Politechnika Wrocławska

Wydział Informatyki
i Telekomunikacji

STRUKTURY DANYCH

Projekt nr 1

Wykonali:
Filip SANOWSKI - 280062
Kacper KRUSZELNICKI - 280150

Prowadzący:
mgr. inż Piotr NOWAK

9 kwietnia 2025

Spis treści

link do repozytorium GitHub z pełną implementacją

1 Wstęp

1.1 Tablica dynamiczna

Tablica dynamiczna to struktura danych, która choc ma określony romiar w trakcie tworzenia to może automatycznie zmieniać swój rozmiar gdy zajdzie taka potrzeba oraz przechowuje elementy sekwencyjnie w pamięci. Działa ona w następujący sposób: na poczatku zostaje tworzona tablica o podanym rozmiarze, czyli alokuje blok pamięci o podanym rozmiarze, następnie monitoruje na podsatwie ilości faktycznie przechowywanych elementów (*size*) oraz całkowitej pojemności (*capacity*), w momencie w kórym liczba elementów zbliża się do wartości granicznej, alokowany jest nowy większy blok pamięci, a elementy ze satrej tabliicy sa kopiowane do nowej. Operacje takie jak *get*, i *popBack* mają złożonośc obliczeniową równą *O(1)* ponieważ dostęp do lementu jest realizowany orzez indeks w czasie stałym a usuwanie z końca wymaga tylko zmniejszenia licznika. Natomiat operacje takie jak *pushBack*, *pushFront*, *pushAt*, *popFront*, *popAt*, *search* oraz *resize* posiadają złożonośc obliczeniową równą *O(n)*. Dzieje się tak ponieważ wstawianie na koniec przy pojedyńczych operacjach może wynieśc złożoność *O(n)* przy realokacji (gdy *size* == *capacity*), a reszta operacji przez przesuwanie elementów. Wszystkie przypadki zostały rozpatrzone w sposób pesymistyczny. Tablice dynamiczne swobodny dostęp w czasie *O(1)*, łatwą implementację, dobrą lokalność przestrzenną oraz fakt, że elementy sa przechowywane obok siebie, lecz operacje na tablicach dynamicznych są kosztowne oraz zosatwiają niewykorzystaną pamięć.

1.2 Lista wiązana

Lista wiązana jednokierunkowa to stuktura danych, która jest bardzo często wykorzystywana w informatyce. Składa się ona z połączonych ze sobą *węzłów* (*node*), gdzie każdy z tych węzłów zawiera *dane* (*data*) oraz wskaźniki do innych węzłów (*next*). Węzły są przechowywane gdziekolwiek jest wolna przestrzeń w pamięci, ponieważ nie musza być przechowywane w sposób ciągły (w odróznieniu do tablic) i to daje możliwość dynamicznego zarządzania pamięcią. W omawianej poniżej impelmentacji, została wykorzystana lista wiązana wykorzystująca wskaźniki *head* oraz *end* (w literaturze szeroko znana jako tail). *Head* wskazuje na pierwszy węzeł listy, a *end* wskazuje na ostatni węzeł listy. Dzięki wskaźnikowi head można w czasie stałym *O(1)* dodawać i usuwać elementy z początku listy. Wskaźnik *end* pozwala na dodawanie na koniec listy w czasie stałym *O(1)*, bez konieczności iterowania przez całą listę. Brak wskaźnika *end* wymuszałby przy operacjach dodawania na koniec przechodzenie przez całą listę co miałoby złożoność *O(n)*.Usuwanie z końca listy niestety ma złożoność *O(n)*, ponieważ mimo wskaźnika *end* i tak musi znaleźć przedostatni element. Taka sam złożonośc obliczeniowa tyczy się również operacji wyszukiwania elementu, dodawania i usuwania ze środka.

2 Implementacja

2.1 Tablica dynamiczna

Implementacja tablicy dynamicznej polegała na tym, że utworzono klasę *DynaicArray*, w której utowrzono wskaźnik na tablicę (*arr*) przechowujące dane w ciągłym bloku pamięci, liczbę przechowywanych elementów(*size*) oraz całkowitą dostępną przestrzeń w tablicy (*capacity*). Zarządzanie pamięcią odbywało się poprzez alokowanie początkowego bloku pamięci w konstruktorze (*new int[capacity]*), zwalnianie całej pamięci w destruktorze (*delete[] arr*) oraz automatycznym zwiększaniu pojemności metodą (*resize()*-podeajanie romziaru). Operacje dodawania lementów zostały zawarte w metodach: *pushBack*, *pushFront*, *pushAt*. Operacje usuwania elementów zostały zawarte w metodach: *popback*, *popFront*, *popAt*. Wyszukiwanie elemntów zostało wykonane za pomoca metody *search()*, a dostęp do elemntów za pomocą *get()*. Zaimplementowano również obsługę wyjatków takich jak: sprawdzanie poprawności indeksu, automatyczne powiększanie tablicy przy braku miejsca oraz wyrzucania wyjątków przy nieprawidłowym dostępie.

```
class DynamicArray {
private:
    int* arr;
    int size;
    int capacity;
    void resize();

public:
    DynamicArray(int set_capacity);
    ~DynamicArray();

    void pushBack(int val);
    void pushFront(int val);
    void pushAt(int index, int val);



    void popBack();
    void popFront();
    void popAt(int index);

    int search(int val);
    int get(int index);

    void print() const;
};
```

Rysunek 1: Struktura tablicy dynamicznej

2.2 Lista wiązana

Implementacja listy wiązanej jednokierunkowej polegała na utworzeniu na sam początek struktury węzła (*node* zawierającą elementy takie jak: wartośc liczbową (*data*) oraz wsakźnik na następny węzeł (*next*). Naastępnie w klasie *LinkedList* mamy wskaźnik na początek listy (*head*), wskaźnik na koniec listy (*end*) oraz licznik elementów (*counter*). Oprócz tego mamy metody takie dodawanie, usuwanie elementów z początku, końca i wybranej pozycji listy, wyszukiwanie danego elementu, wyświetlanie zawartości listy oraz sprawdzenie czy lista jest pusta, czyszczenie listy przez destruktor oraz sprawdzenie rozmiaru listy. Zarządzanie pamięcią jest obsługiwane za pomoca: dynamicznej alokacji węzła przy dodawaniu (*new Node*), zwalniania pamięci przy usuwaniu (*delete*). W kodzie również użyto obsługi wyjątków przy przypadkac gdy lista jest pusta, lista ma jeden element lub podano nieprawidłowe pozycje.

```cpp
struct Node {
    int data;
    Node* next;

    Node(int val);
    void showValue() const;
};

class LinkedList {
private:
    Node* head;
    Node* end;
    int counter;

public:
    LinkedList();
    ~LinkedList();

    bool isEmpty() const;
    int size() const;

    void addToStart(int val);
    void addToEnd(int val);
    void addToPosition(int val, int pos);

    void deleteFromStart();
    void deleteFromEnd();
    void deleteFromPos(int pos);

    int find(int findVal);
    void show() const;
};
```

Rysunek 2: Struktura listy jednokierunkowej wiązanej

3 Badania

Badania wydajności operacji na tablicy dynamicznej oraz liście wiązanej zostały przeprowadzone na komputerze o poniższej specyfikacji sprzętowej i programowej:

- **Procesor:** Intel(R) Core(TM) i5-14600K
- **Liczba rdzeni fizycznych:** 14
- **Liczba wątków logicznych:** 20
- **Pamięć operacyjna RAM:** 16 GB
- **System operacyjny:** Microsoft Windows 10 Home 64-bit
- **Wersja kompilatora:** Microsoft Visual C++ (MSVC)

3.1 Implementacja

W celu zbadania efektywności czasowej poszczególnych operacji na tablicy dynamicznej oraz liście wiązanej, zaimplementowaliśmy dedykowany program testowy w języku C++. Testy zostały przeprowa-

dzone w oparciu o bibliotekę `<chrono>`, która umożliwia precyzyjny pomiar czasu wykonania operacji w nanosekundach.

Dla każdej operacji przygotowaliśmy pętlę powtarzającą daną operację **50 razy** (`il = 50`) na wcześniej przygotowanych strukturach zawierających **10 000 elementów** (`rozmiar = 10000`). Dzięki temu możliwe było uzyskanie uśrednionego czasu wykonania, co zwiększa wiarygodność wyników. Dodatkowo prrzeprowadziliśmy tą procedurę 10 razy aby jeszcze bardziej uwiarygodnić nasze wyniki

Każdy pomiar był realizowany w następujący sposób:

1. Przed pomiarem wykonywano przygotowanie struktury (np. `pushBack(i)`, `addToEnd(i)`) w celu uzyskania pełnej listy/tablicy o ustalonym rozmiarze.
2. Właściwa operacja (np. `arr.popBack()` lub `list.deleteFromEnd()`) była wykonywana w zakresie jednej iteracji, której czas mierzono za pomocą `high_resolution_clock`.
3. Po każdej iteracji struktura była przywracana do stanu wyjściowego (np. przez usunięcie dodanego elementu), tak aby warunki każdej próby były takie same.
4. Czasy z każdej iteracji były sumowane w zmiennej `licznik` typu `long long`, a następnie dzielone przez liczbę iteracji w celu wyliczenia średniej.

Dzięki ujednoliceniu schematu testowania oraz zastosowaniu tych samych danych wejściowych dla obu struktur, uzyskano porównywalne i miarodajne wyniki, które pozwalają na analizę teoretycznej oraz praktycznej złożoności czasowej.

```
licznik = 0;
for (int i = 0; i < il; i++)
{

    start = high_resolution_clock::now();
    arr.pushBack(2137);
    stop = high_resolution_clock::now();
    licznik += duration_cast<nanoseconds>(stop - start).count();
    arr.popBack();

}
cout  << licznik / il << endl;
```

Rysunek 3: Przykładowa implementacja badania wydajności

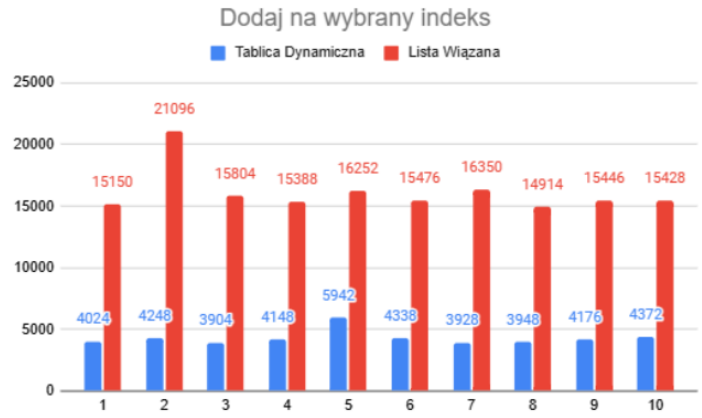
3.2 Dodawanie na początku

Rysunek 4: Wykres przedstawiający porównanie operacji dodawania na początku dla obu struktur.

3.3 Dodawanie na końcu

Rysunek 5: Wykres przedstawiający porównanie operacji dodawania na końcu dla obu struktur

3.4 Dodawanie na pozycji

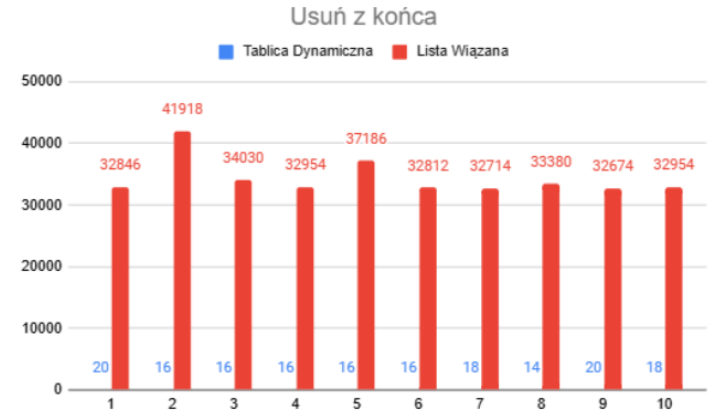


Rysunek 6: Wykres przedstawiający porównanie operacji dodawania na pozycji dla obu struktur

3.5 Usuwanie z początku

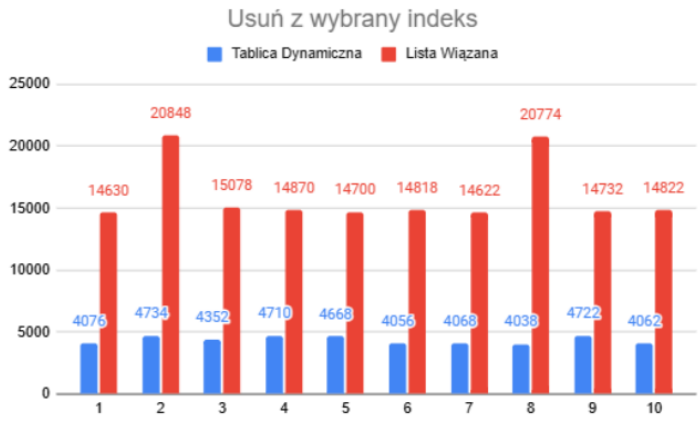
Rysunek 7: Wykres przedstawiający porównanie operacji usuwania na początku dla obu struktur

3.6 Usuwanie z końca



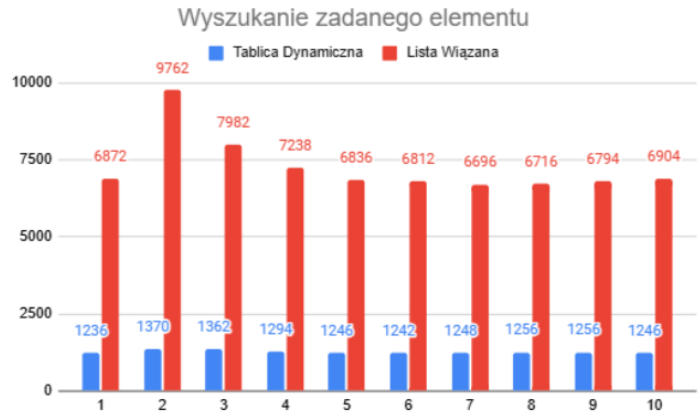
Rysunek 8: Wykres przedstawiający porównanie operacji usuwania na końcu dla obu struktur

3.7 Usuwanie z pozycji



Rysunek 9: Wykres przedstawiający porównanie operacji usuwania na pozycji dla obu strukturi

3.8 Wyszukiwanie zadanego elementu



Rysunek 10: Wykres przedstawiający porównanie operacji wyszukania zadanego elementu dla obu struktur

3.9 Tabele z wynikami

Lista wiązana - czas (nanosekundy)										
nr pomiaru	1	2	3	4	5	6	7	8	9	10
dodaj na koniec	90	102	110	106	98	106	116	104	106	110
dodaj na początek	82	74	106	94	92	94	236	78	82	204
dodaj na wybrany indeks	15150	21096	15804	15388	16252	15476	16350	14914	15446	15428
usuń z końca	32846	41918	34030	32954	37186	32812	32714	33380	32674	32954
usuń z początku	84	72	126	84	90	76	82	86	82	82
usuń z wybrany indeks	14630	20848	15078	14870	14700	14818	14622	20774	14732	14822
szukaj	6872	9762	7982	7238	6836	6812	6696	6716	6794	6904

Rysunek 11: Tabela z wynikami czasu wykonywania danych operacji dla listy jednokierunkowej wiązanej.

Operacje na początku listy są bardzo szybkie, ponieważ wymagają jedynie zmiany referencji. Operacje na danej pozycji są już bardziej czasochłonne, ponieważ trzeba przeiterować przez całą listę. W

Tablica Dynamiczna - czas (nanosekudny)										
nr pomiaru	1	2	3	4	5	6	7	8	9	10
dodaj na koniec	756	726	654	694	612	684	656	696	688	632
dodaj na początek	7042	6544	7902	5922	5908	6956	7000	6980	6840	7066
dodaj na wybrany indeks	4024	4248	3904	4148	5942	4338	3928	3948	4176	4372
usuń z końca	20	16	16	16	16	16	18	14	20	18
usuń z początku	8216	8248	10230	9066	8680	8014	8748	8686	8382	7942
usuń z wybrany indeks	4076	4734	4352	4710	4668	4056	4068	4038	4722	4062
szukaj	1236	1370	1362	1294	1246	1242	1248	1256	1256	1246

Rysunek 12: Tabela z wynikami czasu wykonywania danych operacji dla tablicy dynamicznej.

tablicy dynamicznej operacje na końcu są relatywnie szybkie ze względu na implemnetacje. Operacje zpoczątkiem i konkretną pozycją są wolniejsze ze względu na przesuwanie elementów w tablicy. Przy

Średnia - czas (nanosekundy)		
	Tablica Dynamiczna	Lista wiązana
dodaj na koniec	680	105
dodaj na początek	6816	114
dodaj na wybrany indeks	4303	16130
usuń z końca	17	34347
usuń z początku	8621	86
usuń z wybrany indeks	4349	15989
szukaj	1276	7261

Rysunek 13: Tabela z wynikami średniego czasu wykonywania danych operacji dla obu struktur.

operacjach sekwencyjnych, czyli podczas przeszukiwania elemntów w strukturze tablica dynamiczna wydaje sie dużo bardziej efektywna. Natomaist lista wiązana dużo lepiej się sprawdza przy operacjach wymagajacych zmiany na poczatku i końcu.

4 Wnioski

Tabela dynamiczna wykonuje operacje na końcu tabeli dużo szybciej niż na początku i w środku tabeli ponieważ nie musi przesuwac elemntów. Usuwanie elemntów z końca daje najniższe czasy co zgadza się z teorią. Dodawanie na końcu daje większe czasy niż usuwane ze względu na przypadki w których konieczne jest realokowanie tablicy przy jej przepełnieniu co również jest zgoden z teorią ($O(1)$ w większości przypadków i $O(n)$ przy realokacji).

Lista wiązana ma dobre czasy podczas dodawania elementów na końcu (dzięki zaimplementowaniu *head* i *end*) i na początku oraz usuwania z początku i zgadza się to z teorią ($O(1)$), dzieje się tak ponieważ wymagają tylko zmiany referncji. Usuwanie z końca jest nieoptymalne ze względu na potzrebe przejścia przez całą listę w celu poszukiwania przedostatniego lementu, mimo użycia tutaj *enda* (*taila*). To również zgadza się z teorią. Operacje w środku listy oraz wyszukiwanie danego elementu ma zamierzoną oczekiwany wynik według teorii, ponieważ iteruje po wszystkich elemnetach z listy.

5 Źródła

Literatura

[1] Wikipedia. (2025). *Lista*. Pozyskano z: https://pl.wikipedia.org/wiki/Lista

[2] Wikipedia. (2025). *Tablice*. Dostęp: https://pl.wikipedia.org/wiki/Tablica_(informatyka)