Sprawozdanie nr 2
01.04.2022, Ćwiczenia nr 5.
Listy i drzewa BST.

ALGORYTMY I STRUKTURY DANYCH semestr letni, rok akademicki 2021/2022 piątek 9:45-11:15

1. Andrei Kartavik, Indeks: 153925
2. Ivan Kaliadzich, Indeks: 153936

Wstęp

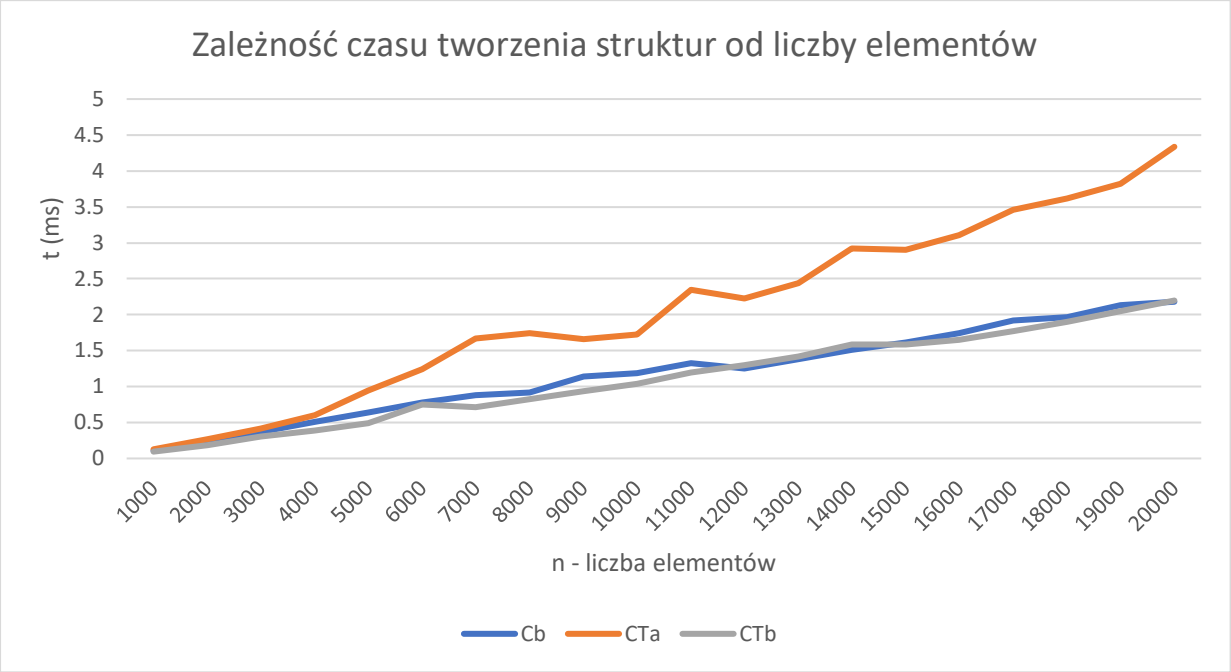
Celem tego sprawozdania jest porównanie różnych operacji dla tablicy (statycznej implementacji listy), drzewa BST zrównoważonego i drzewa BST niezrównoważonego.

Pomiary były wykonywane dla *n* niepowtarzających się liczb dodatnich całkowitych z zakresu [1, n], o wielkości *n* od 1000 elementów do 20 000 elementów, z krokiem 1 000 (20 punktów pomiarowych).

Zależność czasu tworzenia struktur od liczby elementów

n	C_b	C_{Ta}	C_{Tb}
1000	0,109	0,127	0,095
2000	0,251	0,264	0,184
3000	0,367	0,418	0,306
4000	0,511	0,605	0,386
5000	0,642	0,945	0,49
6000	0,782	1,242	0,751
7000	0,882	1,665	0,713
8000	0,919	1,745	0,829
9000	1,137	1,664	0,933
10000	1,187	1,72	1,04
11000	1,33	2,349	1,193
12000	1,252	2,222	1,296
13000	1,38	2,442	1,421
14000	1,508	2,92	1,587
15000	1,613	2,903	1,585
16000	1,74	3,109	1,651
17000	1,919	3,458	1,768
18000	1,967	3,621	1,902
19000	2,131	3,819	2,051
20000	2,18	4,336	2,197

Tablica 1: Zależność czasu tworzenia struktur od n liczby elementów (ms)



Zaczniemy od czasu tworzenia tablicy B (statycznej implementacji listy). Na ten czas wpływają dwa czynniki: czas kopiowania oryginalnej tablicy A do tablicy B, co wynosi O(*n*), ponieważ musimy wstawić każdy element tablicy A do tablicy B, i czas sortowania, metodą szybką, co wynosi $O(n^2)$ w przypadku najgorszym, O(nlogn) w przypadku średnim. W sumie to ma złożoność $O(n^2)$ w przypadku najgorszym i O(nlogn) w przypadku średnim.

Binarne drzewo poszukiwań (BST) – dynamiczna struktura danych będąca drzewem binarnym, w którym lewe poddrzewo każdego węzła zawiera wyłącznie elementy o kluczach mniejszych niż klucz węzła a prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła. Drzewo BST (T_B) zrównoważone (to takie drzewo BST, w którym wysokość lewego i prawego poddrzewa dowolnego węzła różnie się o nie więcej niż 1) budujemy na podstawie tablicy pomocniczej B_c, w której kolejność elementów jest zgodna z kolejnością metody dzielenia połówkowego posortowanej tablicy B. To znaczy, że kolejność jest zgodna z sytuacją, gdybyśmy budowali drzewo BST zrównoważone na podstawie listy posortowanej, biorąc średni element tablicy i robiąc go korzeniem, i powtarzali to rekurencyjnie dla lewego węzła i lewej połowy tablicy, oraz dla prawego węzła i prawej połowy tablicy… Musimy przejść przez każdy element tablicy, co wynosi O(n) i wstawić go na odpowiednią pozycję w naszym drzewie, i z tego powodu, że kolejność elementów w tablicy B_c jest zgodna z metodą dzielenia połówkowego, to daje nam możliwość przechodzenia maksymalnie przez połowę tego drzewa przy wstawianiu kolejnego elementu, co daje nam

złożoność O(logn). Z czego wynika, że w przypadku najgorszym czas tworzenia drzewa BST zrównoważonego wynosi O(nlogn).

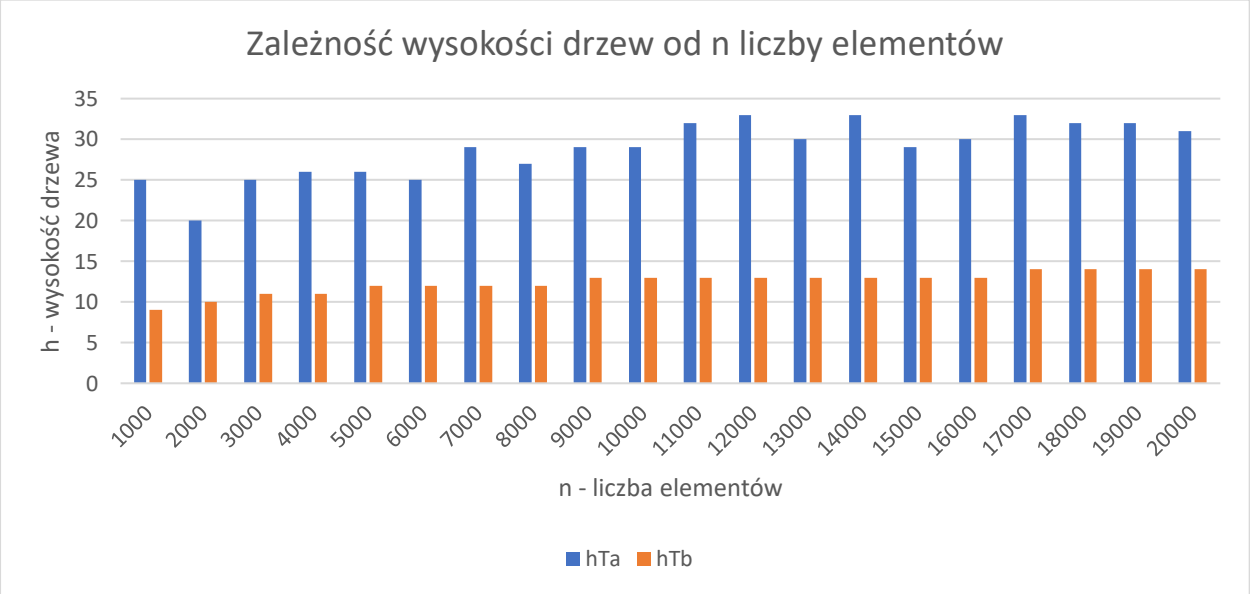
Drzewo BST (T_A) budujemy na podstawie oryginalnej tablicy A, gdzie kolejność elementów jest całkiem losowa, z czego i wynika dłuższy czas tworzenia takiego drzewa w porównaniu z tablicą B, oraz drzewem BST zrównoważonym T_B. Ale jak dokładnie ta kolejność wpływa na czas tworzenia tego drzewa? Ponieważ tworzenie tej struktury ma taki sam pomysł jak i budowa drzewa BST zrównoważonego – przechodzenie przez każdy element i wstawiania go na odpowiednią pozycję, w przypadku tablicy o losowej kolejności elementów może się zdarzyć, że w prawej, lub lewej części drzewa, będzie znacznie więcej elementów, niż w przeciwnej części. Np. taka sytuacja, gdy wszystkie węzły tego drzewa mają dokładnie jeden wierzchołek z prawej, albo z lewej strony, z czego wynika, że wysokość takiego drzewa (Wysokość drzewa, to poziom najgłębszego spośród węzłów (liści), czyli maksimum z poziomów wszystkich węzłów) będzie równa n elementów, co spowoduje, że za każdym razem przy wstawianiu kolejnego elementu będziemy musieli przejść przez całe drzewo, skąd i wynika złożoność O(n). W przypadku najgorszym złożoność tworzenia takiego drzewa jest równa $O(n^2)$, w przypadku najlepszym (gdy drzewo BST okaże się drzewem zrównoważonym) - O(nlogn).

Podsumowując, drzewo BST zrównoważone (T_B) ma najszybszy czas tworzenia, z tego powodu, że w przypadku najgorszym i średnim złożoność czasowa wynosi O(nlogn). Tworzenie tablicy B zajmuje trochę dużej czasu, ze względu na właściwości szybkiej metody sortowania, której złożoność czasowa zależy od wybranego elementu podziału i w najgorszym przypadku może wynosić $O(n^2)$. Tworzenie drzewa BST niezrównoważonego (T_A) obejmuje dużej czasu, z powodu tego, że za każdym razem wysokość drzewa się wariuje, co sprawia, że w najgorszym przypadku złożoność czasowa budowania tej struktury wynosi $O(n^2)$.

Zależność wysokości drzew od liczby elementów

n	h_{Ta}	h_{Tb}
1000	25	9
2000	20	10
3000	25	11
4000	26	11
5000	26	12
6000	25	12
7000	29	12
8000	27	12
9000	29	13
10000	29	13
11000	32	13
12000	33	13
13000	30	13
14000	33	13
15000	29	13
16000	30	13
17000	33	14
18000	32	14
19000	32	14
20000	31	14

Tablica 2: Zależność wysokości drzew od n liczby elementów



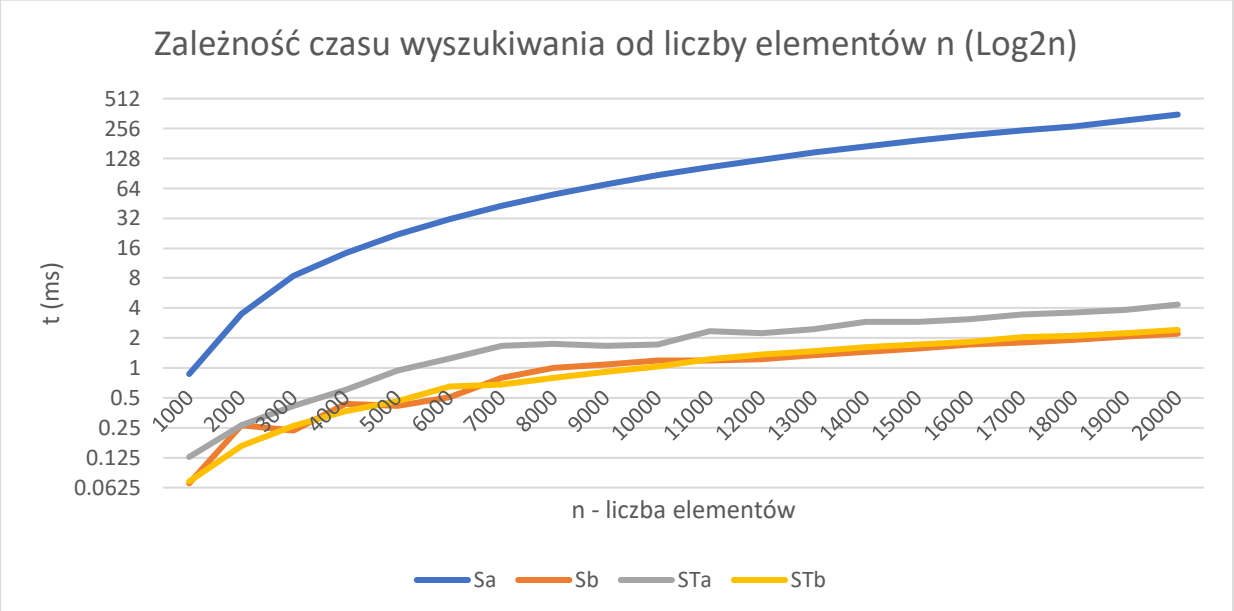
Wysokość drzewa jest istotna, bo ona wpływa na czas wstawiania elementów (im większa jest wysokość, tym więcej czasu może zająć wstawienie elementu), oraz ma ona wpływ na czas wyszukiwania elementów, co omówimy dalej w sprawozdaniu.

Ponieważ drzewo T_B jest zrównoważone, to ono zawsze ma wysokość ≈ $\log_2 n$, co i możemy zobaczyć na wykresie powyżej, a wysokość drzewa T_A, które nie jest drzewem zrównoważonym, zawsze się wariuje, ponieważ kolejność elementów ma duży wpływ na wysokość takiego drzewa (kolejność może spowodować przekrzywienie drzewa w lewą, lub prawą stronę). Gdyby np. wprowadzono elementy wg tablicy B (tablicy posortowanej), mielibyśmy wysokość drzewa h = *n*, gdzie *n* – liczba elementów, z tego powodu, że na każdym poziome wierzchołek miałby dokładnie jeden węzeł, większy od niego samego i byłoby te drzewo przekrzywione w prawo. W ogóle drzewo T_B jest korzystniejsze, bo będzie zajmowało mniej czasu dla wstawiania i wyszukiwania elementów, ale w przypadku, gdy zaczynamy usuwać elementy, lub wstawiać elementy o losowych wartości, to może ono stracić swoje własności drzewa zrównoważonego, co będzie powodowało potrzebność ponownego wyważania tego drzewa. Dlatego warto stosować drzewo BST zrównoważone, zamiast drzewa BST niezrównoważonego, w przypadkach, gdy potrzebujemy częstego wyszukiwania elementów i nie musimy często dodawać/usuwać elementy z takiego drzewa.

Zależność czasu wyszukiwania od liczby elementów

n	Sa	Sb	STa	STb
1000	0,868	0,069	0,127	0,072
2000	3,472	0,264	0,264	0,163
3000	8,378	0,235	0,418	0,263
4000	14,309	0,433	0,605	0,367
5000	21,944	0,415	0,945	0,459
6000	31,539	0,507	1,242	0,654
7000	42,629	0,794	1,665	0,676
8000	55,791	0,994	1,745	0,788
9000	70,211	1,083	1,664	0,913
10000	86,492	1,183	1,72	1,038
11000	104,046	1,18	2,349	1,216
12000	124,683	1,218	2,222	1,372
13000	147,762	1,337	2,442	1,481
14000	168,65	1,459	2,92	1,609
15000	195,136	1,57	2,903	1,709
16000	218,058	1,706	3,109	1,815
17000	243,963	1,799	3,458	2,028
18000	269,941	1,927	3,621	2,114
19000	308,761	2,077	3,819	2,238
20000	353,498	2,208	4,336	2,41

Tablica 3: Zależność czasu wyszukiwania od n liczby elementów (ms)



Jak już powiedzieliśmy, wysokość drzewa ma wpływ na czas wyszukiwania elementów w drzewie BST. Ale z czego to wynika? Wyobraźmy sobie taką sytuację, że mam drzewo BST zrównoważone i chcemy znaleźć w nim element X. Zaczynamy od korzenia, porównując wartość tego wierzchołka z wartością X, jeżeli wartości są równe, zwracamy wynik, jeżeli wartość X jest mniejsza – przechodzimy do lewego węzła, jeżeli jest większa – przechodzimy do prawego. Kontynuujemy tą operację, aż nie dostaniemy tego elementu lub nie dojdziemy do liścia tego drzewa. Dlaczego przytoczyliśmy ten przykład? Dlatego, że patrząc w definicję wysokości drzewa (Wysokość drzewa, to poziom najgłębszego spośród węzłów (liści), czyli maksimum z poziomów wszystkich węzłów), możemy zrozumieć, że im więcej jest wysokość drzewa, tym więcej tych operacji będziemy musieli zrobić, żeby odnaleźć nasz element X, więc złożoność wyszukiwania elementu wynosi O(h), gdzie h – wysokość drzewa. Dlatego czas wyszukiwania kolejnych elementów z listy A w drzewie BST zrównoważonym, gdzie wysokość h wynosi ≈ $\log_2 n$, ma złożoność O(nlogn) (z powodu tego, że wyszukujemy każdy element tablicy A), jest mniejszy, niż czas wyszukiwania kolejnych elementów w drzewie BST niezrównoważonym, gdzie wysokość h może wynosić *n* elementów i mieć złożoność $O(n^2)$ w przypadku najgorszym.

Porównujmy czas wyszukiwania kolejnych elementów tablicy A w drzewie BST zrównoważonym (T_B) i w tablice posortowanej B, zgodnie z metodą dzielenia połówkowego. Jak możemy zobaczyć na wykresie, wyniki są bardzo podobne, przecież wyszukiwanie połówkowe, jak i poszukiwanie w drzewie BST zrównoważonym wynosi O(logn) w przypadku najgorszym. Ale poszukiwanie w tablice posortowanej zgodnie z metodą wyszukiwania binarnego jest trochę szybsza niż poszukiwanie kolejnych elementów w drzewie BST zrównoważonym. To może wynikać z kilka powodów: 1) Aby uzyskać dostęp do pojedynczej wartości w tablicy wystarczy wykonać tylko jeden odczyt, dla drzewa potrzebujemy dwóch: lewego lub prawego wskaźnika oraz wartości. 2) Węzły w drzewie są alokowane na stogu (heap) i mogą one znajdować się wszędzie w pamięci, w zależności od kolejności ich alokacji (również stóg może zostać pofragmentowany) - a tworzenie tych węzłów (z new lub alloc) również zajmie więcej czasu w porównaniu z możliwą jednorazową alokacją dla tablicy. Generalnie, jeśli jest ta metoda podobna, lub szybsza niż wyszukiwanie w drzewie BST zrównoważonym, to jest ona szybsza niż wyszukiwanie w drzewie BST niezrównoważonym.

Jak możemy zobaczyć na wykresie, czas wyszukiwania kolejnych elementów posortowanej tablicy B w tablice A jest największym w porównaniu ze wszystkimi innymi metodami. Wynika to z tego, że za każdym razem, dla kolejnego elementu z listy B, musimy porównywać go z każdym elementem tablicy A, aż nie znajdziemy tego elementu w tablice A. Więc to wynosi $O(n^2)$ w przypadku najgorszym.

Podsumowując, dla wyszukiwania kolejnych elementów najlepszymi są: metoda wyszukiwania binarnego w tablice posortowanej oraz poszukiwanie w drzewie BST zrównoważonym, bo mają lepszą złożoność czasową oraz są szybsze zgodnie z wykresem powyżej. Wyszukiwanie kolejnych elementów w drzewie BST niezrównoważonym jest lepsze, niż wyszukiwanie kolejnych elementów w tablice o losowej kolejności, ponieważ wyszukiwanie elementu w drzewie BST wynosi O(h), gdzie h – wysokość tego drzewa, a dla tablicy to O(n). I z tego powodu, że h będzie równe n tylko w przypadku najgorszym, średni czas wyszukiwania kolejnych elementów w drzewie BST będzie mniejszy.

Zajętość pamięciowa poszczególnych struktur

Tablica potrzebuje $n * d$ bajtów, gdzie n – to liczba elementów, a d – to liczba bajtów, którą zajmuje jeden element w zależności od tupu danych. Np. dla liczby całkowitej d = 4 bajty, więc tablica liczb całkowitych zajmowałaby $4*n$ bajtów.

Z drzewem BST jest trochę trudniej, z powodu tego, że każdy element drzewa zawiera nie tylko dane, ale jeszcze dwa wskaźniki do lewego i prawego węzłów (nawet jeśli mają wartość NULL). Dlatego taka struktura danych potrzebuje $n*(2P + d)$ bajtów, gdzie n – liczba elementów, P – liczba bajtów, którą zajmuje wskaźnik, d – liczba bajtów, którą zajmują dane poszczególnego elementu. Np. dla danych typu liczby całkowitej: d = 4 bajty, P = 4 bajty w 32 bitach i 8 bajtów w 64 bitach, w tym przypadku wezmę 8 bajtów (z powodu tego, że 64 bitnych układów staje się więcej i więcej), w takim razie jeden element będzie zajmował 20 bajtów, a całe drzewo BST $20 * n$ bajtów.

Dlatego, ze względu na zajętość pamięciową, tablice są lepsze niż drzewa BST.

Porównanie struktur i podsumowanie

W porównaniu z innymi strukturami danych tablice są bardzo dobre ze względu na zajętość pamięciową, bo nie muszą przechowywać np. wskaźników na następne elementy itd. Mają natychmiastowy dostęp do elementów O(1), mogą przechowywać różne typy danych, są wbudowane w wielu językach programowania, są bardzo proste do zrozumienia. Ale ich główną wadą jest to, że nie są one dynamiczne, jak np. listy jednokierunkowe i drzewa BST… Kiedy tworzymy tablicę, musimy od razu określić maksymalną liczbę jej elementów, co jest bardzo niewygodne, kiedy chodzi o dodawaniu nowych danych. Oraz tablice mają niezbyt dobrą złożoność czasową dla operacji wstawiania, usuwania i wyszukiwania elementów – O(n). Chociaż tablice stają się dobrą strukturą danych do wyszukiwania elementów, gdy są one posortowane, bo w takim przypadku, wykorzystując metodę wyszukiwania binarnego, wynosi to O(logn), co jest bardzo dobrym wynikiem! Ale warto pamiętać, że przy wstawianiu, lub usuwaniu elementu z tablicy posortowanej, będziemy musieli ją ponownie posortować, co oczywiście dla dużej liczby elementów może być bardzo czasochłonnie.
Dlatego, warto korzystać z tablic w tych przypadkach, kiedy mamy stosunkowo niewiele danych lub kiedy musimy po prostu przechowywać jakieś dane i chcemy mieć do nich natychmiastowy dostęp.

Listy jednokierunkowe, w odróżnieniu od tablic są dynamiczne. Dlatego zajmują więcej pamięci, bo oprócz danych, każdy element listy jednokierunkowej musi zawierać wskaźnik na następny element. Są trochę trudniejsze ze względu na implementacje, ale nadal są prostsze niż implementacje drzew BST. Z powodu tego, że listy jednokierunkowe wykorzystują wskaźniki na każdy następny element, one są gorsze w sensie dostępu elementów, bo ta operacja ma złożoność czasową O(n). Ale to daje i swoje korzyści, bo przy usuwaniu, lub wstawianiu elementów nie musimy przemieszczać każdy element, przecież wszystkie elementy są powiązane wskaźnikami, co nam i daje złożoność czasową O(1) dla tych operacji. Dla wyszukiwania elementów nie jest ona najlepszą strukturą, bo nadal musimy przejść przez całą tablicę, żeby znaleźć odpowiedni element. Możemy w takim razie posortować list jednokierunkowy, ale to jest dosyć skomplikowanie ze względu na implementację zaś warto pamiętać, że jeśli dodamy lub usuniemy element, to będziemy musieli ponownie ją posortować.

Dlatego jeśli głównym celem jest wyszukiwanie elementów, to lepiej skorzystać z tablicy lub drzewa BST. Ale jeśli często wstawiamy, usuwamy dane i nie musimy mieć natychmiastowego dostępu do nich, to lista jednokierunkowa może być bardzo korzystna w takich celach.

Ostatnia struktura to drzewa BST. Zajmują one najwięcej pamięci w porównaniu z innymi omówionymi strukturami danych, bo każdy element zawiera dane i dwa wskaźniki, co wymaga prawie 5 razy więcej pamięci, niż tablicy i w prawie 1.5 razy więcej, niż listy jednokierunkowe. Są trudniejsze ze względu na implementację, w porównaniu z innymi strukturami. Ale ich główną zaletą jest to, że mają one zalety zarówno tablic, jak i list jednokierunkowych… Bo są one dynamicznymi, jak i listy jednokierunkowe, mają stosunkowo szybki dostęp do elementów, jak i tablicy, oraz są stosunkowo szybkie dla wstawiania, usuwania, wyszukiwania elementów. Warto zaznaczyć, że to bardzo zależy od tego, czy drzewo BST jest zrównoważone, czy nie. W przypadku gdy drzewo BST nie jest zrównoważonym, to traci ono prawie wszystkie swoje zalety i staje się bardzo podobnym do listy jednokierunkowej. Bo jak już omówiono wcześniej, czas wstawiania, usuwania, wyszukiwania, dostępu elementu w przypadku najgorszym może osiągać O(n). Ale gdy te drzewo jest zrównoważone, to najgorsza złożoność czasowa tych wszystkich operacji wynosi O(logn), co i robi tą strukturę taką korzystną. Dlatego warto ją wykorzystywać, jak już było powiedziano, w tych sytuacjach, kiedy potrzebujemy częstego wyszukiwania elementów i nie musimy często dodawać/usuwać elementy z takiego drzewa, bo może spowodować to, że straci ono swoją strukturę i będziemy musieli ponownie wyważać te drzewo.