuwaniu każdego słowa z następujących plików (kolejność według rozmiaru): sample.txt, aspell_wordlist.txt, lotr.txt, kjb.txt oraz z plików utworzonych poprzez ich posortowanie (?-sorted.txt) i dla aspell_wordlist.txt losowe poprzestawianie (aspell_wordlist-shuffled.txt). aspell_wordlist.txt jest już posortowany.

Wartości w kolejnych wierszach pod etykietą drzewa to: ilość porównań, ilość modyfikacji węzłów, czas działania. Wartości po lewej to wartości dla całej serii operacji, wartości po prawej to wartości średnie dla jednej operacji.

sample.txt						
	BST		splay		RBT	
insert	273	4.96	252	4.58	195	3.54
	32	0.58	1301	23.6	231	4.2
	51ms	0.92ms	41ms	0.74ms	37ms	0.67ms
search	305	5.54	525	9.54	214	3.89
	0	0.0	1542	28.0	0	0
	2ms	0.03ms	4ms	0.07ms	4ms	0.07ms
delete	160	2.9	308	5.6	190	3.45
	149	2.7	696	12.6	281	5.10
	1ms	0.02ms	2ms	0.04ms	1ms	0.02ms

sample-sorted.txt						
	BST		splay		RBT	
insert	1025	18,636	54	0,982	316	5,745
	32	0,582	187	3,400	341	6,200
	39ms	0,709ms	35	0,636ms	39	0,582ms
search	1057	19,218	319	5,800	272	4,945
	0	0,000	658	11,964	0	0,000
	3ms	0,055ms	2	0,036ms	4	0,036ms
delete	55	1,000	324	5,891	158	2,873
	119	2,164	566	10,291	225	4,091
	1ms	0,018ms	2	0,036ms	2	0,018ms

aspell_wordlist.txt						
	BST		splay		RBT	
insert	739158530	4602,109	495639	3,086	4139352	25,772
	91439	0,569	2401823	14,954	1434424	8,931
	18561	0,116ms	668	0,004ms	639	0,004ms
search	739249969	4602,678	1231040	7,665	2469162	15,373
	0	0,000	2964904	18,460	0	0,000
	7856	0,049ms	268	0,002ms	317	0,002ms
delete	121046603	753,654	862455	5,370	2341271	14,577
	444615	2,768	965735	6,013	829660	5,166
	1103	0,007ms	275	0,002ms	309	0,002ms

aspell_wordlist-shuffled.txt						
	BST		splay		RBT	
insert	2850111	17,745	2760740	17,189	2243724	13,970
	91439	0,569	15954797	99,337	764497	4,760
	520	0,003ms	1003	0,006ms	993ms	0,006ms
search	2941550	18,315	4581532	28,525	2298315	14,310
	0	0,000	16813277	104,682	0	0,000
	375	0,002ms	824	0,005ms	381ms	0,002ms
delete	3066866	19,095	4702071	29,276	1405897	8,753
	440261	2,741	10983511	68,385	303184	1,888
	322	0,002ms	401	0,002ms	594ms	0,004ms

lotr.txt						
	BST		splay		RBT	
insert	2453893	12,886	2004833	10,528	1928073	10,125
	9898	0,052	10878542	57,126	85209	0,447
	471ms	0,002ms	914ms	0,005ms	688ms	0,004ms
search	2463791	12,938	3135920	16,467	1973397	10,363
	0	0,000	10967475	57,593	0	0,000
	245ms	0,001ms	407ms	0,002ms	276ms	0,001ms
delete	2523350	13,251	3944975	20,716	2529797	13,285
	220649	1,159	861562	4,524	71862	0,377
	184ms	0,001ms	404ms	0,002ms	269ms	0,001ms

lotr-sorted.txt						
	BST		splay		RBT	
insert	1015554453	5336,906	190288	1,000	4081841	21,451
	9855	0,052	59125	0,311	132766	0,698
	15772ms	0,083ms	591ms	0,003ms	407ms	0,002ms
search	1015564308	5336,958	457668	2,405	2497845	13,127
	0	0,000	241124	1,267	0	0,000
	5405ms	0,028ms	398ms	0,002ms	271ms	0,001ms
delete	190284	1,000	2759066	14,499	2009573	10,561
	209999	1,104	211563	1,112	78731	0,414
	90ms	0,000ms	254ms	0,001ms	158ms	0,001ms

kjb.txt						
	BST		splay		RBT	
insert	10297076	12,057	7518345	8,803	8638608	10,115
	13856	0,016	39814691	46,619	122340	0,143
	1050ms	0,001ms	1955ms	0,002ms	1449ms	0,002ms
search	10310932	12,073	11797729	13,814	8940498	10,468
	0	0,000	39946694	46,774	0	0,000
	909ms	0,001ms	879ms	0,001ms	984ms	0,001ms
delete	10335645	12,102	19149518	22,422	11817864	13,838
	896506	1,050	1199140	1,404	95393	0,112
	528ms	0,001ms	606ms	0,001ms	748ms	0,001ms

kjb-sorted.txt						
	BST		splay		RBT	
insert	6883054214	8693,491	791747	1,000	18373667	23,206
	13603	0,017	81613	0,103	183350	0,232
	48395ms	0,061ms	810ms	0,001ms	975ms	0,001ms
search	6883067817	8693,508	1689910	2,134	11143788	14,075
	0	0,000	332844	0,420	0	0,000
	31529ms	0,040ms	1035ms	0,001ms	1052ms	0,001ms
delete	791748	1,000	7576720	9,570	8381018	10,585
	818954	1,034	292039	0,369	108706	0,137
	355ms	0,000ms	594ms	0,001ms	515ms	0,001ms

4. Analiza wyników eksperymentów

4.1 Binary search tree

Dla drzewa BST różnica jest widoczna już na samym początku. Nawet dla tak małego zestawu danych jak sample.txt widać, że w przypadku kiedy zestaw danych jest posortowany liczba porównań przy operacji insert wzrasta około 4-krotnie. Podobnie przy operacji search.
Poważna różnica jest jednak dopiero widoczna dla zestawu aspell_wordlist.txt. Liczba porównań przy insercie i search wynosi aż 739 158 530 dla zestawu posortowanego i
2 850 111 dla losowego rozstawienia elementów. Jest to około 260 razy mniej porównań. Czas insertów w pierwszym przypadku wynosi aż 18 sekund gdy w drugim tylko 0,5 sekundy. Średnie czasy wszystkich operacji są znacznie mniejsze w przypadku nieposortowanego ciągu.
Bardzo podobnie jest w przypadku zestawienia wyników dla zestawów lotr.txt i kjb.txt. W przypadku kjb.txt czasy inserów różnią się około 46-krotnie. Ponadto widać, że liczba zmian węzłów nie zależała od tego czy zbiór był posortowany.
Na uwagę zasługują jeszcze wyniki operacji delete. Niskie średnie czasy usuwania (w szczególności średni czas 0,000ms dla jednego z doświadczeń) i średnia liczba porównań oscylująca blisko 1,00 w przypadku zestawów posortowanych są wynikiem tego, że elementy w niektórych zestawach danych się powtarzają (kjb.txt, lotr.txt) i w posortowanym ciągu wynik dają bardzo szybko (na ogół po jednym porównaniu), ponieważ znajdują się w korzeniu

drzewa (mowa tylko o ciągu usunięć). Np. dla pliku nieposortowanego jak aspell_wordlist-shuffled.txt tej anomalii nie widać.

4.2 Splay tree

W splay tree dla zestawu aspell wordlist.txt (zestaw bez powtórzeń) od razu widać znaczną różnicę w liczbie porównań dla wszystkich operacji. BST ma 1400 razy więcej porównań dla inserta, 600 razy więcej w przypadku search i 150 razy więcej w przypadku delete. Jednak gdy ten zestaw danych jest losowo spermutowany sytuacja jeśli chodzi o ilość porównań jest odwrotna. Nie tylko ilość porównań jest większa dla splay tree dla zestawu losowego bez powtórzeń, ale również ilość modyfikacji węzłów i czas. W tej sytuacji splay tree jest nieefektywne.

Zestaw danych lotr.txt i kjb.txt zawierają powtórzenia. W przypadku gdy te dwa zestawy nie są posortowane, Splay wypada gorzej – jest wolniejsze około dwukrotnie dla lotr.txt i nieznacznie wolniejsze dla kjb.txt. W przypadku kiedy są posortowane, średnie czasy są znacznie lepsze dla splay tree. Podobnie jest z ilością porównań. Np. dla operacji search dla kjb-sorted.txt średnia ilość porównań na jedną operacje w BST jest 4000 razy większa niż w Splay tree. BST wykonuje w sumie 6 883 067 817 porównań, gdy Splay 1 689 910.

4.3 Red black tree

Drzewo czerwono czarne w żadnym z testów nie wypada bardzo źle (tak jak np. BST). Wyniki dane przez RBT okazały się w dużym stopniu niezależne od zestawów. Dla zestawu aspell wordlist.txt (uporządkowany, bez powtórzeń) RBT wypadło podobnie do splay tree. Dla tego samego zestawu spermutowanego losowo okazało się mieć najmniejszą liczbę porównań we wszystkich operacjach. Jeśli chodzi o czasy działania i inne parametry RBT w tym przypadku (będącym przypadkiem idealnym dla BST) wypadło mniej efektywnie, lecz nie była to duża różnica.

Dla zestawu lotr.txt (nieposortowanego) RBT wypada bardzo podobnie do BST. Czas wszystkich insertów jest 1,5 razy większy dla RBT niż w BST, jednak średni czas pojedynczego inserta jest 2 razy większy dla RBT. Liczba porównań była jednak mniejsza, Dla search zarówno czas całkowity jak i średni wypadły bardzo podobnie. Dla usuwania średnie czasy były takie same jednak całkowity czas był nieznacznie dłuższy niż w BST. Dla lotr.txt posortowanego RBT wypada już znacząco lepiej niż BST i nieznacznie lepiej niż Splay.

Dla zestawu kjb.txt (nieposortowanego) RBT okazało się nieznacznie wolniejsze od BST i nieznacznie szybsze od Splay tree. Jednak średnie czasy wszystkich operacji były takie same jak dla Splay. Dla kjb.txt posortowanego, które okazało się świetną sytuacją dla Splay, RBT wypada bardzo podobnie pod względem czasu, wykonuje jednak ono dużo więcej porównań i modyfikacji węzłów (18 373 667 porównań dla RBT i 791 747 porównań dla Splay). Podsumowując wyniki dla RBT – jako jedyne z badanych drzew nie miało ono przypadku, w którym wypadałoby niekorzystnie.

5. Przypadki idealne dla poszczególnych struktur

5.1 Binary search tree

Sytuacją idealną dla BST będzie zestaw danych o jak największym stopniu losowości, bez powtórzeń. Wtedy drzewo będzie możliwie dobrze zbalansowane i oczekiwane koszty operacji będą wynosiły O(log(n)). Przykładem takich danych może być np. książka.

5.2. Splay tree

Idealnym przypadkiem dla Splay tree będzie posortowany zestaw. Powtórzenia w nim nie będą szkodzić, ponieważ próbując wstawić, wyszukać lub usunąć drugi taki sam element już po jednym porównaniu algorytm kończy pracę (żadne inne z badanych drzew nie da takiego rezultatu). Zaproponowany przeze mnie zestaw to 2 703 830 słów w języku polskim.

scrabble-polish-words.txt						
	BST		splay		RBT	
insert	11981702932	2905,009	16334473	3,96	120460951	29,206
	1485895	0,36	79210688	19,205	22771227	5,521
	221263ms	0,054ms	5221ms	0,001ms	5405ms	0,001ms
search	11983188827	2905,37	33669373	8,163	84325326	20,445
	0	0	89136238	21,611	0	0
	130136ms	0,032ms	3623ms	0,001ms	4042ms	0,001ms
delete	265912877	64,472	65029290	15,767	1704260	0,413
	9229978	2,238	17476977	4,237	225657	0,055
	4170ms	0,001ms	2876ms	0,001ms	2454ms	0,001ms

W tej sytuacji Splay wypadł najlepiej zarówno pod względem czasu (nieznaczna różnica na korzyść czasowa RBT pojawiła się przy usuwaniu) jak i ilości porównań kluczy (poza usuwaniem). Gdyby w zestawie pojawiły się powtórzenia przewaga splay tree byłaby jeszcze bardziej widoczna. Im więcej by ich było, tym większa byłaby przewaga splay.

5.3. Red black tree

Przypadek idealny dla RBT niełatwo wskazać, ponieważ struktura ta zachowuje swoje własności dla różnych typów danych. Jednak można spróbować znaleźć taki zbiór danych, że zarówno BST jak i Splay nie będą najwydajniejsze. Przykładem może być połączenie zbioru posortowanego (to będzie problem na BST) ze zbiorem losowym bez powtórzeń (gorsze działanie splay) czyli np. zbiór scrabble-polish-words.txt połączony z fragmentem jakieś książki.

6. Wady i zalety rozważanych struktur

6.1. Binary search tree

Zaletą BST jest prostota implementacji, mała ilość wymaganej pamięci i szybkość działania przy losowych danych. Ponadto ilość modyfikacji węzłów jest bardzo mała. Warto ich użyć jeśli wiemy, że nasze dane będą miały duży stopień losowości. Wadą tej struktury jest to, że łatwo można je zdegenerować do listy i ogromna liczba porównań kluczy dla danych posortowanych z powtórzeniami. Inną wadą jest też to, że w przypadku ciągłego wyszukiwania tej samej wartości (co ma miejsce w przypadkach posortowanych) wykonujemy cały czas tą samą pracę.

6.2. Splay tree

Zaletą drzew splay jest prostota implementacji, mała ilość wymaganej pamięci oraz średni koszt operacji na poziomie drzew avl i RBT. Szczególną ich zaletą jest optymalizowanie dostępu do najczęściej używanych wartości, stąd drzewa splay są używane w implementacji pamięci cache. Jest to powód dla którego drzewa splay tak dobrze sobie radziły z posortowanymi zestawami danych zawierającymi dużo powtórzeń. Jedną z wad tej struktury jest możliwość bycia zdegenerowanym do listy (tak jak w BST). Inną wadą jest duża liczba modyfikacji węzłów. Jest to powód dla którego dla losowych danych bez powtórzeń drzewo nie wypadało najlepiej.

6.3. Red black tree

Największą zaletą RBT jest odporność na różne zestawy danych. Drzewo nie daje się zdegenerować i zawsze zapewnia logarytmiczny czas dostępu do elementów. W sytuacji kiedy nie wiemy jakie dane będziemy musieli przechowywać warto wybrać właśnie tę strukturę. Wadą RBT jest trudność w implementacji i stopień skomplikowania operacji wynikający z faktu, że po każdym wstawieniu/usunięciu trzeba zachować własności tego drzewa.