|  |  |  |  |
| --- | --- | --- | --- |
|  | **Nazwa kursu:** Struktury Danych i Złożoność Obliczeniowa (projekt nr 2.) | | |
| **Temat projektu:** „Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych.” | | |
| **Autor:**  Piotr Ławniczak 209775 | **Data:** 12.04.2015 | |
| **Grupa:** wtorek 11:15 | |
| **Prowadzący:**  Dr inż. Jarosław Mierzwa | **Ocena:** |

# Cel projektu

Celem projektu jest dokładna analiza wydajności struktur danych takich jak: tablica, lista dwukierunkowa oraz kopiec binarny pod kątem czasu wykonywania operacji polegających na odczycie, zapisie jak i wyszukiwaniu danych w zależności od rozmiaru struktur na podstawie ich własnej implementacji.

# Wstęp teoretyczny

Struktura danych to pojęcie odnoszące się do sposobu uporządkowania danych w komputerze.  
Główne zastosowania tworzonego zbioru elementów implikują wybranie konkretnej struktury danych do jego obsługi. Różnią się one między sobą przede wszystkim szybkością w konkretnych sytuacjach (np. jedne są szybsze w obsłudze małej ilości danych, drugie zaś mogą lepiej sobie radzić z większą liczbą elementów).

Na podstawie literatury dowiaduję się o złożoności obliczeniowej opracowywanych przeze mnie struktur:

1. Tablica:

-wstawianie: O(n)

-usuwanie: O(n)

-wyszukiwanie (po przechowywanych wartościach): O(n)

1. Lista dwukierunkowa:

-wstawianie: O(1)

-usuwanie: O(1)

-wyszukiwanie: O(n)

1. Kopiec binarny:

-wstawianie: O(log n)

-usuwanie: O(log n)

-naprawianie kopca: O(n)

-wyszukiwanie: O(n)

# Plan eksperymentu

Czas potrzebny do wykonania danej operacji będzie mierzony przy użyciu licznika cykli procesora wykonanych od jego uruchomienia - TSC (Time Stamp Counter). Dostęp do niego umożliwia funkcja QueryPerformanceCounter. Jego wartość będzie odczytywana i zapisywana tuż przed rozpoczęciem mierzonego polecenia, oraz zaraz po. Różnica podzielona przez ilość cykli jakie procesor wykonuje w czasie sekundy (odczytaną przy użyciu funkcji QueryPerformanceFrequency) daje szczegółowy czas w sekundach do sześciu miejsc po przecinku. Elementem struktur będzie liczba typu integer generowana losowo z zakresu [-100; 100] za pomocą funkcji *rand()*. Na potrzeby badań napisałem prosty program. Jedną z jego ciekawszych funkcji jest zapisywanie raportu z ostatniego eksperymentu do pliku tekstowego o nazwie „raport.txt”. Na końcu takiego pliku znajdują Komputer na jakim zostaną przeprowadzone doświadczenia posiada czterordzeniowy procesor Intel i7 taktowany zegarem 3,00 GHz oraz 8 GB pamięci RAM DDR3 o częstotliwości taktowania 1600 MHz.

Kolejno wykonane zostaną:

1. Dodawanie losowego elementu na początek tablicy

2. Usuwanie elementu z początku tablicy

3. Dodawanie losowego elementu na koniec tablicy

4. Usuwanie elementu z końca tablicy o rozmiarze

5. Dodawanie elementu w losowe miejsce do tablicy o rozmiarze

6. Wyszukiwanie losowego elementu w tablicy

7. Usuwanie z losowego miejsca elementu z tablicy

8. Dodawanie elementu na początek listy

9. Usuwanie elementu z początku listy

10. Dodawanie elementu na koniec listy

11. Usuwanie elementu z końca listy

12. Dodawanie elementu w losowe miejsce

13. Wyszukiwanie losowego elementu w liście

14. Usuwanie elementu z losowego miejsca z listy

15. Dodawanie elementu do kopca binarnego

16. Wyszukiwanie losowego elementu w kopcu

17. Usuwanie losowego elementu z kopca

Doświadczenie zostanie powtórzone pięciokrotnie. Ilości elementów dla operacji to kolejno:  
dla tablicy: 2500, 5000, 10000, 20000, 30000,  
dla listy: 10000, 20000, 40000, 60000, 80000,  
dla kopca: 2500, 5000, 10000, 15000, 20000.

# Implementacja algorytmów struktur

1. Tablica
2. **dodawanie**

Alokowana jest nowa tablica o jeden element większa od obecnej, po czym startuje pętla przekopiowująca element po elemencie do niej. W zależności od podanego przez użytkownika indeksu na który należy wstawić podaną wartość – następuje przerwanie pętli na jeden krok oraz wstawienie nowego elementu. Pętla kontynuuje swoje działanie dopóki iterator nie dotrze do ostatniego miejsca w tablicy. Na koniec usuwana jest stara, nieaktualna już baza, a przypisywana jej referencja do tej nowo utworzonej.

1. **usuwanie**

Tworzona jest nowa tablica mniejsza o jeden element od obecnej, następnie startuje pętla która przekopiowuje element po elemencie do nowej tablicy. Jeśli iterator równy jest indeksowi elementu jaki chcemy usunąć, dany element nie jest kopiowany – zamiast tego następuje przeskok na kolejny i kopiowanie jest kontynuowane. Na koniec usuwana jest stara, nieaktualna już baza, a przypisywana jej referencja do tej nowo utworzonej.

1. **wyszukiwanie**

Wyszukiwanie danego elementu w tablicy to iteracja element po elemencie i porównywanie czy jego wartość nie jest równa wartości poszukiwanej.

1. Lista dwukierunkowa
2. **dodawanie**

Funkcja dodawania przyjmuje dwa argumenty – wartość oraz miejsce w które ma podaną liczbę zapisać. Tworzony jest nowy element z polem o podanej wartości. Następnie rozpatrywane są cztery warunki: podany indeks wychodzący po za zakres, indeks równy indeksowi członu, indeks równy indeksowi ogona oraz pozostałe przypadki (środkowe części listy). W ogólnym przypadku pod zmienną tymczasową przypisywana jest referencja do elementu pod zadanym indeksem. Następnie, element znajdujący się aktualnie pod zadanym indeksem jest „wypychany” o jednostkę dalej a w jego miejsce włączany nowoutworzony element.

1. **usuwanie**

Mechanizm usuwania polega na iteracji element po elemencie z wykorzystaniem obecnych wskaźników na następniki aż do znalezienia pierwszego wystąpienia elementu o poszukiwanej wartości bądź dotarcia do momentu w którym dany obiekt nie posiada już wskaźnika na element następny. Wskaźniki poprzednika i następnika są modyfikowane w ten sposób by „przeskoczyć” usuwany obiekt, wówczas pominięty w liście obiekt usuwany jest z pamięci przy pomocy komendy „delete”.

1. **wyszukiwanie**

Algorytm wyszukiwania jest najprostszym spośród trzech omawianych typów operacji na liście. Swoje działanie opiera na tymczasowym wskaźniku do którego najpierw przypisywany jest człon listy. Następnie element po elemencie sprawdzana jest równość zawierającego w nim pola z poszukiwaną przez użytkownika wartością.

1. Kopiec binarny
2. **dodawanie**

Na początku tworzona jest tablica o jeden większa od obecnej oraz nowy element który zostanie w nią wpisany. Następnie, element po elemencie zostają przekopiowane dane z tablicy starej do nowej, a na ostatnie wolne miejsce zostanie wpisany świeżo stworzony element którego wartość przekazywana jest jako parametr funkcji. Stara, niepotrzebna już tablica zostaje usunięta celem zwolnienia pamięci, następnie w miejsce wskaźnika do niej przypisywana jest referencja do nowej tablicy. Na koniec uruchomiona zostaje funkcja która „naprawia” kopiec, co przywraca mu jego najważniejszą własność polegającą na tym że potomek nie może być większy od rodzica.

1. **usuwanie**

Usuwanie w kopcu polega na przeniesieniu jego najbardziej wysuniętego w prawo liścia w miejsce korzenia (korzeń zostaje bezpowrotnie nadpisany). Następnie wywoływana jest funkcja naprawiania, która „spycha” nowo przypisany element na odpowiednie miejsce (zgodne z definicją kopca).

1. **Wyszukiwanie**

Ostatnia z operacji została zaimplementowana za pomocą funkcji rekurencyjnej. Rekurencja następuje element po elemencie od korzenia w dół, z każdym poziomem drzewa podwajając instancje funkcji wyszukiwania. Jej przypadkiem podstawowym jest wyjście poza zakres kopca, bądź odnalezienie poszukiwanego elementu. Dzięki zastosowaniu rekurencji i opracowaniu zwięzłego, dokładnego algorytmu - operacja jest wykonywana bardzo szybko.

# Wyniki eksperymentu

Na podstawie pięciu raportów uzyskanych z pięciu osobnych eksperymentów na każdej ze struktur sporządzam tabele, następnie, za ich pośrednictwem – wykresy. Wyniki dodawania oraz usuwania elementów zostały uśrednione po ich czasach ich wykonania w zależności od pozycji indeksu – początku, końca, lub miejsca losowego. W przypadku listy, celem uniknięcia zafałszowania wyników, pozycja losowa została wykluczona ze swojego udziału w uśrednianiu z powodu najprawdopodobniej błędu w implementacji, który przyczynił się do ogromnych rozbieżności czasowych. Wszystkie raporty z badań znajdują się w katalogu *raporty,* natomiast dane z nich przedstawione w postaci tabelarycznej – są zawarte także w arkuszu kalkulacyjnym *dane.xlsx*.

1. Tablica – dodawanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elem. | 2500 | 5000 | 7500 | 10000 | 12500 | 15000 | 17500 |
| początek [ms] | 0,020082 | 0,033231 | 0,043565 | 0,066305 | 0,082511 | 0,104714 | 0,113365 |
| losowo [ms] | 0,016954 | 0,026914 | 0,035941 | 0,046527 | 0,060235 | 0,073455 | 0,087012 |
| koniec [ms] | 0,018464 | 0,030746 | 0,042514 | 0,054459 | 0,069853 | 0,084149 | 0,100404 |

Tabela 1. Uśrednione czasy dodawania elementu z relokacją pamięci

1. Wykres zależności czasu dodawania elementu od rozmiaru tablicy

2. Wykres zależności czasu potrzebnego na dodanie elementu w losowe miejsce w zależności od rozmiaru tablicy

3. Wykres zależności czasu dodawania elementu na koniec od rozmiaru tablicy

Na podstawie powyżej przedstawionych wyników można stwierdzić, że czas potrzebny na dodanie jednego elementu jest wprost proporcjonalny do rozmiaru tablicy. Natychmiastowy dostęp do dowolnego indeksu wewnątrz tablicy sprawia, że miejsce w tablicy, w które dodajemy nowy element nie ma większego wpływu na czas operacji.

1. Tablica – usuwanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elem. | 2500 | 5000 | 7500 | 10000 | 12500 | 15000 | 17500 |
| początek [ms] | 0,016807 | 0,026875 | 0,033688 | 0,044572 | 0,064629 | 0,088659 | 0,104577 |
| losowo [ms] | 0,014642 | 0,022105 | 0,030306 | 0,039157 | 0,051834 | 0,063202 | 0,076080 |
| koniec [ms] | 0,015067 | 0,022452 | 0,030066 | 0,038453 | 0,050256 | 0,059141 | 0,069853 |

Tabela 2. Uśrednione czasy usuwania elementu z tablicy wraz z relokacją miejsca w pamięci

4. Wykres zależności czasu usuwania elementu z początku tablicy wraz z relokacją miejsca w pamięci

5. Wykres zależności czasu usuwania elementu z losowego miejsca tablicy wraz z relokacją miejsca w pamięci

6. Wykres zależności czasu usuwania elementu z końca od rozmiaru tablicy z relokacją pamięci

Analizując otrzymane wyniki, odnajduję analogię do dodawania elementu do tablicy. Czasy w obu przypadkach są praktycznie takie same, jedynie usuwanie z początku wykazuje nieco dłuższy czas. Przyczyną takiego stanu rzeczy jest najprawdopodobniej mało zoptymalizowany pod kątem wydajności algorytm, który różni się od tego który dodaje element w środek lub na koniec tablicy.

1. Tablica – wyszukiwanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elem. | 2500 | 5000 | 7500 | 10000 | 12500 | 15000 | 17500 |
| wyszukiwanie [ms] | 0,005821 | 0,005948 | 0,006246 | 0,006075 | 0,006070 | 0,006349 | 0,006437 |

Tabela 3. Zależność czasu wyszukiwania elementu od rozmiaru tablicy

Tablica zapewnia błyskawiczny dostęp do danych, złożoność obliczeniowa dostępu do nich wynosi O(1). Rozbieżności czasowe w zależności od wielkości struktury powinny być większe (są one spowodowane niewielkim zakresem liczb w niej przechowywanych, a co za tym idzie – wyszukiwanie zajmuje mniej czasu). Jednakże zanotowane wyniki wystarczą do tego by móc jednoznacznie stwierdzić iż ilość czasu potrzebna na wyszukiwanie danych w tablicy jest liniowo zależna od ilości elementów w niej przechowywanych, co można wyrazić złożonością O(n).

1. Lista dwukierunkowa – dodawanie

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elementów | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 |
| początek [ms] | 0,006671 | 0,006696 | 0,006999 | 0,006984 | 0,007160 | 0,007287 | 0,007336 |
| losowo [ms] | 0,034942 | 0,080454 | 0,108018 | 0,125946 | 0,156856 | 0,166210 | 0,184358 |
| koniec [ms] | 0,005992 | 0,006070 | 0,006163 | 0,006456 | 0,006265 | 0,006265 | 0,006520 |

Tabela 4. Zależność czasu dodawania elementu od ilości elementów listy

7. Wykres zależności czasu dodawania elementu na początek w zależności od rozmiaru listy

8. Wykres zależności czasu dodawania elementu w losowe miejsce w zależności od rozmiaru listy

9. Wykres zależności czasu dodawania elementu na koniec w zależności od rozmiaru listy

W przypadku listy dwukierunkowej, na podstawie otrzymanych wyników możemy zauważyć poważną rozbieżność zapotrzebowania czasu w przypadku dodania elementu na początek/koniec i na środek.  
Pomimo że dodanie elementu na początek i koniec listy dwukierunkowej to bardzo szybka i prosta operacja o złożoności jedynie O(1), to dodanie elementu w środek listy jest znacznie bardziej złożonym i czasochłonnym zadaniem. Jest to spowodowane w głównej mierze tym że lista, w odróżnieniu od tablicy, nie cechuje się błyskawicznym dostępem do danych. Aby dostać się pod zadany indeks, należy zacząć od początku bądź końca listy i iterując element po elemencie odnaleźć poszukiwany indeks - ma to złożoność obliczeniową O(n). Dopiero wtedy algorytm przystępuje już do właściwego dodania wartości do struktury (złożoność O(1)), co w rezultacie daje sumaryczną złożoność operacji dodania elementu w środek listy O(n).

1. Lista dwukierunkowa – usuwanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 |
| początek [ms] | 0,006070 | 0,005855 | 0,006046 | 0,006177 | 0,006055 | 0,006217 | 0,006285 |
| losowo [ms] | 0,034284 | 0,065059 | 0,113013 | 0,124620 | 0,119581 | 0,099866 | 0,114572 |
| koniec [ms] | 0,005728 | 0,005699 | 0,005762 | 0,006041 | 0,005782 | 0,005747 | 0,005923 |

Tabela 5. Zależność czasu usuwania elementu od wielkości listy dwukierunkowej

10. Wykres zależności czasu usuwania z początku w zależności od rozmiaru listy

11. Wykres zależności czasu usuwania elementu z losowego miejsca od ilości danych jakie zawiera lista

12. Wykres zależności czasu usuwania elementu z końca od wielkości listy

W przypadku usuwania łatwo można dostrzec analogię do dodawania. Usuwanie element z końca lub początku listy realizowane jest natychmiastowo i niezależnie od wielkości struktury. Natomiast sytuacja komplikuje się jeśli chcemy operować na danych wewnątrz tej listy – najpierw musi nastąpić iteracja do zadanego indeksu by następnie zainicjować proces usuwania danego elementu.

1. Lista dwukierunkowa – wyszukiwanie pierwszego elementu o zadanej wartości

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elementów | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 |
| wyszukiwanie [ms] | 0,006143 | 0,006309 | 0,006090 | 0,006803 | 0,006686 | 0,006402 | 0,006432 |

Tabela 6. Zależność czasu wyszukiwania elementu od rozmiaru listy dwukierunkowej

13. Wykres zależności czasu wyszukiwania od rozmiaru listy

W przypadku wyszukiwania w liście dwukierunkowej, wniosek jest prosty – czas jest liniowo zależny od wielkości struktury w której poszukujemy elementu. Wykres nie przedstawia tego w pełni jednoznacznie – jest to spowodowane małym zróżnicowaniem wartości (zakres od -100 do 100), w rezultacie nie ma potrzeby przeszukiwania głęboko struktury. W tym przypadku granica czasu wykonywania operacji dąży do skończonej wartości bez względu na rozmiar struktury. Biorąc pod uwagę szerszy zakres wartości, wyniki jakie otrzymamy są już w pełni jednoznaczne.

1. Kopiec binarny – dodawanie elementu

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 |
| dodawanie [ms] | 0,086172 | 0,165604 | 0,244690 | 0,323003 | 0,401907 | 0,479722 | 0,559756 |

Tabela . Zależność czasu dodawania elementu wraz z obowiązkową naprawą od rozmiaru kopca

14. Wykres zależności czasu dodawania elementu wraz z obowiązkową naprawą od rozmiaru kopca

Już na pierwszy rzut oka widać, że wstawianie elementu do kopca binarnego nie jest efektywnym elementem, przynajmniej stworzonej przeze mnie jego implementacji. Z pierwotnych założeń wynika, że linia trendu wykresu czasu dodawania elementu w zależności od ilości przechowywanych danych powinna być funkcją rosnącą logarytmicznie. W obecnym jednak przypadku, zależność ta rośnie idealnie liniowo.  
Błąd spowodowany jest tym, że proces naprawy jaki odbywa się po każdym dodaniu elementu jest wykonywany na każdym istniejącym w zbiorze elemencie. Taka operacja ma złożoność O(n), podczas gdy poprawna powinna mieć O(log n). W świetle samego dodania elementu (bez zachowania warunku integralności kopca) i jego złożoności O(1) – w znaczącym stopniu determinuje to ogólną złożoność obliczeniową dodawania elementu do kopca binarnego.

1. Kopiec binarny – usuwanie elementu wraz z naprawą jego struktury

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| czas\ilość elementów | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 |
| usuwanie [ms] | 0,096860 | 0,178290 | 0,254723 | 0,335539 | 0,412122 | 0,494877 | 0,568421 |

Tabela . Zależność czasu usuwania elementu od rozmiaru kopca binarnego

15. Wykres zależności czasu usuwania elementu w zależności od rozmiaru kopca binarnego

W przypadku usuwania elementu z kopca, dostrzec można analogię do dodawania. Ten sam błąd – niewłaściwie zaimplementowana funkcja naprawy kopca obniżająca w znaczącym stopniu wydajność operacji.

1. Kopiec binarny – wyszukiwanie pierwszego elementu o zadanej wartości

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 |
| wyszukiwanie [ms] | 0,016773 | 0,017902 | 0,016602 | 0,016680 | 0,017902 | 0,017604 | 0,020209 |

Tabela . Zależność czasu wyszukiwania elementu od rozmiaru kopca binarnego

16. Wykres zależności czasu wyszukiwania od rozmiaru kopca binarnego

Wydajność wyszukiwania w kopcu ze względu na brak zapotrzebowania na użycie funkcji *naprawKopiec()* nie jest już tak nadszarpnięta. Problemem jednak są spaczone wyniki ze względu na zbyt mały zakres przechowywanych liczb. W przypadku kiedy ich zakres będzie większy, logarytmiczny rozkład wartości czasu będzie odpowiednio bardziej widoczny.

# Wnioski

Na podstawie uzyskanych wykresów nasuwają się jednoznaczne wnioski:

-Złożoność obliczeniowa zaimplementowanej przeze mnie tablicy w przypadku każdej z operacji wynosi O(n), a więc prezentuje się ona zgodnie z pierwotnymi założeniami. Jest to struktura danych o najbardziej powszechnym zastosowaniu.

-Złożoność obliczeniowa listy dwukierunkowej w przypadku dodawania i usuwania elementów na jej początek lub koniec wynosi O(1). Nietrudno zauważyć, że jest zdecydowanie szybsza niż tablica, natomiast nie zapewnia równie szybkiego dostępu do danych jak tablica.

-Kopiec binarny to idealna struktura jeśli naszym głównym kryterium wyboru jest szybkość wyszukiwania elementów. Zapotrzebowanie czasowe w przypadku wyszukiwania wraz z rozmiarem kolekcji rośnie logarytmicznie. Z uwagi na wyróżniające się na tle pozostałych struktur ułożenie danych (są one posortowane od wartości największej do najmniejszej) zapotrzebowanie czasowe na wyszukanie danego elementu zależy także od jego wartości – im większej wartości jest dany element, tym znalezienie jego zajmuje mniejszą ilość czasu. Niestety, sytuacja ma się znacznie gorzej w przypadku modyfikacji danych jakie zawiera kopiec. Mimo że złożoność algorytmu dodawania i usuwania jest identyczna jak w przypadku tablicy i listy, to na wykonanie identycznych operacji trzeba poczekać około ogromną krotność dłużej niż w przypadku listy dwukierunkowej. Winę za taką postać rzeczy ponosi najprawdopodobniej wadliwa implementacja, która po każdym pojedynczym dodaniu/usunięciu elementu przeprowadza naprawę całego kopca, analizując poprawność pozycji każdego z elementów, zamiast zająć się jedynie „wyniesieniem” świeżo dodanego elementu na odpowiednią pozycję. Miałoby to pesymistyczną złożoność O(log n), jednakże w obecnym przypadku wynosi ona O(n).

-Listy są dobrym kontenerem do przechowywania dużej ilości danych, w szczególności jeśli zachodzi częsta potrzeba ich modyfikacji. Radzą sobie one bardzo szybko z ich zapisem jak i odczytem, a ich najważniejszą cechą jest to, że nie wymagają relokacji całej kolekcji w pamięci w przypadku dodawania bądź usuwania elementów.

# Bibliografia

[1] Robert L. Kruse – „Data Structures and Program Design in C++”

[2] Piotr Wróblewski – „Struktury Danych i Techniki Programowania – Algorytmy”

[3] Thomas H. Cormen – „Wprowadzenie do Algorytmów”

[4] Wikipedia (wersja angielska) - <http://en.wikipedia.org/wiki/Binary_heap>

[5] Big-O Cheat Sheet (wersja angielska) - <http://bigocheatsheet.com/>