**Algorytmy i Struktury Danych – Laboratoria**

Dr. Hab. Inż. Małgorzata Sterna

Informatyka, semestr 2, grupa i2, wtorki godz. 13.30

21.03.2017r.

**Ćwiczenie nr.2 – BST (Binary Search Tree)**

Krzysztof Pasiewicz, 132302

Mikołaj Frankowski, 132220

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quant | CB | CL | CTR | CTB |
| 10000 | 0,001 | 0,001 | 0,005 | 0,002 |
| 20000 | 0,004 | 0,003 | 0,006 | 0,004 |
| 30000 | 0,005 | 0,005 | 0,01 | 0,007 |
| 40000 | 0,006 | 0,007 | 0,015 | 0,009 |
| 50000 | 0,008 | 0,009 | 0,018 | 0,011 |
| 60000 | 0,011 | 0,012 | 0,021 | 0,014 |
| 70000 | 0,012 | 0,013 | 0,025 | 0,015 |
| 80000 | 0,012 | 0,015 | 0,029 | 0,018 |
| 90000 | 0,015 | 0,015 | 0,037 | 0,02 |
| 100000 | 0,015 | 0,018 | 0,036 | 0,023 |

Tab.1 – Tworzenie poszczególnych struktur

Wyk. 1 – Tworzenie poszczególnych struktur.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quant | SsB | SbB | SL | STR | STB |
| 10000 | 0,128 | 0,002 | 0,329 | 0,001 | 0,001 |
| 20000 | 0,487 | 0,003 | 1,336 | 0,004 | 0,003 |
| 30000 | 1,074 | 0,005 | 3,081 | 0,006 | 0,007 |
| 40000 | 1,883 | 0,007 | 5,394 | 0,009 | 0,008 |
| 50000 | 2,994 | 0,008 | 8,612 | 0,011 | 0,01 |
| 60000 | 4,336 | 0,011 | 12,933 | 0,014 | 0,011 |
| 70000 | 5,835 | 0,019 | 17,817 | 0,019 | 0,014 |
| 80000 | 7,578 | 0,015 | 24,205 | 0,02 | 0,016 |
| 90000 | 9,424 | 0,016 | 30,324 | 0,027 | 0,023 |
| 100000 | 12,365 | 0,019 | 42,495 | 0,029 | 0,022 |

Tab. 2 – Czas wyszukiwania w poszczególnych strukturach.

Wyk. 2 - Czas wyszukiwania w poszczególnych strukturach.

|  |  |  |
| --- | --- | --- |
| Quant | HTR | HTB |
| 10000 | 32 | 14 |
| 20000 | 32 | 15 |
| 30000 | 36 | 15 |
| 40000 | 37 | 16 |
| 50000 | 41 | 16 |
| 60000 | 39 | 16 |
| 70000 | 39 | 17 |
| 80000 | 43 | 17 |
| 90000 | 38 | 17 |
| 100000 | 42 | 17 |

Tab. 3 – Wysokość drzew

Wyk. 3 – Wysokość drzew.

Spośród drzew TR i TB te pierwsze mają wyraźnie większą wysokość. Dzieje się tak, ponieważ w przypadku drzewa TB wprowadzanie danych w kolejności wynikającej z dzielenia połówkowego powoduje stworzenie drzewa dokładnie wyważonego, czyli takiego, w którym dla każdego elementu liczba elementów w jego lewym i prawym poddrzewie różni się co najwyżej o jeden. To z kolei gwarantuje osiągnięcie optymalnej, jak najmniejszej, wysokości równej . Właśnie to powoduje, że drzewo TB jest najkorzystniejszym możliwym rozwiązaniem. Wysokość drzewa TR jest natomiast wrażliwa na dane wejściowe i waha się między , w najlepszym przypadku, a , w najgorszym. Dzieje się tak, ponieważ liczby są losowane i nie da się przewidzieć w jaką sekwencję się ułożą. Inną możliwością jest wpisanie elementów do drzewa według tablicy B. To jednak zawsze skutkuje wysokością równą , ponieważ posortowane liczby układają się w jedną gałąź maksymalnie ją wydłużając, a co za tym idzie, zwiększając wysokość. Ten fakt czyni omawiane właśnie drzewo najmniej korzystnym.

Wysokość drzewa binarnego ma bardzo duże znaczenie- to od niej zależy złożoność algorytmów operacji na nich wykonywanych, takich jak dodawanie czy wyszukiwanie elementów. Im większa wysokość tym mniej korzystna złożoność. Jeżeli chodzi o BST, złożoność tworzenia tej struktury wynosi od O() do nawet O(). Natomiast czas wyszukiwania elementu w drzewie jest rzędu O(), gdzie h oznacza wysokość. Już sam ten fakt pokazuje jej ogromne znaczenie dla tej struktury. Na samą wysokość ma natomiast duży wpływ ułożenie danych wejściowych. Najkorzystniejszą możliwością jest wprowadzanie danych w kolejności wynikającej z dzielenia połówkowego () a najmniej korzystną wpisywanie według posortowanej tablicy (). Aby osiągnąć przypadek optymistyczny jeżeli chodzi o wysokość stosuje się wyważanie BST. Nie zawsze jednak jest to opłacalne. Samo wyważanie też zajmuje pewien czas spowalniając pracę programu. Powoduje też utrudnienia związane z dodawaniem nowych elementów. Dlatego metodę te stosuje się głównie, gdy duży jest stosunek liczby wyszukiwań do wstawień.

Porównując wyszukiwanie w tablicy B i w liście L zauważyć można, że znacznie korzystniejsza jest pierwsza struktura. Dzięki możliwości zastosowania wyszukiwania binarnego złożoność operacji wynosi zaledwie O(). Natomiast w liście, z powodu nienaturalnego dostępu do elementów, nie można zastosować powyższej metody. Trzeba zatem przejść przez wszystkie elementy aż do szukanego. To skutkuje bardzo niekorzystną złożonością- O().

Wyszukiwanie połówkowe w tablicy jak i wyszukiwanie w zrównoważonym drzewie BST cechują się bardzo podobnym czasem wyszukiwania – złożoność O(log2n). Obie struktury wymagają uprzedniego posortowania danych, drzewo BST w odróżnieniu od listy ma dłuższy czas tworzenia, ale jest lepsze jeśli chcemy dodawać do niego nowe elementy czego nie można zrobić w przypadku tablicy.

Wszystkie powyższe struktury charakteryzują się taką samą ogólną złożonością pamięciową i wynosi ona O(n). Jednakże róznice możemy dostrzec w zajmowanej przez pojedyncze elementy ilości pamięci. W Tablicy należy zapamiętać tylko wartość elemnentu, w Liście komputer zapisuje zarówno wartość elementu jak i wskaźnik na kolejny element listy, a w drzewie BST każdy element składa się z wartości oraz dwóch wskaźników: na prawy i lewy liść.

Wady i zalety tablic, list i drzew BST:

**TABLICE:**

Zalety:

- dowolny dostęp do elementów,

-szybki czas tworzenia – liniowy, wydłużający się wraz ze wzrostem ilości elementów,

-łatwość implementacji (gotowa struktura),

-najmniejsze wymagania pamięciowe,

-łatwa do posortowania,

Wady:

-niewielkie możliwości manipulacji (ciężko dodać lub usunąć element, brak możliwości zmiany wielkości struktury),

-przeszukiwanie liniowe bardzo czasochłonne,

**LISTA:**

Zalety:

-krótki czas tworzenia,

-duże możliwości manipulacji (łatwość dodawania i usuwania elementu oraz zmiany wielkości struktury),

- łatwość implementacji (gotowy obiekt lub prosta struktura),

Wady:

- trudna do posortowania,

-czasochłonny dostęp do elementów,

-długi czas wyszukiwania elementu – zawsze złożoność O(n),

**BST:**

Zalety:

-krótki czas wyszukiwania (O(log2n), pesymistycznie O(n)),

-łatwiejsze dodawanie elementu niż w liście, chociaż wymaga wielu operacji porównania,

Wady:

-najwieksze wymagania pamięciowe,

-najtrudniejsza w implementacji,

-duża wrażliwość na dane wejściowe,

-najdłuższy czas tworzenia O(n\*log2n)