Wtorki 16:50
Grupa I3
Kierunek Informatyka
Wydział Informatyki
Politechnika Poznańska

Algorytmy i struktury danych

Sprawozdanie z zadania w zespołach nr. 2
prowadząca: dr hab. inż. Małgorzata Sterna, prof PP

Wybrane złożone struktury danych

autorzy:
Piotr Więtczak nr indeksu 132339
Tomasz Chudziak nr indeksu 136691

16 kwietnia 2018

1 Opis implementacji i czasy tworzenia wybranych struktur danych

Do implementacji wybranych struktur danych użyliśmy języka C++, a do pomiarów czasu klasy *std* :: *chrono* :: *high_resolution_clock* z biblioteki $<$ *chrono* $>$.Wszystkie pomiary rozmiarów wykonano funkcją sizeof().

Tablica

Tablica rozmiar pojedynczego elementu: 4 B.

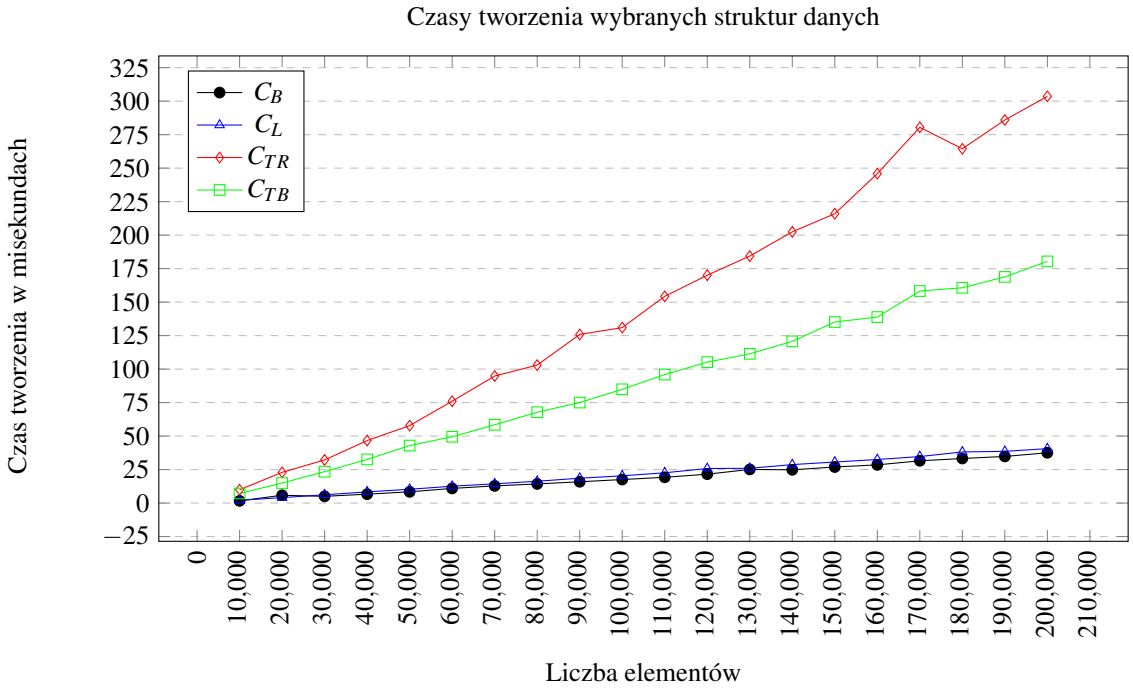
Lista jednokierunkowa

Lista jednokierunkowa opiera się na dwóch rodzajach struktur. Pierwsza z nich to głowa, w niej znajdują się wskaźniki pokazujące na pierwszy i ostatni element listy, sama w sobie nie przechowuje żadnego elementu. Drugi rodzaj to element składa się on przechowywanej wartości oraz ze wskaźnika lokalizującego następny element. Każda lista posiada jedną głowę i pewną ilość elementów odpowiadającą ilości liczb do zapamiętania. Rozmiar głowy w liście jednokierunkowej: 16 B. Rozmiar elementu w liście jednokierunkowej: 16 B Na jeden element składa się pole wartości przechowujące liczbę naturalną, oraz wskaźnik na następny element w liście.

Drzewo poszukiwań binarnych

BST zbudowana jest w jednej klasie. Podczas tworzenia tej struktury początkowo tworzy się tylko głowa to ona odpowiedzialna jest za wskazanie gdzie rozpoczyna się korzeń. By odróżnić ją od elementów przechowujących liczby do zapamiętania daliśmy jej wartość -1 (założyliśmy że ta konstrukcja będzie przechowywać tylko i wyłącznie liczby większe, równe 0). W klasie tej zaimplementowaliśmy konstruktor oraz trzy metody. Pierwsza przyłącza element do struktury. Podczas dodawania pierwszego elementu głowa zaczyna wskazywać na niego. Dodawanie kolejnych odbywa się tak samo jak dla zwykłego BST z tą różnicą, że wpierw algorytm musi przejść przez głowę nie uwzględniając jej w dalszym szukaniu. Druga sprawdza wysokość drzewa, robi to w sposób rekurencyjny nie uwzględniając głowy. Trzecia odpowiedzialna jest za sprawdzanie czy element znajduje się w strukturze, polega ona na przeszukiwaniu tej struktury. Rozmiar elementu w drzewie przeszukiwań binarnych: 32 B. Na jeden element składa się pole wartości przechowujące liczbę naturalną, wskaźnik na lewego syna elementu i wskaźnik na prawego syna elementu.

Wykresy przedstawiające średni czas tworzenia struktury, w milisekundach, od rozmiaru zestawu danych



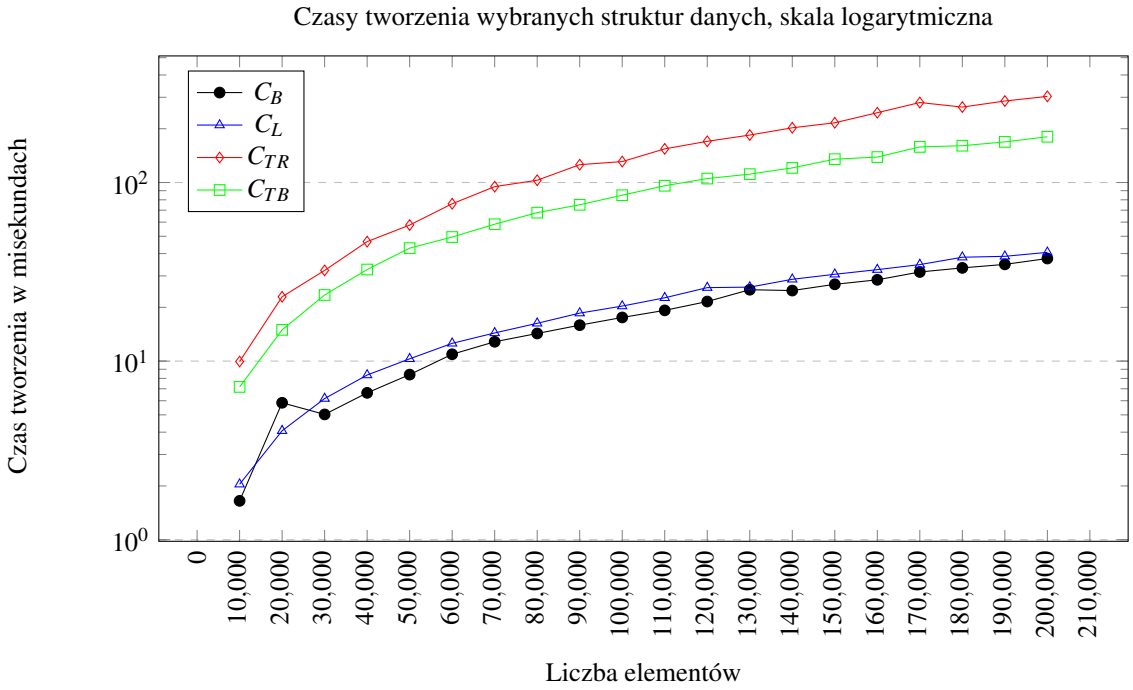
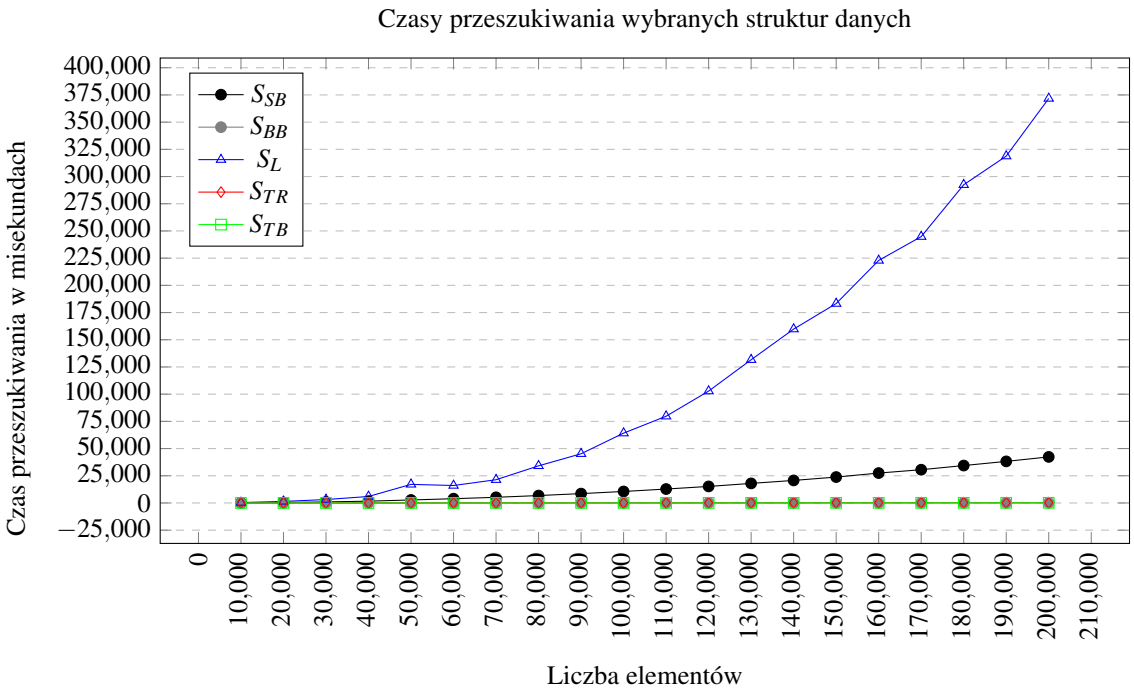


Tabela przedstawiająca średni czas tworzenia struktury w milisekundach

Liczba ele.	C_B $[ms]$	C_L $[ms]$	C_{TR} $[ms]$	C_{TB} $[ms]$
10000	1.650	2.044	9.945	7.169
20000	5.843	4.073	22.923	14.959
30000	5.028	6.169	32.196	23.456
40000	6.641	8.355	46.658	32.567
50000	8.402	10.297	57.789	42.923
60000	10.921	12.575	76.041	49.510
70000	12.831	14.377	94.813	58.417
80000	14.280	16.281	102.981	67.797
90000	15.891	18.565	125.854	75.034
100000	17.556	20.329	130.962	84.913
110000	19.236	22.617	154.307	95.912
120000	21.554	25.787	170.075	105.213
130000	25.102	25.953	184.363	111.399
140000	24.829	28.711	202.444	120.719
150000	26.912	30.632	215.974	135.144
160000	28.519	32.516	245.861	138.828
170000	31.551	34.703	280.478	158.247
180000	33.262	38.168	264.447	160.696
190000	34.803	38.614	285.932	168.794
200000	37.533	40.629	303.575	180.382

2 Przeszukiwanie wybranych struktur danych

Wykresy przedstawiające średnie czasy wyszukiwania wszystkich elementów struktury, w milisekundach, od rozmiaru zestawu danych



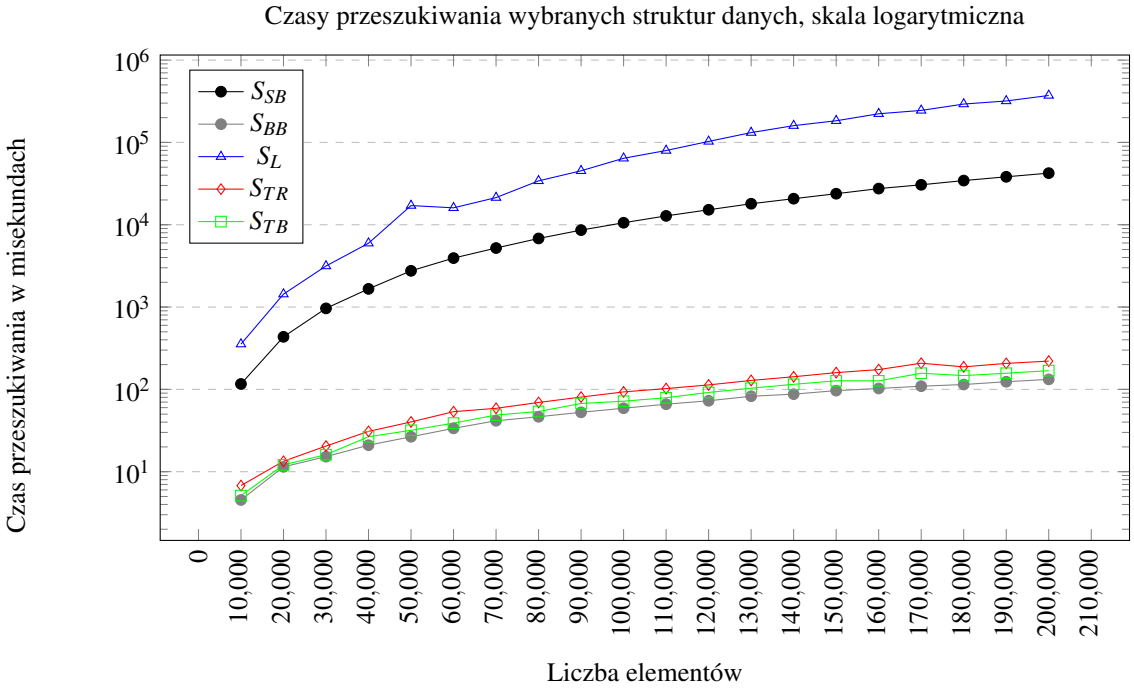


Tabela przedstawiająca czas wyszukiwania wszystkich elementów struktury w milisekundach

Liczba ele.	S_{SB} $[ms]$	S_{BB} $[ms]$	S_L $[ms]$	S_{TR} $[ms]$	S_{TB} $[ms]$
10000	116.347	4.534	354.608	6.778	5.125
20000	434.923	11.445	1435.100	13.393	12.102
30000	963.589	15.227	3150.970	20.482	16.104
40000	1663.180	20.992	5964.640	30.896	26.580
50000	2755.070	26.554	17094.000	40.196	31.822
60000	3933.720	33.620	16052.900	53.579	38.924
70000	5211.100	41.562	21362.700	58.858	48.745
80000	6811.880	46.460	34108.400	69.323	53.877
90000	8601.650	52.667	45225.500	80.632	67.380
100000	10571.600	58.985	64175.800	93.133	71.823
110000	12800.800	66.020	79715.500	102.264	79.161
120000	15197.600	72.712	102809.000	113.286	91.825
130000	17988.800	82.524	131624.000	128.353	103.391
140000	20721.400	87.497	159801.000	142.335	115.199
150000	23809.200	96.570	183256.000	159.782	127.070
160000	27484.000	102.584	222896.000	173.954	126.894
170000	30563.800	109.273	244782.000	207.902	157.062
180000	34389.200	114.565	292448.000	187.756	147.043
190000	38219.600	123.794	318782.000	206.874	157.062
200000	42337.700	132.252	371754.000	220.576	168.860

Wyszukiwanie elementów w tablicy metodą wyczerpującą dla wszystkich elementów nie różni się złożonością, która wynosi $O(n^2)$, od wyszukiwania w liście jednokierunkowej. Działają one na podobnej zasadzie, dla każdego wyszukania, sprawdzane są wszystkie elementy od początkowego, aż do szukanego. Mimo to przeszukiwanie listy okazało się wolniejsze, a wraz ze wzrostem liczby elementów różnica pogłębia się. Powodem tego może być to że lista odwołuje się do elementów po adresie w pamięci, a nie jak w przypadku tablicy po numerze indeksu.

Czasy otrzymane dla przeszukiwania binarnego tablicy B i drzewa TB, osiągały podobne wartości, jednak we wszystkich przypadkach wyszukiwanie w tablicy poradziło sobie lepiej, zapewne z tego samego powodu co w przypadku listy poruszanie się po strukturze za pomocą adresów poszczególnych elementów jest mniej wydajne od używania ich indeksów. W obu przypadkach przebieg wyszukiwania wszystkich elementów wygląda podobnie, za każdym razem sprawdzane jest czy poszukiwany element jest równy aktualnie wybranemu, w przeciwny razie, wykorzystując uporządkowanie struktur, wyszukiwanie jest kontynuowane tyko dla połowy danych zawierającej elementy mniejsze/większe od niego. Co powoduje że po każdym porównaniu odrzucana jest połowa elementów. Złożoność takiego wyszukiwania dla wszystkich elementów wynosi $O(n\log_n)$.

3 Wysokość drzewa poszukiwań binarnych

Wykres przedstawiający wysokości drzew od rozmiaru zestawu danych

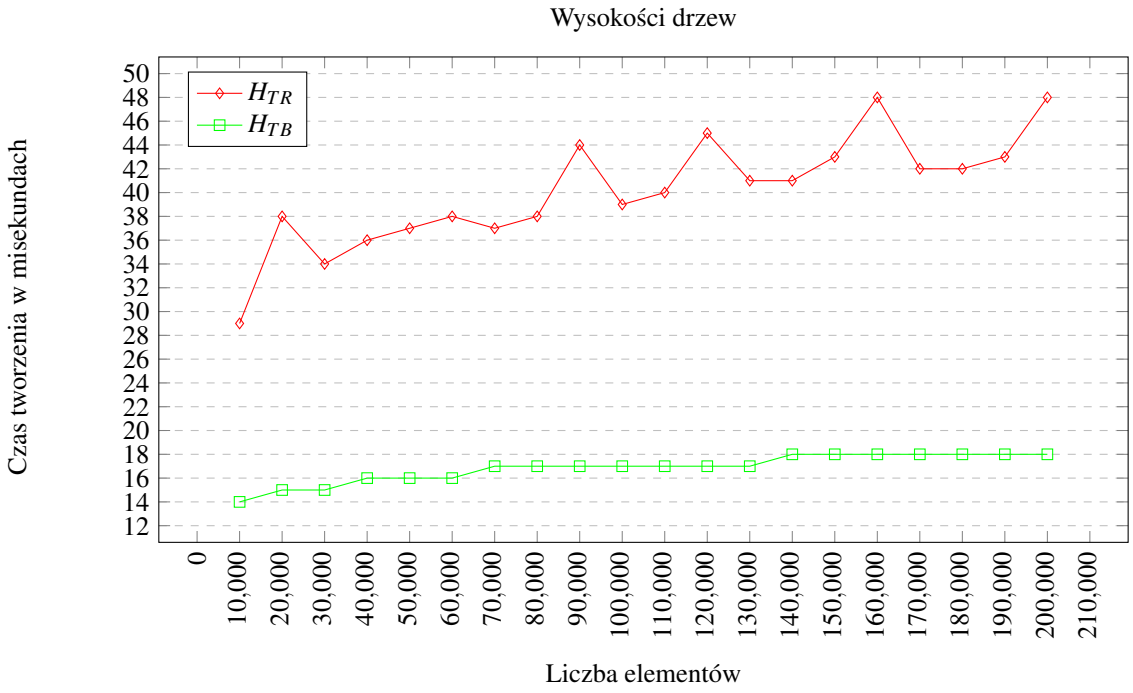


Tabela przedstawiająca średnie wysokości drzew od liczby elementów

Liczba ele.	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000	150000	160000	170000	180000	190000	200000
H_{TR}	29	38	34	36	37	38	37	38	44	39	40	45	41	41	43	48	42	42	43	48
H_{TB}	14	15	15	16	16	16	17	17	17	17	17	17	17	18	18	18	18	18	18	18

Wysokość badanych drzew TR okazuje się być ponad dwukrotnie większa od drzew TB. Do drzewa TR wprowadzane są dane o losowych wartościach, przez co wysokość jest większa od wartości optymistycznej $O(\log n)$. Natomiast do drzewa TB wprowadzane dane pochodzą z podziału połówkowego posortowanej tablicy, stworzone tak drzewo będzie dokładnie wyważone, a jego wysokość optymalna. Gdyby dane wprowadzano do BST według tablicy B, czyli posortowane rosnącą, drzewo przyjęło by postać jednej gałęzi o długości odpowiadającej liczbie wprowadzonych elementów, jego wysokość osiągnęła by wartość pesymistyczną $O(n)$.

Wysokość drzewa przeszukiwań binarnych ma bardzo duże znaczenie. To właśnie od niej zależy maksymalna ścieżka jaką musi przejść algorytm w poszukiwaniu elementów czy podczas próby dodania nowego. Najkorzystniejsze jest drzewo wyważone. Ma to związek z wysokością struktury, średni i optymistyczny czas przeszukiwania będzie wynosił $O(\log_n)$. Najmniej korzystne jest drzewo zdegenerowane do listy, oznacza to, że wszystkie elementy maja wpływ na wysokość drzewa, to zjawisko można osiągnąć wprowadzając do drzewa dane w posortowanej kolejności. Dla tego przypadku czas wyszukiwania zwiększa się do $O(n)$. Oznacza to, że struktura będzie działać najwydajniej gdy jest wyważona. Zmniejsza to średni dostęp do wyszukiwanego elementu.

4 Wady i zalety wybranych struktur danych

4.1 Tablica:

Zalety:

- krótki czas przeszukiwania binarnego
- dostęp do elementu po numerze indeksu
- łatwa implementacja (można skorzystać ze struktury standardowo zaimplementowanej w większości języków programowania)
- krótki czas tworzenia
- niskie wymagania pamięciowe

Wady:

- długi czas przeszukiwania wycieńczającego
- mała elastyczność (utrudniona możliwość dodania i usuwania elementów)

4.2 Lista jednokierunkowa:

Zalety:

- krótki czas tworzenia
- stosunkowo łatwa w implementacji
- duża elastyczność (łatwe dodawanie i usuwanie elementów)

Wady:

- długi czas przeszukiwania
- brak dostępu do elementu po numerze indeksu
- wyższe wymagania pamięciowe od tablicy

4.3 Drzewo poszukiwań binarnych

Zalety:

- krótki czas przeszukiwania w przypadku optymistycznym
- średnia elastyczność (możliwość dodawania i usuwania elementów)

Wady:

- brak dostępu do elementu po numerze indeksu
- największe wymagania pamięciowe z badanych struktur
- najtrudniejsza w implementacji z badanych struktur
- w przypadku pesymistycznym złożoność przeszukiwania wynosi $O(n)$
- średnia elastyczność (dodanie i usuwanie elementów jest trudniejsze do implementacji niż w liście jednokierunkowej i może mieć wysoką złożoność dla elementów znajdujący się daleko od korzenia)
- wrażliwa na dane wejściowe (w przypadku pesymistycznym, kiedy dane wejściowe są posortowane, drzewo przyjmie formę zdegradowanej lisy, złożoność tworzenia osiąga $O(n^2)$)

Spis treści