|  |  |  |  |
| --- | --- | --- | --- |
|  | **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:** 12.05.2015 | |
| **Grupa:** wtorek 11:15 | |
| **Prowadzący:**  Dr inż. Jarosław Mierzwa | **Ocena:** |

# Cel projektu

Celem projektu jest dokładna analiza wydajności algorytmów wyznaczania minimalnego drzewa rozpinającego (algorytm Prima, algorytm Kruskala), oraz algorytmów wyznaczania najkrótszej ścieżki w grafie (algorytm Dijkstry, algorytm Forda-Bellmana). Ich analiza rozpatrywana będzie pod kątem czasu wykonywania w zależności od rozmiaru oraz gęstości grafu.

# Wstęp teoretyczny

Minimalnym drzewem rozpinającym (w skrócie ang. MST) nazywa się drzewo rozpinające danego grafu o najmniejszej z możliwych wag, tj. takie, że nie istnieje dla tego grafu inne drzewo rozpinające o mniejszej sumie wag krawędzi. Sposobów jego wyznaczania jest kilka, licz najważniejszym z nich jest algorytm Prima.

Algorytm Prima jest swojego rodzaju algorytmem zachłannym. Dysponując grafem spójnym i nieskierowanym, algorytm oblicza podzbiór zbioru krawędzi, dla którego graf nadal pozostaje spójny, jednak suma wag wszystkich takich krawędzi w podzbiorze jest najmniejsza z możliwych.

# 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

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 |

miejsca w pamięci

# 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/>