PAMSI - Sprawozdanie 2

Filip Guzy 218672

26 marca 2016

Implementacja struktur danych

W celu efektywniejszej realizacji struktur danych opartych na tablicy dynamicznej stworzono klasę wirtualną (odpowiednik interfejsu w języku Java lub C#) DataStructure, zawierający metodę wirtualną add_num() służącą do dodawania pojedynczych elementów do struktury danych (w tym przypadku do tablicy dynamicznej). Zaimplementowano również klasę wirtualną MainTimer zawierającą metody możliwe do wykorzystania przy pomiarach czasu wykonania algorytmów, takie jak: get_ms_time() - mierzącą czas w milisekundach w konkretnej chwili wykonywania algorytmów, tim_start() oraz tim_stop(), wykonujące pomiary początku i końca danej operacji, a także metodę return_time() zwracającą czas zmierzony przez wykorzystanie metod stopera. Obie klasy znajdują się w pliku struktura.hh. W pliku algorytmy.hh zawarto klasy dziedziczące po klasach DataStructure oraz MainTimer, które wykorzystują metody klas wirtualnych. Klasa Timer dziedzicząca po klasie MainTimer korzysta z biblioteki systemowej time.h w celu zaimplementowania metod mierzących czas. Programista dzięki metodom wirtualnym klasy opisującej stoper ma możliwość dostosowania sposobów pomiaru czasu do swojego systemu operacyjnego.

Statystyczna analiza efektywności algorytmów

W celu uzyskania wartości możliwie najlepiej odzwierciedlających czasy wykonania poszczególnych algorytmów zapełniających tablice powtórzono pomiary dziesięciokrotnie dla każdej ilości danych testowych. Zastosowano odpowiednie oznaczenia dla poszczególnych algorytmów:

1. $k = 2 * k$ - algorytm podwajający rozmiar tablicy dynamicznej w przypadku każdego przepełnienia.
2. $k = k + 100$ - algorytm powiększający rozmiar tablicy dynamicznej o 100 w przypadku każdego przepełnienia.
3. $k = k + 1$ - algorytm powiększający rozmiar tablicy dynamicznej o jeden w przypadku każdego przepełnienia.

Na wejście programu podano ilość danych do wczytania, a na wyjściu otrzymano czas realizacji operacji (w milisekundach) dla każdego z algorytmów. Wszystkie pomiary przedstawiono w poniższej tabeli.

Ilość danych	$k = 2 * k$	$k = k + 100$	$k = k + 1$
10	0,000999928	0	0,000999928
	0,00199997	0	0,000999928
	0,000999928	0	0,00100005
	0,000999928	0,00100005	0
	0,00100005	0	0,00100005
	0,00200009	0	0
	0,00200009	0	0,00100005
	0,000999928	0,00100005	0,000999928
	0,00199997	0	0
	0,00100005	0	0,000999928
100	0,00999999	0,000999928	0,282
	0,00899994	0,00199997	0,053
	0,00999999	0,00199997	0,054
	0,00999999	0,00199997	0,068
	0,00899994	0,00199997	0,053
	0,00699997	0,00199997	0,041
	0,00899994	0,00200009	0,082
	0,0100001	0,000999928	0,053
	0,00999999	0,00199997	0,054
	0,00999999	0,00199997	0,054

Ilosc danych	$k = 2 * k$	$k = k + 100$	$k = k + 1$
1000	0,028	0,0439999	3,884
	0,0270001	0,064	3,08
	0,027	0,0450001	4,206
	0,027	0,0470001	3,81
	0,029	0,044	3,832
	0,028	0,046	3,932
	0,0300001	0,045	3,852
	0,028	0,046	3,062
	0,0280001	0,045	3,883
	0,028	0,044	3,01
10000	0,176	3,721	277,261
	0,179	3,769	271,885
	0,204	3,056	275,169
	0,131	2,817	274,995
	0,172	3,732	279,963
	0,179	3,752	281,165
	0,218	3,708	291,788
	0,18	3,721	278,057
	0,175	3,708	279,129
	0,174	3,711	298,66
100000	2,411	280,809	27325,1
	2,04	299,738	27259,6
	1,998	277,449	27388
	2,383	279,887	26940,4
	2,432	274,153	27437,9
	2,406	274,572	27211,1
	2,395	288,284	27714,9
	2,443	278,77	27048,1
	2,415	272,211	27123,8
	2,577	275,387	27212,2
1000000	77,995	27465,3	-
	53,406	27730,6	-
	70,836	28001,9	-
	78,013	27889,7	-
	18,959	27584,7	-
	119,444	27458,7	-
	92,748	27242,9	-
	79,159	27443,6	-
	77,683	27439,2	-
	79,407	27817,3	-
1000000000	13407,2	-	-
	16452,4	-	-
	13199,5	-	-
	13211,3	-	-
	13450,6	-	-
	13477,1	-	-
	13339,6	-	-
	13393,6	-	-
	13250,2	-	-
	13239,2	-	-

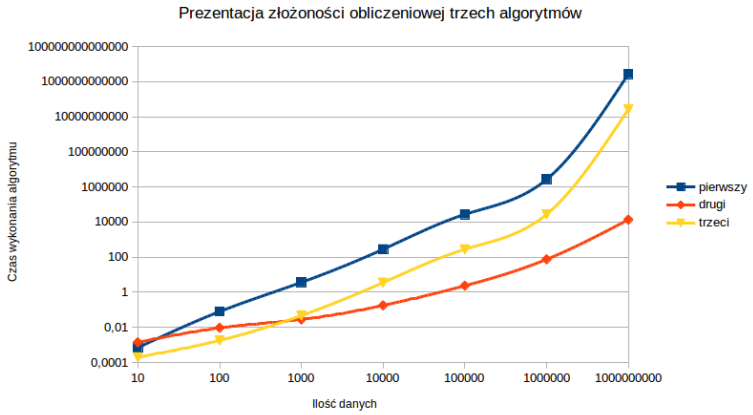
W tabeli nie umieszczono czasów wykonania algorytmów $k = k + 1$ dla miliona liczb oraz $k = k + 1$ i $k = k + 100$ dla miliarda liczb, ponieważ szacowany czas ich wykonania w pierwszym przypadku wynosi 7,5h, natomiast w drugim odpowiednio: 7500000h ($k = k + 1$) i 75000h ($k = k + 100$) co powoduje, że ich statystyczna ocena byłaby nieefektywna.

Stosując średnią arytmetyczną dla każdej serii danych otrzymujemy odpowiednie wartości:

Ilosc danych	$k = 2 * k$	$k = k + 100$	$k = k + 1$
10	0,0013999932	0,00020001	0,0006999862
100	0,009399984	0,0017999736	0,0794
1000	0,02800003	0,04700001	3,6551
10000	0,1788	3,5695	280,8071
100000	2,35	280,126	27266,11
1000000	74,765	27607,39	-
1000000000	13642,07	-	-

Poprawka z dnia 7.04.2016:
Złożoność obliczeniowa dla algorytmów powiększania o stałe C powinna być kwadratowa, czyli O(n^2), natomiast złożoność algorytmów podwajających powinna zawierać się w O(n). Jak można zauważyć, tak też jest w rzeczywistości, ponieważ zwiększając ilość danych dziesięciokrotnie czas wykonania algorytmów zwiększających o stałą C rośnie stukrotnie, natomiast algorytmu podwajającego dziesięciokrotnie. Najlepiej widać to dla ilości danych powyżej 1000, gdzie zaczynają pojawiać się różnice pomiędzy ich efektywnością.

Poprawka z dnia 9.04.2016:
Zależność czasu wykonania od ilości danych dla trzech algorytmów przedstawiono na poniższym wykresie. Kolorem niebieskim oznaczono $k = k + 1$, żółtym $k = k + 100$, a czerwonym $k = k * 2$.



Wnioski

Z przeprowadzonych pomiarów i po wykonaniu uśrednień można wywnioskować, że dla najmniejszych rozmiarów serii danych (do 10) wszystkie trzy algorytmy prezentują podobną efektywność. Przy nieco większych ilościach (do 1000) zbliżone czasy przedstawiają algorytmy $k = 2 * k$ oraz $k = k + 100$, natomiast mniej efektywny staje się $k = k + 1$. Najefektywniejszy dla największych ilości danych jest algorytm $k = 2 * k$ (do 10^9).

Przy większych projektach zespołowych warto wykorzystywać koncepcję interfejsów, ponieważ pozwala ona na implementację danych zgodną z powszechnie przyjętymi wzorcami projektowymi w programowaniu obiektowym, a co w tym najważniejsze umożliwia programistom uniknięcie kłopotów z dziedziczeniem klas w przypadku nagłych potrzeb rozszerzania funkcjonalności projektu.