Złożone struktury danych
Sprawozdanie nr 2

Adam Piaseczny
151757

Igor Szczepaniak
151918

Grupa piątkowa 11:45

Spis treści

1 Wprowadzenie

W tym sprawozdaniu zbadaliśmy sposoby wyszukiwania danych w różnych strukturach, byliśmy w stanie dzięki temu zweryfikować wiadomości poznane w trakcie zajęć. Kod źródłowy został napisany w języku `python` ze względu na prostotę implementacji.

Drogą prób i błędów doszliśmy do wniosku, że najbardziej optymalne dla naszych testów będzie ograniczenie maksymalnej ilości elementów do 20000 ze względu na problemy wiążące się z czasem wyszukiwania liniowego w tablicach o większej ilości elementów. Przewidzieliśmy 20 punktów pomiarowych, zaczynając od 1000 elementów z krokiem co 1000.

2 Tworzenie struktur

Dla wspomnianych wcześniej 20 wartości n wygenerowaliśmy tablicę A zawierającą niepowtarzające się liczby całkowite dodatnie. Następnie wykonaliśmy kopię tej tablicy za pomocą `numpy.ndarray.copy()`, którą potem posortowaliśmy korzystając z metody sortowania szybkiego (z elementem `pivot` na środku). Czas dodatkowego przypisania pamięci dla nowej tablicy B oraz jej sortowania oznaczyliśmy jako C_B.

Korzytając z tablicy A stworzyliśmy binarne drzewo poszukiwań, które w dalszej części pracy oznaczyliśmy jako TA. Zaimplementowaliśmy klasę `Tree`, która posiada zmienną instancyjną `root` służącą do zapisywania korzenia drzewa binarnego będącego instancją `Node`. Każdy obiekt `Node` zawiera od zera do dwóch potomków (`self.left`, `self.right`), przy czym każdy potomek "widzi" tylko swoich potomków (nie jest w stanie odnieść się do swojego rodzica). Każdy wierzchołek drzewa posiada również liczbę całkowitą (`self.data`), według której jest porównywany z innymi wierzchołkami w strukturze. Wszystkie operacje na drzewie (wyszukiwanie, wprowadzanie danych oraz badananie wysokości struktury) rozpoczynają się na elemencie `root`. Wprowadzanie wartości do drzewa, jak i jej wyszukiwanie, wartości opiera się na następującym algorytmie, gdzie rozpatrywany wierzchołek to `self`:

1. Jeśli wprowadzana wartość jest mniejsza od `self.data`
   (a) Jeśli `node.left` nie istnieje, tworzymy nowy wierzchołek z wprowadzaną wartością jako `self.data`, przypisujemy go jako `node.left` i kończymy wykonywanie funkcji.
   (b) W przeciwnym wypadku wykonujemy ten sam algorytm dla już istniejącego `node.left`.
2. Jeśli wprowadzana wartość jest większa albo równa `self.data`.
   (a) Jeśli `node.right` nie istnieje, tworzymy nowy wierzchołek z wprowadzaną wartością jako `self.data`, przypisujemy go jako `node.right` i kończymy wykonywanie funkcji.
   (b) W przeciwnym wypadku wykonujemy ten sam algorytm dla już istniejącego `node.right`.

Wyszukiwanie elementu w drzewie działa w sposób zbliżony do powyższego algorytmu, z tą różnicą, że jeżeli na drodze działania algorytmu natrafimy na `Node`, który nie istnieje, kończymy algorytm z informacją, że szukanego elementu nie ma w strukturze. Jeśli natomiast natrafimy na `self.data` równa szukanej wartości kończymy algorytm zwracając `self`.

Używając biblioteki `sys` dowiedzieliśmy się, że każdy element drzewa zajmuje 48 bajtów pamięci, podczas gdy każdy element tablicy zajmuje 8 bajtów pamięci. Łączna ilość pamięci zajęta przez drzewo to $(48 \times n + 1)$. Dodatkowe 48 bajtów pamięci jest zajmowane przez `Tree`, które jest odnośnikiem do korzenia drzewa i służy do przeprowadzania operacji na drzewie poszukiwań. Dzięki temu odnośnikowi klasa `Tree` pozwala oszczędzić 16 bajtów na każdym wierzchołku w drzewie, które byłyby poświęcone odnośnikowi do `Tree.root`. Pomimo wyżej wspomnianego zaoszczędzenia pamięci struktura drzewa binarnego dla danego n elementów będzie zawsze bardziej kosztowna pod względem pamięci niż tablica, gdyż łączny rozmiar tablicy to $8 \times n$.

Wprowadzając dane nieuporządkowane z tablicy A, idąc od pierwszego elementu do ostatniego w liście, zauważyliśmy, że wprowadzanie nowego elementu do drzewa TA z czasem wymaga coraz więcej porównań oraz wywołań rekurencyjnych wyżej opisanego algorytmu (aż do h, gdzie h jest wysokością drzewa, czyli najniższy poziom istniejący w drzewie). Przy tablicy A nie możemy w kontrolowany sposób wpłynąć na maksymalną wysokość drzewa (ze względu na nieposortowany rozkład danych), przez co czas wprowadzania danych - $O(h)$, jest znacznie zwiększony.

Gdyby wprowadzać dane z tablicy B w taki sam sposób jak z tablicy A (od indeksu 0 do $n-1$) otrzymaliśmy drzewo o wysokości n, gdzie każdy wierzchołek miałby dokładnie jednego potomka `node.right`. Jest to najgorszy możliwy przypadek wprowadzania danych do BST, ponieważ otrzymana struktura zajmuje więcej pamięci niż tablica. W takim drzewie można wyszukiwać dane tylko liniowo (sprawdzanie elementów jeden po drugim) oraz każda funkcja rekurencyjna (dodawanie elementów, wyszukiwanie albo sprawdzanie głębokości) wykonywałaby się aż n razy. Mając to na względzie wprowadzając dane z tablicy B użyliśmy dzielenia połówkowego tablicy posortowanej. Algorytm dzielenia połówkowego postępuje w następujący sposób:

1. Wyznacza środkowy element tablicy
2. Dodaje wybrany element do tablicy wynikowej
3. Wykonuje dzielenie połówkowe dla elementów z lewej strony środkowego
4. Wykonuje dzielenie połówkowe dla elementów z prawej strony środkowego

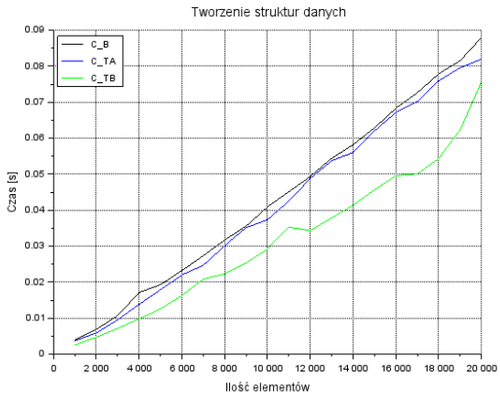
Algorytm dzielenia połówkowego, tworząc tablice pomocniczą, gdzie każdy kolejny element stanowi medianę dzielonego fragmentu tablicy, sprawiło, że przypadek średni podczas budowania drzewa jest bliższy optymistycznemu niż pesymistycznemu znacząco zmniejszając wysokość drzewa. Wprowadzając kolejne elementy wyżej opisanej tablicy pomocniczej stworzyliśmy drzewo TB. Struktura ta przy wprowadzaniu kolejnych elementów podczas jego tworzenia wymagała mniejszej ilości porównań (niż w przypadku drzewa TA) zmniejszając tym samym wyskość drzewa h. Czas poświęcony na tworzenie drzewa TB (pomijając czas tworzenia tablicy pomocniczej) oznaczony jest jako C_{TB}.

W następującej tabeli przedstawiliśmy zależność czasu t od ilości elementów n dla C_B, C_{TA}, C_{TB}.

t[s]\n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
C_B(t)	0.0039535046	0.0069227219	0.0106663227	0.0171151638	0.0192873478	0.0231803417	0.0273646832	0.0316935062	0.0355618000	0.0408860207
C_TA(t)	0.0036582947	0.0058877468	0.0095236778	0.0137891769	0.0178986073	0.0219568253	0.0246876240	0.0301155567	0.0351301670	0.0373646736
C_TB(t)	0.0024954796	0.0047024727	0.0071885109	0.0098190308	0.0125310898	0.0162966728	0.0208459377	0.0222617149	0.0253544807	0.0292221069
t[s]\n	11000	12000	13000	14000	15000	16000	17000	18000	19000	20000
C_B(t)	0.0451359272	0.0493522167	0.0544027328	0.0581908226	0.0628466606	0.0683228970	0.0726035118	0.0778793812	0.0815277100	0.0879700184
C_TA(t)	0.0425172329	0.0488282681	0.0536998272	0.0560466766	0.0619755745	0.0671728611	0.0701804638	0.0759353161	0.0795733452	0.0819494247
C_TB(t)	0.0352524281	0.0343510151	0.0378840446	0.0413631439	0.0456659794	0.0495895863	0.0500306606	0.0543353558	0.0623500347	0.0756380081

Rysunek 1: $t = f(n)$ - Tworzenie struktur (tabela)

Z wyników z stworzyliśmy następujący wykres:



Rysunek 2: $t = f(n)$ - Tworzenie struktur

Drzewo TB podczas wprowadzania elementów musi wykonać w średnim przypadku mniej porównań między danymi w drzewie, niż drzewo TA. Z tego powodu czas tworzenia TB jest niższy dla dowolnej ilości elementów n w porównaniu do procesu tworzenia drzewa TA. W praktyce czas tworzenia drzewa TB dla danych nieposortowanych wymaga stworzenia tablicy B oraz uporządkowania jej, co wymaga dodatkowego czasu zawartego w C_B. Wymóg niezbędnego sortowania tablicy B powoduje, że łączny czas tworzenia drzewa TB jest większy od TA dla danych nieposortowanych.

3 Wyszukiwanie elementów

W tym teście zmierzyliśmy jak długi czas zajmuje wyszukiwanie elementów w poszczególnych strukturach danych.

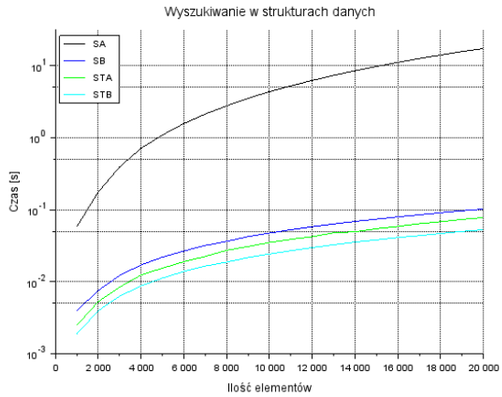
W tablicy A zastosowalismy wyszukiwanie liniowe, czyli po pierwszym napotkanym szukanym elemencie zwracamy jego indeks i kończymy wykonywanie funkcji. Wyszukiwanie binarne dzieli tablicę posortowaną na dwie równe części według mediany i sprawdza czy szukana wartość jest większa czy mniejsza od mediany: jeśli jest większa to wywołujemy funkcje rekurencyjnie dla prawej strony tablicy, w przeciwnym przypadku wykonujemy dla lewej strony tablicy, przez rozpoczęciem wywołań rekurencyjnych sprawdzamy czy mediana jest równa szukanej wartosci, jeśli tak - zwracamy jej indeksi kończymy wykonywanie algorytmu. Szukanie każdego elementu z A w B za pomocą wyszukiwania binarnego oznaczone jest jako S_B podczas gdy wyszukiwanie liniowe każdego elementu z B w A jest oznaczone jako S_A.

Dodatkowo wyszukujemy wszystkie elementy z A w drzewie TA (S_{TA}), oraz wszystkie elementy z B w drzewie TB (S_{TB}).

t[s]\n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
SA	0.0580030441	0.1769890785	0.3927890301	0.7154350758	1.0912916660	1.5742928028	2.1402818680	2.7786334038	3.5344562531	4.3634362221
SB	0.0039489269	0.0075757027	0.0121905804	0.0171226978	0.0220200062	0.0267930984	0.0319115162	0.0364289284	0.0422406197	0.0473275661
STA	0.0024803162	0.0052908421	0.0084271908	0.0124205112	0.0154263496	0.0190159798	0.0225129604	0.0273669243	0.0309039116	0.0350992680
STB	0.0018844604	0.0039511681	0.0063227654	0.0087776184	0.0112461567	0.0139142513	0.0165072441	0.0187515736	0.0217376232	0.0243051529
t[s]\n	11000	12000	13000	14000	15000	16000	17000	18000	19000	20000
SA	5.2619740009	6.2656570435	7.3776454926	8.5569758892	9.7924404144	11.1315562248	12.5889571667	14.0802796841	15.7431794167	17.4047279835
SB	0.0528895378	0.0583114624	0.0634987831	0.0690026760	0.0743086338	0.0797502041	0.0852955818	0.0909752846	0.0968749046	0.1021724224
STA	0.0388183117	0.0421743393	0.0477985859	0.0494324684	0.0545623302	0.0588071823	0.0640639782	0.0683040619	0.0737302303	0.0779719830
STB	0.0269826412	0.0298608303	0.0326733112	0.0355318069	0.0381535530	0.0411116123	0.0435657501	0.0468106747	0.0499357700	0.0527621746

Rysunek 3: $t = f(n)$ - Wyszukiwanie w strukturach (tabela)

Z powyższych danych sporządziliśmy następujący wykres:



Rysunek 4: $t = f(n)$ - Wyszukiwanie w strukturach

Wyszukiwanie liniowe w najgorszym przypadku ma złoność czasową $O(n)$. Takowy przypadek ma miejsce, kiedy wyszukiwany element znajduje się na końcu tablicy (aby go znaleźć należy sprawdzić aż n elementów). Najgorszy przypadek dla wyszukwiania binarnego ma złożoność czasową $O(\log n)$. Dzieje się tak podczas najgłębszego zagnieżdżenia rekurencji, na przykład w przypadku szukania pierwszego albo ostatniego elementu tablicy.

Na wykresie możemy również zauważyć, że nawet przy losowym wprowadzaniu danych do drzewa TA wyszukiwanie w nim jest szybsze niz wyszukiwanie binarne w liście B. Dzieje się tak z powodu faktu, że przy wykonywaniu rekurencyjnym wyszukiwania binarnego tworzą się kolejne funkcje na stosie rekurencyjnym co w przypadku języka `python` jest bardzo kosztowne dla pamięci. Dodatkowym "balastem pamięciowym" przypisanym do kolejnych wywołań jest również mediana, która oprócz zajmowania czasu na samo jej "obliczenie", rozszerza w kolejnym wywołaniu rekurencyjnemu słownik `locals()`, zawierający zmienne w zasięgu lokalnym.

Jeśli wyszukujemy w binarnym drzewie poszukiwań TA, każde wywołanie rekurencyjne wyszukiwania `Node.search()` nie rozszerza słownika `locals()`, z racji braku zmiennych lokalnych - wyszukiwanie w drzewie wykonuje tylko porównania. Przez niższą średnią wysokość drzewa TB spowodowaną dzieleniem połówkowym B wyszukiwanie w drzewie TB jest znacznie szybsze od wyszukiwania w drzewie TA z racji mniejszej ilości porównań. Z tego powodu ten rodzaj wyszukiwania jest najszybszy ze wszystkich przez nas analizowanych.

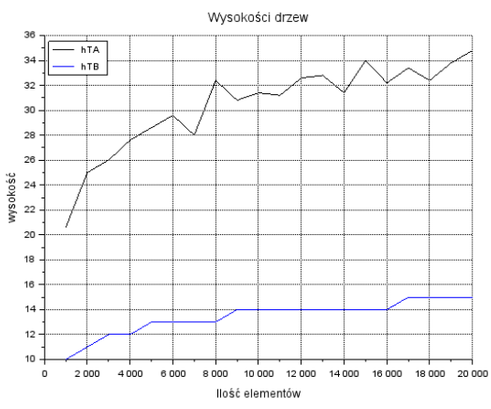
4 Wysokość

W tym teście sprawdziliśmy w jaki sposób ilość elementów wpłynęła na wysokość drzew TA oraz TB. Uśrednione wyniki naszych testów przedstawiliśmy w poniższej tabeli:

h\n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
hTA	20.6	25	26	27	28.6	29.6	28	32.4	30.8	31.4
hTB	10	11	12	12	13	13	13	13	14	14
h\n	**11000**	**12000**	**13000**	**14000**	**15000**	**16000**	**17000**	**18000**	**19000**	**20000**
hTA	31.2	32.6	32.8	31.4	34	32.2	33.4	32.4	33.8	34.8
hTB	14	14	14	14	14	14	15	15	15	15

Rysunek 5: Wysokość struktur (tabela)

Z powyższych danych sporządziliśmy następujący wykres:



Rysunek 6: Wysokość struktur

Jak widać na powyższym wykresie wraz ze wzrostem ilości elementów wysokość drzewa TA rośnie znacznie szybciej niż wysokość drzewa TB. Dzieje się tak, ponieważ rozkład danych w drzewie TB jest bliżej przypadku optymistycznego przez użycie dzielenia połówkowego w jego tworzeniu. Wysokość drzew ma znaczący wpływ na szybkość operacji w tej strukturze danych co można dostrzec zestawiając powyższy wykres z danymi z wcześniejszych testów - duża wysokość drzewa powoduje dłuższy czas operacji na drzewie ze względu na ilość porównań.

Przy tworzeniu drzewa TA nie były wymagane dodatkowe operacje przed rozpoczęciem iteracji przez tablicę A aby dodać kolejne elementy do drzewa TA. Za to w przypadku drzewa TB musiała zostać stworzona tablica pomocnicza jeszcze przed procesem dodawania elementów do drzewa. Jeśli chcielibyśmy dodać nowy element do stworzonego już drzewa TB i utrzymać jego możliwie minimalną wysokość, musielibyśmy przejść przez proces dzielenia połówkowego posortowanej tablicy zawierającej nowy element w celu wyważenia drzewa, czyli optymializacji rozkładu danych w bst.

5 Wnioski

Używanie tablic jest prostsze w implementacji niż drzewo wyszukiwań, tablica zajmuje znacznie mniej pamięci niż drzewo binarne i pozwala na bezpośredni dostęp do danych (używając indeksu). Wyszukiwanie liniowe w nieposortowanej liście jest za to znacznie wolniejsze niż inne metody wyszukiwania rozważane w tej pracy, aczkolwiek jeśli dane są posortowane, rekurencyjne wyszukiwanie binarne jest porównywalnie szybkie do wyszukiwania w drzewach bst.

Drzewa bst są trudniejsze w implementacji oraz wymagają znacznie więcej pamięci na przechowywanie zmiennych instancyjnych. Dodatkowy czas poświęcony na tworzenie drzewa rekompensowany jest przez szybsze czasy wyszukiwania (nawet w drzewie TA mimo dużej wysokości tej struktury). Najszybszą rozpatrywaną metodą jest wyszukwiwanie w drzewie doskonale zrównoważonym, którego wadą jest potrzeba ponownego wyważania drzewa jeśli będzie niezbędne dodanie elementów.

6 Zakończnenie

Wszystkie struktury mają swoje wady oraz zalety i warto rozważyć wiele czynników podczas decydowania, która struktura jest lepsza do danego zastosowania. Kod do wykorzystanego programu oraz pliki wyjściowe z danymi są dostępne pod witryną:

https://github.com/TypicalAM/Algorithms-Lab