STRUKTURY DANYCH I ZŁOŻONOŚĆ OBLICZENIOWA

Zadanie projektowe nr 1: Badanie efektywności operacji na danych w podstawowych strukturach danych

Autor: Bartosz Rodziewicz, 226105

Prowadzący: dr inż. Dariusz Banasiak

Grupa: Wtorek, TN, 7:30

# Wstęp teoretyczny

## Złożoność obliczeniowa

Miara ilości zasobów potrzebnych do rozwiązania problemów obliczeniowych. Rozważanymi zasobami są takie wielkości jak czas, pamięć lub liczba procesorów.

## Tablica

Rodzaj struktury danych w której wszystkie elementy alokowane są w jednym spójnym bloku pamięci.

### Dodawanie

Dodawanie do tablicy polega na zalokowaniu nowej o element większej tablicy, przepisaniu elementów o indeksie mniejszym niż na który dodajemy nową wartość (jeśli istnieją), wpisaniu naszej wartości i przepisaniu dalszych elementów, aż do końca tablicy (jeśli istnieją).

Złożoność czasowa (wynikająca z konieczności przepisania wszystkich elementów) wynosi **O(n)**.

### Usuwanie

Usuwanie elementu z tablicy polega na zalokowaniu nowej, o element mniejszej tablicy i przepisaniu wszystkich elementów z wyjątkiem usuwanego.

Złożoność czasowa (wynikająca z konieczności przepisania wszystkich elementów) wynosi **O(n)**.

## Lista dwukierunkowa

Rodzaj struktury, w której każdy element jest osobno alokowany w pamięci. Element poza swoją wartością zawiera adres położenia poprzedniego i następnego elementu. W liście dwukierunkowej mamy bezpośredni dostęp do pierwszego i ostatniego elementu.

### Dodawanie i usuwanie na początku, bądź końcu

Jako że mamy bezpośredni dostęp do pierwszego i ostatniego elementu, dodanie lub usunięcie któregoś z nich kompletnie nie zależy od pozostałych, więc złożoność czasowa wynosi **O(1)**.

Samo dodawanie pierwszego lub ostatniego elementu polega na podmienieniu zapamiętanej wartości pierwszego lub ostatniego elementu na ten nowy, a obecnego zapisanie jako adres następnego/poprzedniego.

Usunięcie działa podobnie, adres drugiego bądź przedostatniego zapisujemy jako pierwszy/ostatni i kasujemy informacje o poprzednim/kolejnym elemencie z niego.

### Dodawanie i usuwanie w dowolnym miejscu

W przypadku dodawania i usuwania elementów na dowolnym miejscu poza samym dodaniem/usunięciem (stworzeniem/usunięciem elementu i odpowiednim ustawieniem wskaźników następny/poprzedni) musimy jeszcze doskoczyć do odpowiedniego miejsca w liście. Tutaj, inaczej niż w przypadku tablicy, musimy zrobić to manualnie, przechodząc po kolejnych elementach.

Dla listy dwukierunkowej mamy to ułatwienia/przyśpieszenie, że do elementów znajdujących się w drugiej połowie listy nie musimy przechodzić od początku, a możemy od końca, co daje nam mniej elementów do „przejścia”.

Z tego powodu złożoność czasowa tej operacji wynosi **O(n/2) = O(n)**.

## Kopiec

Jest to rodzaj struktury danych, oparty na drzewie, które posiada jeden warunek – w każdym węźle wartość rodzica, jest niemniejsza od wartości obu synów. Struktura ta opiera się na drzewie kompletnym, co powoduje, że najwygodniejszą metodą jej implementacji, jest implementacja na tablicy.

### Dodawanie

Dodawanie elementu polega na dopisaniu tego elementu, do pierwszego wolnego liścia od lewej w ostatnim niepełnym rzędzie. Po tym dopisaniu, trzeba wykonać sprawdzenie w górę, czy jest zachowana zasada kopca.

Złożoność obliczeniowa sprawdzenia warunku kopca, dla nowo powstałego elementu, jest zależna od ilości poziomów drzewa, czyli wynosi O(log(n)).

Jednak z powodu wykonanej przeze mnie implementacji kopca na tablicy, złożoność ta wzrasta do **O(n)** z powodu czasu koniecznego na realokowanie tej tablicy.

### Usuwanie

Usuwanie elementu z kopca polega na usunięciu wartości w korzeniu (innych usuwań nie implementowałem, ponieważ działały by tablicowo), wpisania ostatniego elementu do korzenia i przywróceniu wartości kopca w dół. Złożoność wychodzi O(log(n)).

Jednak z powodu wykonanej przeze mnie implementacji kopca na tablicy, złożoność ta wzrasta do **O(n)** z powodu czasu koniecznego na realokowanie tej tablicy.

# Implementacja programowa struktur

## Klasa Programu

Klasa ta odpowiada za wyświetlanie menu i podejmowanie w nim działań.

## Klasa Struktury

Jest to klasa praktycznie czysto wirtualna, po której dziedziczą wszystkie kolejne struktury danych. Wprowadziłem ją, aby znacznie uprościć klasę programu (wystarcza stworzenie jednego menu dla wszystkich struktur), jak i zwiększyć czytelność i porządek w kodzie.

Zawiera deklarację wirtualnych metod które powinny być zawarte każdej dziedziczącej strukturze.

## Klasa Tablicy

Klasa realizująca strukturę dynamicznej tablicy. Poza implementacją każdej wirtualnej metody z klasy Struktury zawiera jedynie jedno prywatne pole wskaźnika na początek dynamicznej tablicy.

## Klasa Listy

Klasa realizująca listę dwukierunkową. Posiada w sobie pomocniczą klasę pojedynczego elementu, z którego lista ta jest zbudowana.

Dodatkowo posiada wymagane prywatne wskaźniki na pierwszy, jak i ostatni element oraz kilka prywatnych metod pomagających wykonać dodawanie i usuwanie elementów.

Już po skończeniu tej klasy zauważyłem, że implementacja metody dodającej, jak i usuwającej powinna zostać podzielona na więcej prywatnych podmetod, aby zachować większy porządek w kodzie, jednak nie chciałem już przeformatowywać tej klasy i zostało tak jak jest.

Dodatkowo pod koniec zauważyłem, że metody:

* dodająca z pliku (gdzie pomocniczo wykorzystuje std::vector),
* generowania losowych wartości

są napisane w ten sam sposób w każdej klasie i można by je przenieść jako uniwersalne dla każdej struktury.

## Klasa Kopca

Jako jedyna z klas struktur nie dziedziczy bezpośrednio z klasy Struktury, a z klasy Tablicy, ponieważ moja realizacja kopca jest wykonana na tablicy.

Poza niektórymi przedefiniowanymi metodami z tablicy posiada metody odpowiedzialne za przywracanie zasady kopca, jak i uzyskiwania indeksu rodzica, bądź potomków.

## Klasa licznika

Ostatnia główna klasa w programie. Jest ona reprezentacją licznika QueryPerformanceCounter wykorzystywanego przeze mnie do pomiaru czasu trwania poszczególnych algorytmów.

Praktycznie w całości zawiera kod ze strony podanej w treści zadania dotyczącej tego licznika.

# Metoda testowania i plan eksperymentu

Testować moje algorytmy zamierzam za pomocą licznika z klasy licznik (QueryPerformanceCounter).

Testy będą wykonywane na czterech rozmiarach struktur:

* Zawierającej 50 elementów,
* Zawierającej 1 000 elementów,
* Zawierającej 10 000 elementów,
* Oraz zawierającej 20 000 elementów.

Dla każdego rozmiaru będę wykorzystywał trzy rodzaje zakresów danych:

* [0, 100],
* [0, 16383],
* [0, 32767]

Daje mi to 12 przypadków do testów.

Test polega na stworzeniu struktury z konkretnego przypadku za pomocą metody generowania, uruchomienie licznika, wykonanie testowanego algorytmu (np. dodanie do tablicy wartości z zakresu konkretnego przypadku), zapisanie wyniku, cofnięcie operacji, i kilkukrotne powtórzenie jej.

Powtarzać zamierzam 90 razy i każdy wynik zapisuję do pliku.

Później z tych danych wyliczam średnią dla każdej operacji i tworzę wykresy, które załączam w dalszej części dokumentu. Do wyliczenia średniej pozbyłem się wartości, które bezsprzecznie były błędne (5 największych i 5 najmniejszych).

**Wyniki pomiarów podane są w mikrosekundach.**

Na sam koniec dokonuję porównania poszczególnych struktur w przełożenie na konkretne operacje.

# Wyniki testów

## Tablica

### Dodawanie

Wyniki testów dodawania do tablicy niezależnie od pozycji, jak i zakresu przechowywanych danych nie różnią się od siebie praktycznie wcale. Układają się w praktycznie prostą linię i potwierdzają złożoność **O(n)**.

### Usuwanie

W przypadku operacji usuwania mamy podobną sytuację jak przy dodawaniu – wyniki są praktycznie takie same, ponieważ cały potrzebny czas jest zużywany na przepisanie tablicy. W tym przypadku również wyniki układają się w proste linie potwierdzając złożoność **O(n)**.

## Lista

### Dodawanie

Dodawanie elementów do listy na początku, bądź końcu posiada złożoność **O(1)**, co generalnie wyniki moich testów potwierdzają – „duże” wahania wykresu (jak na przyjętą skalę) wynikają z dokładności licznika, który zwracał wartość 0, bądź dopiero 0.31 – taka dokładność wynika z budowy tego licznika (opiera się on na taktach procesora) i taktowania mojej konkretnej jednostki.

Dodawanie do środka posiada złożoność **O(n)**, co potwierdzają moje testy.

### Usuwanie

Tak jak pisałem przy dodawaniu dokładność licznika wynosiła 0.31, co przy wynikach w okolicy 0-0.5 nie jest w stanie dać miarodajnych wyników, uważam, że wykresy potwierdzają złożoność **O(1)**.

Wykres dla dodawania tworzy całkowicie prostą linię i potwierdza złożoność **O(n)**.

## Kopiec

Dodawanie do kopca posiada złożoność O(log n), jednak z powodu implementacji tablicowej u mnie złożoność ta wzrasta do **O(n)**, z powodu czasu wymaganego na przepisanie elementów. Taką złożoność potwierdzają moje testy.

Identycznie, jak w przypadku dodawania, konieczność przealokowania tablicy powoduje zwiększenie złożoności do **O(n)**, co potwierdza powyższy wykres.

# Porównanie struktur

## Dodawanie na początku

## Dodawanie w środku

## Dodawanie na końcu

## Usuwanie na początku

## Usuwanie w środku

## Usuwanie na końcu

# Wnioski

Implementacja timera do mierzenia tak małych odstępów czasu powinna zostać wykonana lepiej.

Najwygodniejszą i najbardziej uniwersalną strukturą wg. mnie jest lista. Patrząc na wyniki, zwycięża we wszystkich testach (wiem, jednak, że w przypadku wyszukiwania zwycięstwo nie byłoby już oczywiste).

Za najprostszą strukturę nadal jednak uważam tablicę i uważam, że do małych ilości elementów jest wystarczająca.