# Algorytmy i struktury danych - lista 4

Albert Piekielny

19/05/19

## 1 Wprowadzenie

Celem czwartej listy laboratoryjnej było zaimplementowanie i zasymulowanie działania trzech struktur drzewiastych : drzewa czerwono-czarnego, drzewa poszukiwań binarnych oraz samoorganizującego się drzewa splay. Każda z nich miała przechowywać ciągi znaków przyjmując porządek leksykograficzny. Ponadto wyżej wymienione struktury powinny udostępnieać funkcjonalności tj. :

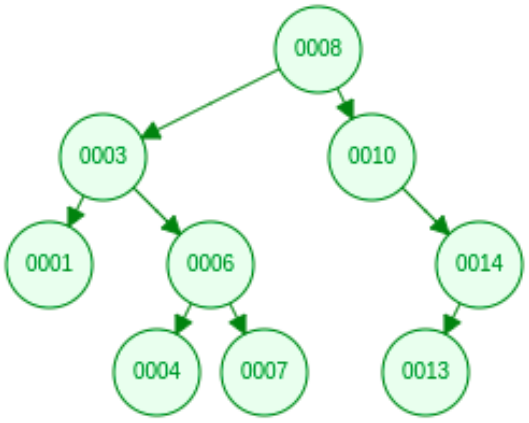
- insert - pozwalająca wstawić do struktury dowolny ciąg s
- delete - pozwalająca usunąć ze struktury ciąg s
- search - pozwalajaca wyszukać w strukturze ciąg s
- load - pozwalająca wczytać z pliku źródłowego wyrazy, a następnie wstawić je do struktury
- inorder - pozwalająca wypisać elementy struktury w posortowanej kolejności

Dodatkowo należało wykonać testy polegające na wstawieniu do struktur danych z plików testowych, kolejno zapytanie się o każde słowo w strukturze, oraz ostatecznie usunięcie każdego słowa. Na koniec należało na podstawie czasu działania programu oraz liczby wszystkich wykonanych operacji wywnioskować, które ze struktur są najbardziej efektywne.

# 2 Omówienie struktur

## 2.1 Drzewa poszukiwań binarnych

Drzewa poszukiwań binarnych (ang. Binary Search Tree) jest drzewem binarnym, w którym każdy węzeł spełnia poniższe reguły. Drzewo poszukiwań binarnych jest zakorzenionym drzewem binarnym, którego wewnętrzne węzły przechowują klucz (i opcjonalnie powiązaną wartość) i każde z nich ma dwa wyróżnione pod-drzewa powszechnie oznaczone jako lewe i prawe. Drzewo spełnia dodatkowo binarną właściwość, która stwierdza, że klucz w każdym węźle musi być większy lub równy jakiemukolwiek kluczowi przechowywanemu w lewym poddrzewie i mniejszym lub równym dowolnemu kluczowi przechowywanemu w prawym poddrzewie.Podczas wstawiania lub wyszukiwania elementu w drzewie BST klucz każdego odwiedzanego węzła należy porównać z kluczem elementu, który ma zostać wstawiony lub znaleziony.Kształt drzewa wyszukiwania binarnego zależy całkowicie od kolejności wstawiania i usuwania i może się zdegenerować.Po długiej wymieszanej sekwencji losowego wstawiania i usuwania oczekiwana wysokość drzewa zbliża się do pierwiastka kwadratowego liczby kluczy $\sqrt{n}$, która rośnie znacznie szybciej niż $\log n$




Rysunek 1: Przykład drzewa poszukiwań binarnych

## 2.2 Drzewa czerwono-czarne

Drzewo czerwono-czarne (ang. Red–black tree) jest rozszerzeniem podstawowej struktury o algorytm równoważenia wykonywany po każdej operacji *insert* oraz *delete*. W przypadku tej struktury elementy-liście nie przechowują żadnych informacji, dlatego często w ich miejsce wprowadza się dla zaoszczędzenia pamięci i uproszczenia kodu pojedynczego wartownika.W drzewie czerwono-czarnym z każdym węzłem powiązany jest dodatkowy atrybut - kolor, który może być czerwony lub czarny. Oprócz podstawowych własności drzew poszukiwań binarnych, wprowadzone zostały kolejne wymagania, które trzeba spełniać:

I. Każdy węzeł jest czerwony albo czarny.

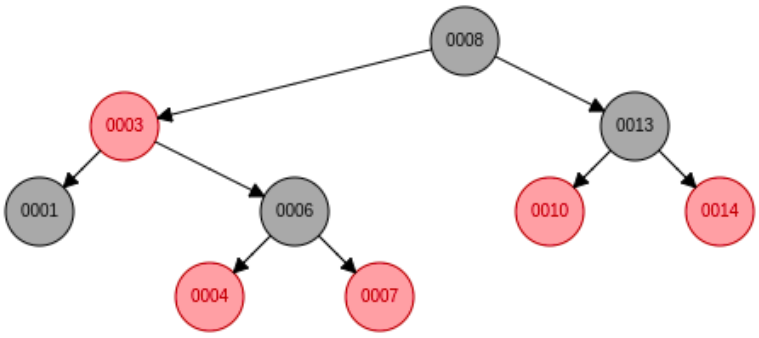
II. Korzeń jest czarny.

III. Każdy liść jest czarny (Można traktować null jako liść).

IV. Jeśli węzeł jest czerwony, to jego synowie muszą być czarni.

V. Każda ścieżka z ustalonego węzła do liścia liczy tyle samo czarnych węzłów.

Z powyższych warunków można udowodnić, że dla $n$ węzłów głębokość drzewa czerwono-czarnego $h$ wyniesie co najwyżej $2\log n + 1$, przez co elementarne operacje będą wykonywać się w czasie $O(\log n)$.

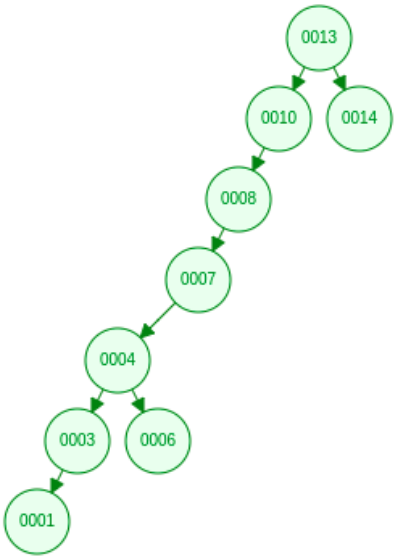



Rysunek 2: Przykład drzewa czerwono-czarnego

## 2.3 Drzewa splay

Splay tree struktura danych w formie samodostosowującego się drzewa poszukiwań binarnych (BST). 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 $O(\log n)$ – można np. tak wykonać operacje, że drzewo zdegeneruje się do listy.Procedura splay polega na wykonaniu sekwencji kroków, z których każdy przybliża element $x$ do korzenia. Każdy krok polega na wykonaniu jednej lub dwóch rotacji względem krawędzi wchodzących w skład początkowej ścieżki od $x$ do korzenia $s$.




Rysunek 3: Przykład drzewa splay

# 3 Wykonywne symulacje

W celu sprawdzenia efektywności działania każdej ze struktur wykonałem operacje dodawania, usuwania i wyszukiwania wyrazów zawartych w plikach: aspell_wordlist.txt, KJB.txt, lotr.txt, sample.txt.W pierwszej kolejności operacje były wykonywane na oryginalnej kolejności wczytywanych wyrazach z plików natomiast w następnym kroku każda operacja bazowała na losowych permutacjach wyrazów w pliku.Każda z czynności została powtórzona 100 razy, a następnie wynik został uśredniony. Wyniki z symulacji zamieszczone są poniżej gdzie dane to czas w $ns$.

| Insert[ns] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Nazwa pliku | Sample.txt | | KJB.txt | | Lotr.txt | | Aspell_ wordlist.txt | |
| Permutacje | Tak | Nie | Tak | Nie | Tak | Nie | Tak | Nie |
| SPLAY | 336.867 | 636.859 | 673.990.268 | 492.610.384 | 151.615.523 | 128.522.930 | 149.426.930 | 98.48.593 |
| RBT | 1.336.803 | 562.522 | 912.532.845 | 674.838.889 | 145.041.240 | 126.535.806 | 95.516.864 | 39.792.060 |
| BST | 1.267.715 | 405.425 | 368.654.916.350 | 321.874.500.897 | 10.221.812.342 | 8.757.569.368 | 96.968.187 | 44.579.529.366 |

Rysunek 4: Tabela prezentująca czas operacji insert

| Search [ns] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Nazwa pliku | Sample.txt | | KJB.txt | | Lotr.txt | | Aspell_ wordlist.txt | |
| Permutacje | Tak | Nie | Tak | Nie | Tak | Nie | Tak | Nie |
| SPLAY | 253.459 | 1.724.184 | 385.169.124 | 291.202.701 | 98.916.007 | 89.054.454 | 203.792.209 | 24.525.986 |
| RBT | 126.853 | 397.592 | 224.499.538 | 187.106.692 | 63.322.200 | 53.898.670 | 101.981.012 | 30.367.848 |
| BST | 51.452 | 81.318 | 369.254.025 | 635.351.321 | 79.475.561 | 81.515.711 | 113.970.943 | 59.386.245.712 |

Rysunek 5: Tabela prezentująca czas operacji search

| Delete[ns] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Nazwa pliku | Sample.txt | | KJB.txt | | Lotr.txt | | Aspell_ wordlist.txt | |
| Permutacje | Tak | Nie | Tak | Nie | Tak | Nie | Tak | Nie |
| SPLAY | 48.003 | 39.754 | 149.528.180 | 130.697.142 | 27.543.202 | 26.521.234 | 11.122.479 | 3.513.933 |
| RBT | 1.091.550 | 801.404 | 984.635.466 | 801.446.640 | 193.921.552 | 168.988.878 | 153.629.673 | 48.187.720 |
| BST | 86.397 | 77.282 | 446.932.570 | 504.502.484 | 90.967.247 | 79.016.997 | 99.633.571 | 4.191.072 |

Rysunek 6: Tabela prezentująca czas operacji delete

# 4 Anliza symulacji

Rozpoczynając analizę danych zawartych w tabeli warto najpierw skategoryzować pliki, które były użyte do testów.Pliki *sample.txt*, *lotr.txt*,$KJB.txt$ są plikami zawieracjącymi powtarzające się wyrazy, natomiast $aspell\_wordlist.txt$ zawiera unikatowe wystąpienia wyrazów posortowanych alfabetycznie.Kolejnym kryterium na które należy zwrócić uwagę jest rozmiar plików. W tym wypadku najmniejszy z nich to *sample.txt* liczący sobie 260 słów następnie $aspell\_wordlist.txt$ 121 tysięcy słów, *lotr.txt* 141 tysięcy słów oraz ostatecznie $KJB.txt$ zawierający 821 tsięcy słów.

Biorąc pod uwagę operację insert zauważamy, że permutowanie wyrazów zawartch w pliku ma istotny wpływ na szybkość operacji. Najbardziej jest to dostrzegalne w przypadku pliku $aspell\_wordlist.txt$. Drzewo splay w przypadku posortowanej listy danych dołącza kolejne elementy w sposób znacznie szybszy niż w przypadku dodawania elementów do struktury w losowej kolejności, ponieważ kolejność nieleksykograficzna wymusza na drzewie częste przebudowy z użycuem operacji $Splay(T, x)$. Podobie jest w przypadku RBT chociaż różnica w czasie pomiędzy uporządkowanym ciągiem a tym permutowanym jest około dwuktornie gorsza. Sytuacja wygląda inaczej jeśli spojrzymy na drzewo poszukiwań binarnych. Czas dodawania wyrazów w kolejności losowej jest znacznie szybszy, dzieje się tak ponieważ w przypadku uporządkowanych wyrazów BST degenertuje się do listy tworząc najgorszy przypadek użycia, a drzewo staje sie skrajnie niezbalansowane. Wówczas złożoność operacji insert wzrasta do $O(n)$.Dla dużych plików, które zawierają liczne powtórzenia zestawienie rezultatów w tabeli pokazuje, że jeśli są to

drzewa samoorganizujące to wyniki są o mniej więcej połowę gorszę jeśli zachodziła permutacja.

Operacja delete w przypadku drzew samoorganizujących wymaga dodatkowego nagładu pracy w przypadku powstania drzewa niezbalansowanego.Jeśli struktura jest zbalansowana drzewo binarne radzi sobie lepiej w porównaniu do drzew samoorganizujących, wnioskując to na przykładzie listy uporządkowanych wyrazów.

Rozważając operacje *search* wymusza ona na drzewie wykonywanie operacji splay co kosztuje nas dodatkowy czas.W tym przypadku drzewa BST oraz RBT w momencie zbalansowania okazują sie lepszą formą, jeśli zbiór danych jest dostatecznie duży.

# 5 Wnioski

Rozmiar danych ma istotny wpływ na kształt i zachowanie się struktury przechowującą informacje. Dodatkowo należy zwrócic uwagę na to, że kolejność wprowadzania danych wpływa niejednokrotnie w sposób drastyczny na zmianę struktury w przypadku drzew samoorganizujących się.Ze wszystkich struktur w tym zestawieniu BST wypada najmniej korzystnie bierze sie to z faktu iż nie posiada ono w sobie algorytów równoważących strukturę.Warto pochylić się nad wadami i zaletami poszczególnych struktur.

Najważniejszą wadą drzew splay jest to, że wysokość drzewa splay może być liniowa. Na przykład będzie to miało miejsce po uzyskaniu dostępu do wszystkich $n$ elementów w kolejności malejącej. Ponieważ wysokość drzewa odpowiada najgorszemu czasowi dostępu, oznacza to, że rzeczywisty koszt operacji może być wysoki.Pod tym względem również drzewo poszukiwań binarnych również może zostać zdegenerowane do postaci listy.W drzewie czerwono-czarnym taki przypadek nie zachodzi, a dodatkowo z jego właściwości wiemy że droga od najblizszego do najdalszego elementu od korzenia może być conajwyżej dwukrotnie większa.

Główną zaletą drzew wyszukiwania binarnego w porównaniu z innymi strukturami danych jest to, że powiązane algorytmy sortowania i algorytmy wyszukiwania, takie jak „in-order traversal", mogą być bardzo wydajne, są również łatwe do kodowania. Wadą struktury RBT jest fakt ciągłego sprawdzania czy dana struktura jest zbalansowana i nanoszenie ewentualnych korekt ponadto implementacja tej struktury nie jest już tak łatwa jak w przypadku BST oraz wymagają dodatkowego pola przechowującego kolor

węzła co więże sie z koszte pamięci.

Zaletą drzew splay jest optymalizowanie dostępu przez przesuwanie wartości, do których ostatnio uzyskiwano dostęp, blisko korzenia, co powoduje ich dużą przydatność w implementacji pamięci cache i algorytmów odśmiecania.