Struktury Danych

Michał Pawlik, Michał Białek

10.04.2024

Implementacja struktur - kod zamieszczony na platformie github

`https://github.com/MichalBialek01/DataStructuresTest.git`

Cel projektu

Struktury danych w języku C++

W języku C++ rozróżniamy wiele struktur danych. Są to tablice, wektory, listy, kolejki, stosy, mapy, zbiory, kopce i drzewa. Badanymi przez nas strukturami są tablice dynamiczne oraz pojedyńczo i pdowójnie wiązane listy.

Obie struktury, w porównaniu do reszty, cechują się indeksowaniem elementów, przechowywaniem ich w kolejności dodawania, oraz przechowalnością duplikatów.

Tablice

Nazwy w innych językach programowania:

1. Java – ArrayList
2. C++ – Tablica dynamiczna
3. JavaScript – Array
4. Python – List

Porównanie tablicy statycznej oraz tablicy dynamicznej

Tablice statyczne i dynamiczne to podstawowe struktury danych używane w programowaniu do przechowywania i zarządzania zbiorami elementów. Zarówno tablice statyczne, jak i dynamiczne mają swoje specyficzne cechy, zalety, i ograniczenia, które wpływają na ich wykorzystanie w różnych przypadkach.

Główną różnicą pomiędzy statyczną tablicą a dynamiczną jest to, że inicjalizując tablicę statyczną, musimy podać jej rozmiar, który podczas całego trwania programu nie ulega zmianie. Dynamiczna tablica, podczas działania programu może zmieniać swój rozmiar, alokując więcej miejsca, lub zwalniając je, co nie jest możliwe w tablicy statycznej. Z tego powodu, tablice dynamiczne są uważane, za bardziej elastyczne, i optymalne pod względem wykorzystania.

Tablice statyczne

Zalety:

- Prostota użycia
- Brak potrzeby dynamicznego zwięszkania wielkości tablicy, poniważ ma stały, zdefiniowany rozmiar
- Mniejsze zużycie pamięci wynikające z braku mechanizmów zarządzania pamięcią

Wady:

- Brak elastyczności: Rozmiar tablicy musi być znany na etapie kompilacji, co może być ograniczeniem w wielu aplikacjach.
- Możliwe marnotrawienie pamięci: Jeśli nie wszystkie zarezerwowane miejsca są wykorzystane, pamięć zostaje zmarnowana.

Tablice dynamiczne

Zalety:

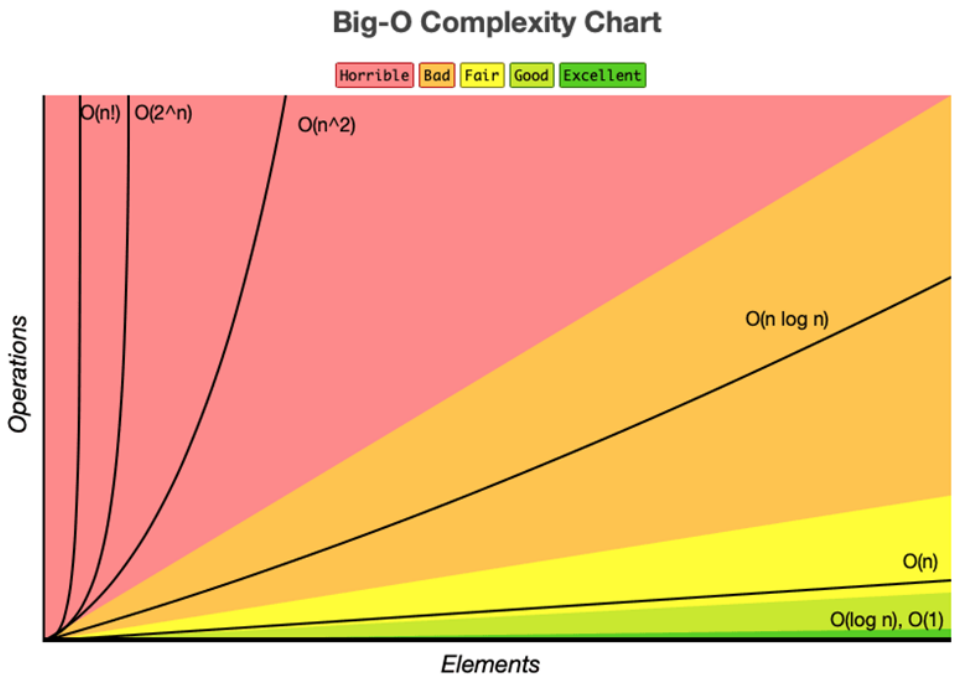
- Elastyczność: Możliwość zmiany rozmiaru w czasie wykonania pozwala na efektywniejsze wykorzystanie pamięci.
- Zazwyczja posiadają zdefinowane metody, które ułatwiają operacje na strukturach

Notacja dużego O – omówienie ogólne

Notacja dużego O służy do analizy złożoności czasowej i pamięciowej w zależności od ilości danych wejściowych.Jest to przyadtne narzędzie, które pozwala zrozumieć jak algorytm, lub struktra danych będzie się skalować wraz ze wzrostem danych wejściowych, co jest przydatne do porównywania poszczególnych struktura danych, i wiedząca jaką szacunkową ilością danych wejścowych będziemy obsługiwać, wybrać najlepiej dopasowaną strukturę.

Wyróżnia się następujące rzędy złożoności:

- $O(1)$ – złożoność stała
- $O(\log n)$ – złożoność logarytmiczna
- $O(n)$ – liniowa
- $O(n^2)$ – kwadratowa. Dla niektórych problemów uznawana za szybką, dla innych za powolną
- $O(n^3)$ – sześcian
- $O(2^n)$ – wykładnicza, zwykle nieakceptowalnie powolna
- $O(n!)$ – silnia
- $O(n^2 + n + C)$ - wielomianowa



Zdjęcie 1: Big-O Complexity Chart

Notacja dużego O – Tablica statyczna, tablica dynamiczna

Tablica statyczna:

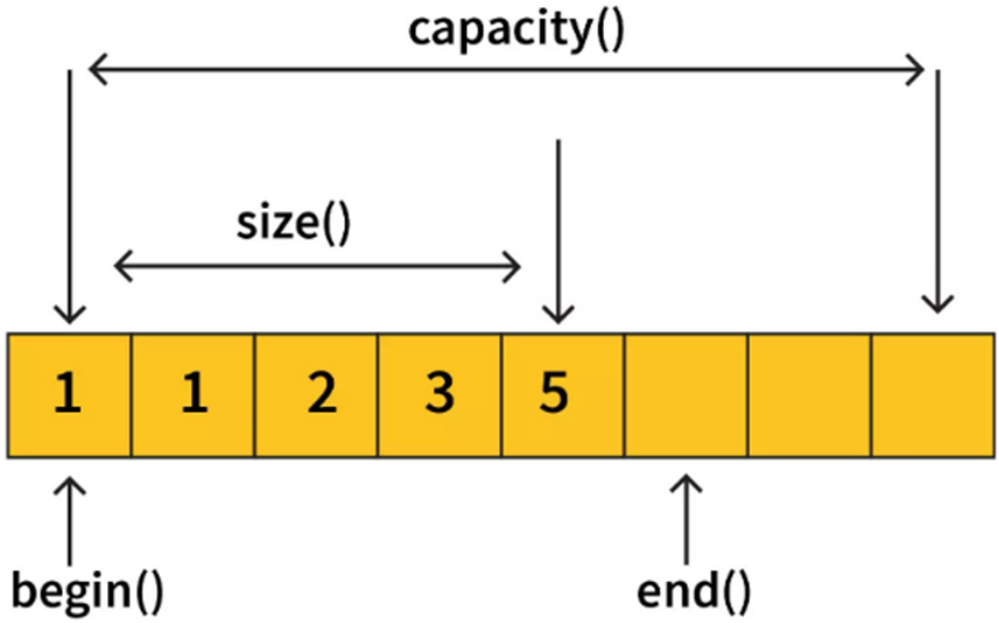
- Dostęp do elementu (np. (tablica[i])) ma złożoność $O(1)$, co oznacza, że czas dostępu do dowolnego elementu jest stały i nie zależy od rozmiaru tablicy. Jeśli jednak dodajemy lub usuwamy elementy tylko na końcu tablicy, to operacje te są niemożliwe bez zmiany jej rozmiaru (co jest niedopuszczalne dla tablic statycznych).
- Wstawianie i usuwanie elementów w środku tablicy ma złożoność $O(n)$, gdzie n jest liczbą elementów w tablicy.

Tablica dynamiczna:

- Dostęp do elementu nadal ma złożoność $O(1)$ w przypadku bezpośredniego dostępu.
- Wstawianie elementu na końcu tablicy ma złożoność $O(1)$ w przypadku średnim, gdy tablica ma wystarczającą ilość zaalokowanego miejsca. Jednakże, w najgorszym przypadku, gdy wymagane jest zwiększenie rozmiaru tablicy (np. przez podwojenie jej pojemności), złożoność operacji może wzrosnąć do $O(n)$, ze względu na potrzebę skopiowania istniejących elementów do nowej, większej tablicy.
- Usuwanie elementu z końca tablicy generalnie ma złożoność $O(1)$. Usuwanie elementów z innych pozycji wymaga przesunięcia pozostałych elementów, co daje złożoność $O(n)$.
- Zmiana rozmiaru tablicy jest operacją wewnętrzną i ma złożoność $O(n)$ w najgorszym przypadku, ponieważ wymaga stworzenia nowej tablicy większego rozmiaru i skopiowania wszystkich elementów.

Zasada działania tablicy dynamicznej, kiedy ją stosować

Tablica dynamiczna jest zalecana w przypadku danych, na których nie będzie często wykorzystywana operacja usuwania, dodawania elementów na początku (najgorszy przypadek) oraz na elementach środkowych (średni przypadek). Takie zalecenie wynika z tego, że podczas usuwania elementów na początku w tablicy dynamicznej wszystkie pozostałe elementy muszą być przesunięte, co wymaga czasu i dodatkowej pracy.



Zdjęcie 2: Tablica dynamiczna

Tablicę dynamiczną możemy osiągnąć na wiele sposobów:

1. Stosując operatory `new[]` oraz `delete[]`, i następnie operując wskaźnikiem na adresy poszczególnych elementów utworzonej tablicy.
2. Bardziej manualne podejście poprzez funkcje `malloc()` i `free()`.
3. Wykorzystanie kontenerów poprzez bibliotekę STL, która posiada wektor, który automatycznie zmienia swoją wielkość w miarę dodawania lub usuwania elementów. Jest bardzo podobny do tablicy dynamicznej, ale oferuje dodatkowe funkcje, takie jak dynamiczne zwiększanie rozmiaru. Jest on jedną z najbezpieczniejszych metod tworzenia oraz operacji na dynamicznych tablicach, ponieważ ma bezpieczne mechanizmy dynamicznej zmiany rozmiaru, wbudowane metody i funkcje do operowania na elementach, funkcje iteracyjne, oraz w łatwy sposób przy pomocy szablonów umożliwia wymuszenie przechowywanie określonych typów.

Podstawowe metody tablicy dynamicznej

Dodawanie elementu:

- add(element): Dodaje nowy element na koniec tablicy.

- insert(index, element): Wstawia nowy element na określonej pozycji.

Usuwanie elementu:

- remove(element): Usuwa pierwsze wystąpienie określonego elementu.
- removeAt(index): Usuwa element na określonej pozycji.
- clear(): Usuwa wszystkie elementy z tablicy.

Pobieranie elementu:

- get(index): Zwraca element na określonej pozycji.
- find(element): Znajduje pierwsze wystąpienie określonego elementu i zwraca jego indeks.

Modyfikacja elementu:

- set(index, element): Ustawia wartość elementu na określonej pozycji.
- replace(oldElement, newElement): Zamienia pierwsze wystąpienie określonego elementu na nowy element.

Informacje o tablicy:

- size(): Zwraca liczbę elementów w tablicy.
- isEmpty(): Sprawdza, czy tablica jest pusta.
- capacity(): Zwraca aktualną pojemność tablicy (ilość elementów, jakie może przechowywać bez realokacji).

Operacje specjalne:

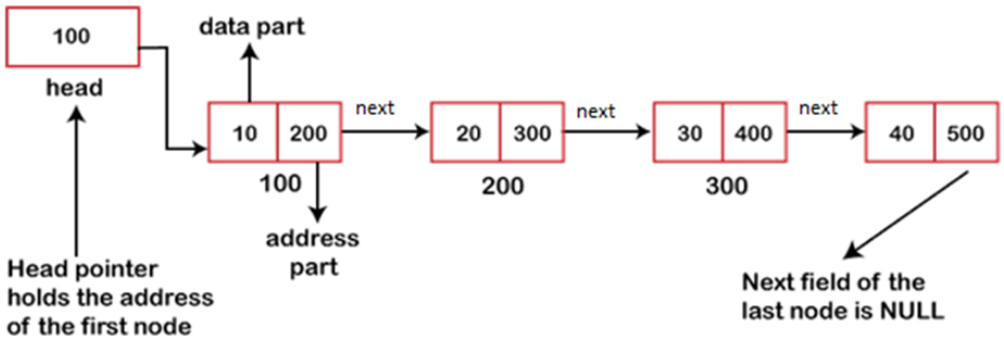
- resize(newSize): Zmienia rozmiar tablicy na nowy rozmiar.
- reserve(newCapacity): Rezerwuje pamięć dla określonej pojemności, ale nie zmienia liczby elementów.
- begin(), `end()`: Zwraca iteratory wskazujące na początek i koniec tablicy.
- `sort()`: Sortuje elementy w tablicy.
- `reverse()`: Odwraca kolejność elementów w tablicy.

Listy – pojedyńczo wiązana lista, podwójnie wiązana lista

Listę wiązaną, ze względu na swoją strukturę możemy podzielić na jednokierunkową listę wiązaną, oraz dwukierunkową listę wiązaną.

Jednokierunkowa lista składa się z wskaźnika head, zawierającego adres pierwszej encji struktury danych.Dana encja oprócz zawartości, posiada część adresową, która posiada wskaźnik na następny element.Każda encja pojedyńczo wiązanej listy posiada namiar na następny element, aż do końca, gdzie ostatnia encja zawiera wskaźnik wskazujący na null. Rozwiązanie te pozwala na dostawanie się z łatwością do elementów "od przodu".

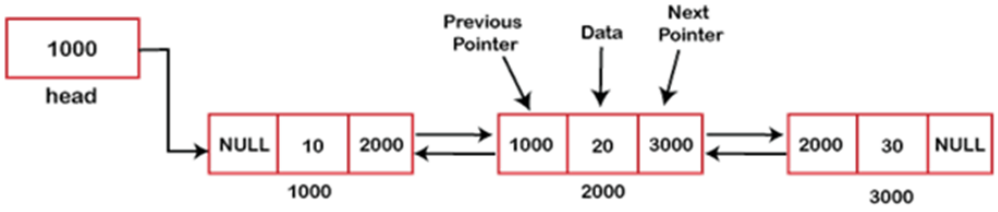
Jednokierunkową liste wiązaną przedstawia poniższa grafika



Zdjęcie 3: Jednokierunkowa lista wiązana

Dwukierunkowa lista wiązana działa na podobnej zasadzie jak pojedyńczo wiązana lista,lecz w tym przypadku każda encja (node) w pamięci, ma namiary na poprzedni i następny element. Jest to rozwiązanie optymalniejsze, jeżeli chcemy operaować na ostatnich elementach struktury, ponieważ podczas usuwania,wyszukiwania,modyfikacji danych na końcu, nie musimy przechodzić przez wszystkie wkaźniki przed elementem docelowym, lecz szybciej się dostać od końca struktury. Kosztem tej optymalizacji jest to, że każda encja musi posiadać dodatkową referencję, co zwiększa zasobność tej implementacji.

Dwukierunkową listę wiązaną przedstawia poniższa grafika



Zdjęcie 4: Dwukierunkowa lista wiązana

Tabela porównania: jednokierunkowa lista wiązana vs dwukierunkowa lista wiązana

Tabela porównania jednokierunkowa lista wiązana vs dwukierunkowa lista wiązana

Porównanie list	**Jednokierunkowa lista wiązana**	**Dwukierunkowa lista wiązana**
Dostępność	Tylko kierunek do przodu struktury	Dostęp od przodu oraz od tyłu struktury
Ilość wskaźników	Każdy węzeł (node) ma wskaźnik wyłącznie na następny element struktury	Każdy węzeł ma wskaźnik na poprzedni, oraz następny element struktury.
Zasobność pamięci	Ze względu na mniejszą ilość wskaźników, wykorzystuje mniejszą ilość pamięci.	Ze względu na większą ilość wskaźników, wykorzystuje większą ilość pamięci
Efektywność	Ze względu na brak dostępności elementów od tyłu struktury jest mniej efektywny	Ze względu dostępności elementów od tyłu struktury jest bardziej efektywny
Możliwe implementacje	Możliwość implementacji na stosie	Możliwość implementacji na stosie, stercie, drzewie binarnm
Złożoność czasowa usuwania i dodawania elementów na danej pozycji	O(n)	O(1)
Złożoność czasowa odczytywania elementów	O(n)	O(n/ 2) ponieważ dostęp od przodu i od tyłu optymalizuje proces odczytu ze względu na mechanizm detekcji od której strony szybciej dojdzie się do elementów.

Zasada działania listy wiązanej, kiedy ją stosować

Współcześnie w większości przypadków domyślną implementacją listy, jest dwukierunkowa lista wiązana. Dzięki niej każdy węzeł (node) na namiar na poprzedni, oraz następny element listy, podczas gdy jednokierunkowa lista wiązana ma jedynie namiar na następny element. Odczyt i operacja na danych na początku oraz końcu struktury jest bardzo szybka, dlatego ta struktura najczęściej zawiera metody, które umożliwiają operacje bezpośrednio na elementach początkowych oraz końcowych. Odczyt oraz operacja na danej zmiennej pośredniej (w najgorszym przypadku środkowym) jest dłuższa, ponieważ aby wykonać operację/odczyt danego węzła pośrodkowego, mechanizm musi odczytywać referencję od pierwszego elementu struktury, po referencjach elementów pośrednich do docelowego, lub od ostatniego elementu struktury do docelowego. W dwukierunkowej liście czy dojście nastąpi od przodu czy od tyłu zależy od mechanizmu.

Pomiary czasu wykonania operacji na tablicach.

Czynniki związane z algorytmem

- Złożoność obliczeniowa: Fundamentalnym czynnikiem wpływającym na czas wykonania operacji jest jej złożoność obliczeniowa, wyrażana często w notacji dużego O.

- Realokacja pamięci: W przypadku tablic dynamicznych, operacje wymagające realokacji pamięci mogą znacznie wpłynąć na czas wykonania.

Czynniki sprzętowe

- Pamięć: Dostępność i szybkość pamięci RAM mogą wpłynąć na czas wykonania operacji.
- Procesor: Częstotliwość taktowania procesora oraz liczba rdzeni również mają wpływ na szybkość wykonywania operacji.

Czynniki systemowe

- Zarządzanie pamięcią przez system operacyjny: Może mieć wpływ na wydajność operacji na tablicach.
- Obciążenie systemu: Równoległe wykonywanie innych procesów i operacji systemowych może obciążać procesor i pamięć.
- Optymalizacje kompilatora: Kompilatory mogą różnie optymalizować kod, co wpływa na wydajność operacji.

Czynniki związane z implementacją

- Język programowania: Różne języki programowania mogą mieć różną efektywność w zarządzaniu pamięcią i wykonywaniu operacji.
- Sposób implementacji: Konkretne detale implementacji mogą mieć znaczący wpływ na wydajność.

Inne czynniki

- Rozmiar danych: Większe tablice wymagają więcej czasu na przetworzenie.
- Rozkład danych: Początkowy rozkład danych w tablicy może wpłynąć na czas wykonania niektórych operacji.

Opisy metod klas

- deleteAll(): Usuwa wszystkie elementy z listy.
- deleteFromMiddle(): Usuwa element ze środka listy (lub element znajdujący się najbliżej środka w przypadku parzystej liczby elementów).
- addAtMiddle(const T value): Dodaje element w środku listy (lub po lewej stronie środka w przypadku parzystej liczby elementów).
- addFirst(const T value): Dodaje element na początku listy.
- addLast(const T value): Dodaje element na końcu listy.
- contains(const T value): Sprawdza, czy lista zawiera określoną wartość.
- deleteFirst(): Usuwa pierwszy element z listy.
- deleteLast(): Usuwa ostatni element z listy.
- get(int index): Zwraca wartość elementu na określonym indeksie.
- getSize() const: Zwraca liczbę elementów w liście.
- print(): Pomocnicza metoda do wypisywania zawartości listy

Różnice implementacyjne pomiędzy pojedyńczo wiązaną listą, a listą podwójnie wiązana

- Węzły w DoubleLinkedList, każdy węzeł zawiera referencje do poprzedniego i następnego węzła (prev, next), co pozwala na łatwe przemieszczanie się w obu kierunkach po liście.

  W SinglyLinkedList, węzły zawierają tylko referencję do następnego węzła (next), co ogranicza przemieszczanie się do przodu listy.

- Dodawanie i usuwanie elementów w DoubleLinkedList, dodawanie i usuwanie elementów z dowolnej pozycji, w tym z początku i końca listy, jest bardziej bezpośrednie dzięki dostępowi do poprzedników. Na przykład, metoda deleteLast bezpośrednio ustawia prev poprzedniego węzła na nullptr i aktualizuje tail.

  W SinglyLinkedList, usunięcie ostatniego elementu wymaga przeszukania całej listy do przedostatniego elementu, aby ustawić jego next na nullptr i zaktualizować tail.

Porównanie złożoności czasowej ?

addFirst1_1 oraz addFirst1_2

Jak można zauważyć, dodawanie elementów na początku struktury jest najszybsze, jednakowe dla pojedynczo i podwójnie wiążącej listy. Dodawanie elementów na początku struktury jest znacząco wolniejsze dla tablicy dynamicznej. Wynika to z tego, że tablica po uzyskaniu granicznej, zdefiniowanej w klasie pojemności, musi zaalokować nową pamięć, a następnie przepisać stare wartości do powiększonej tablicy. Operacja ta jest

bardzo kosztowna czasowo, dlatego w przypadku częstego dodawania elementów na początku zalecane jest stosowanie list.

W celach optymalizacyjnych, znając przybliżoną ilość elementów przechowywanych w tablicy, warto ustawić odpowiednią wartość początkowej pojemności (*initialCapacity*), ilość danych, przy której następuje inicjalizacja powiększonej tablicy i realokacja starych danych, oraz mnożnik powiększenia nowej tablicy.

Metody get1 (getFromBeginning), get2 (getFromEnd) oraz getFromMiddle

Jeżeli dane na całym swoim zakresie (w szczególności elementy środkowe) będą często odczytywane, należy zastosować tablicę dynamiczną, ponieważ niezależnie od lokalizacji elementu, tablica oferuje stały dostęp $O(1)$ do elementów.

Jeżeli częstą operacją będzie odczyt danych znajdujących się na początku struktury lub w pierwszej połowie zawartości struktury, zalecane jest zaimplementowanie pojedynczo linkowalnej listy. Z tego wynika, że pojedynczo wiązana lista jest szczególnym przypadkiem, kiedy jej implementacja jest warta zastosowania.

W przypadku, gdy dane często będą odczytywane od końca lub od początku struktury, należy zaimplementować podwójnie wiązaną listę, ponieważ umożliwia szybki odczyt danych z początku oraz końca struktury. Najgorszym przypadkiem jest sytuacja, w której musimy odczytać wartości środkowe, ponieważ zarówno od końca jak i początku struktury ilość referencji, przez które musimy przejść, jest taka sama. W niektórych językach programowania, np. Java, domyślną implementacją listy wiązanej jest podwójnie wiązana lista, ponieważ małym kosztem (każda encja ma jedną referencję więcej) idzie uzyskać dużą optymalizację czasową.

addMiddle

W przypadku tablicy dynamicznej operacja dodawania elementu jest zależna od lokalizacji elementu w strukturze oraz od tego, czy znajduje się wolne miejsce w dynamicznej tablicy, i nie trzeba powiększać tablicy. Najgorszym przypadkiem jest dodawanie elementu na początku, wraz z potrzebą zwiększenia tablicy, ponieważ musi nastąpić operacja zwiększenia tablicy, a następnie każdy element, który znajduje się za dodawanym elementem, musi zostać przesunięty.

Dla list, jedynym kosztem czasowym jest dostanie się do danego elementu, usunięcie wskaźników dla elementów pomiędzy środkowym elementem, oraz dodanie nowych referencji (2 lub 4) w zależności od tego, czy jest zastosowana pojedynczo wiązana lub podwójnie wiązana lista.

Według naszych badań w testowanym zakresie tablica dynamiczna okazała się szybsza od list wiążących. Naszym zdaniem, może to być spowodowane tym, że operacje przydzielania nowych zasobów oraz przepisywania wartości są bardzo szybkie, i dopiero przy znacząco większej ilości danych widać przewagę list wiązanych.

deleteFirst

Dla tej operacji najlepszym wyborem będzie pojedynczo wiązana lista, następnie podwójnie wiązana lista, i najgorzej sprawdzi się tablica dynamiczna.

deleteLast

Dla tego przypadku najgorszym rozwiązaniem jest zastosowanie SingleLinkedList, ponieważ do usunięcia elementu, potrzeba przechodzić przez wskaźniki każdej encji struktury, aż do ostatniego elementu. Następnie należy rozważyć zastosowanie tablicy dynamicznej, lub podwójnie wiązanej listy, gdyż operacja usuwania ostatniego elementu ma taką samą złożoność czasową $O(1)$.

Kiedy używać DynamicArray?

- Gdy potrzebny jest szybki dostęp do elementów po indeksie.

- Gdy operacje dodawania i usuwania są głównie wykonywane na końcu listy.
- W aplikacjach, gdzie przewidywalna alokacja pamięci i wydajność dostępu są ważniejsze niż koszt potencjalnego przesuwania elementów.

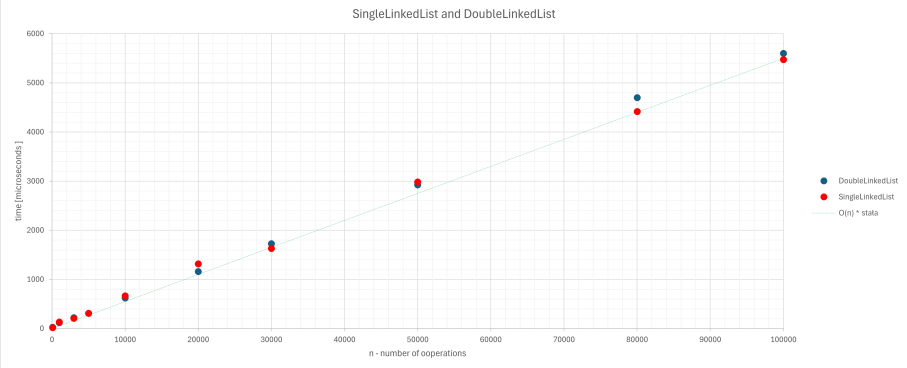
Kiedy używać LinkedList?

- Gdy często dodajemy lub usuwamy elementy z początku lub środka kolekcji.
- W przypadkach, gdy niezbędna jest duża elastyczność zarządzania pamięcią bez kosztów związanych z realokacją.
- Gdy liczba elementów jest nieprzewidywalna lub dynamicznie zmienia się w czasie, a koszt realokacji w "DynamicArray" mógłby być problematyczny.

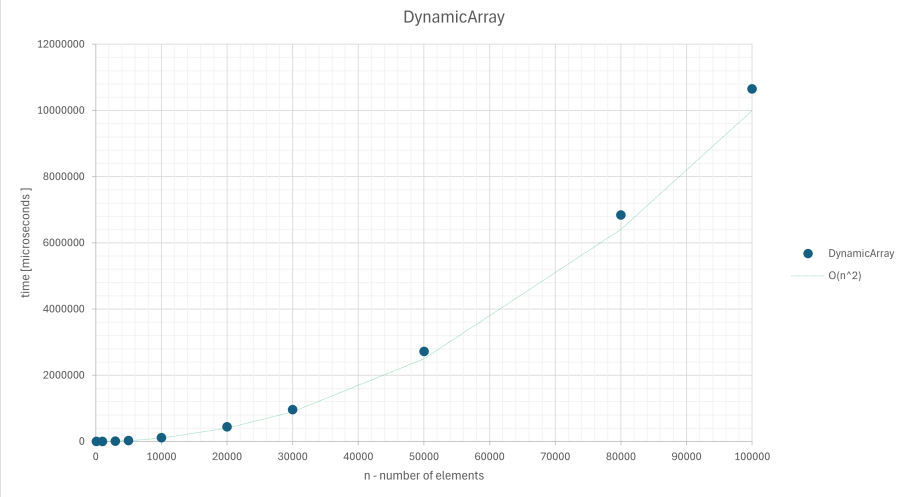
Wyniki Pomiarów:

Dodawanie elementów na początku struktury(add first$_{11}$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DynamicArray [µs]
100	18	27.9	26.4
1000	133.2	120.6	3853.9
3000	207.8	222.1	9705.5
5000	309.6	311.5	27286.7
10000	664.2	620.6	110891.3
20000	1316.2	1160.4	442916.2
30000	1629.4	1725.9	961124.7
50000	2982.8	2922	2719149.4
80000	4413.7	4698.9	6837579.7
100000	5470.9	5599.1	10654356.9



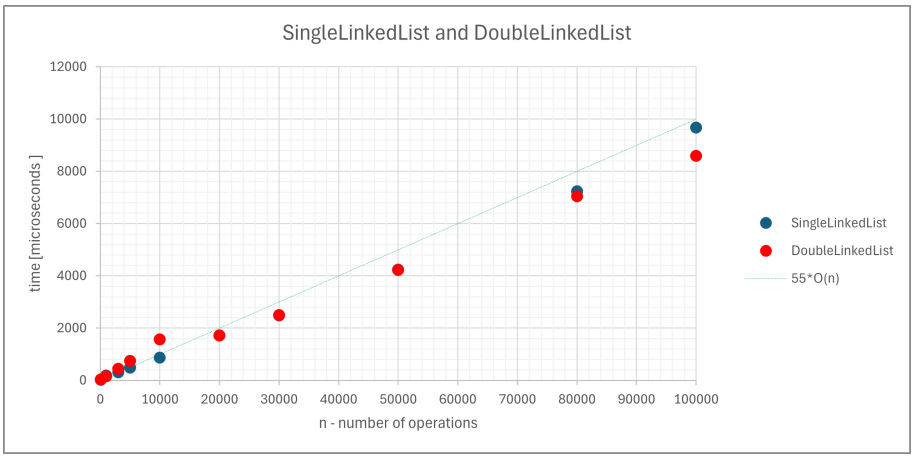
Zdjęcie 5: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



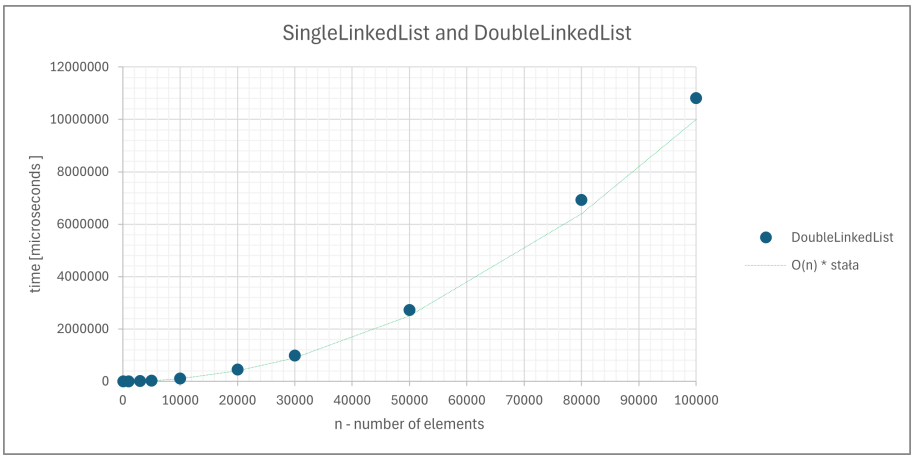
Zdjęcie 6: Tablica dynamiczna

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	26.1	32.4	19.2
1000	184.8	160	1353.5
3000	308.2	445.1	11002.3
5000	488.5	742.5	27616.4
10000	867.8	1572.4	108293.3
20000	1715.5	1720.5	447966.7
30000	2491.8	2501.3	988892.8
50000	4239.6	4226.1	2716774.9
80000	7239.6	7040	6922654.7
100000	9677.2	8594.5	10809018.5

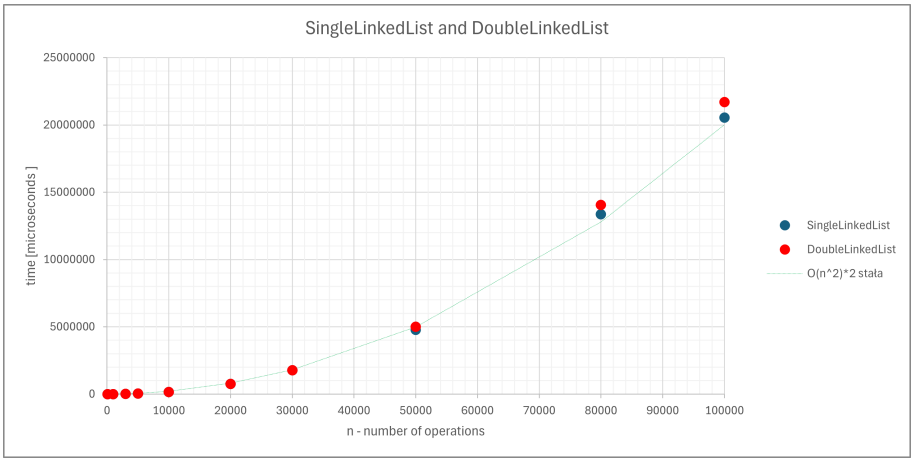
Dodawanie elementów na początku struktury(add first$_{12}$)



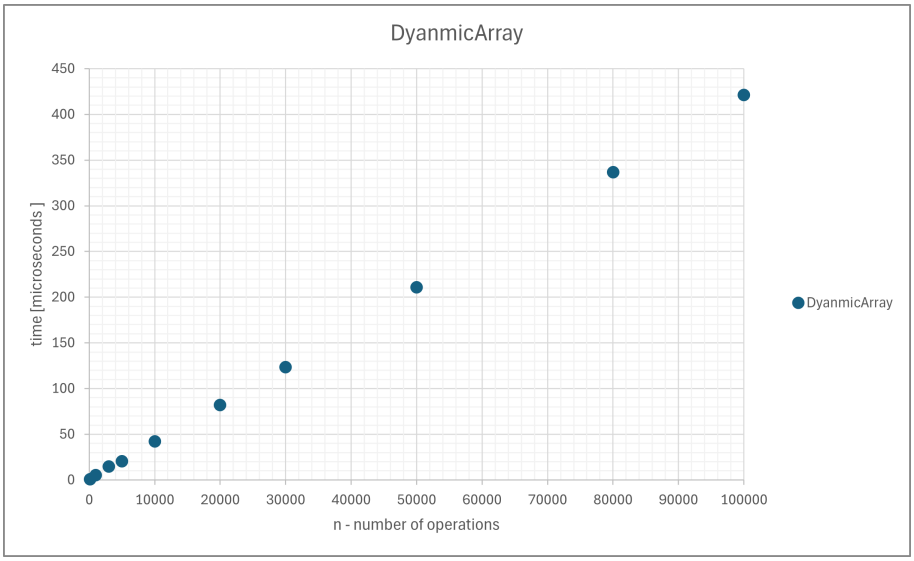
Zdjęcie 7: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



Zdjęcie 8: Tablica dynamiczna



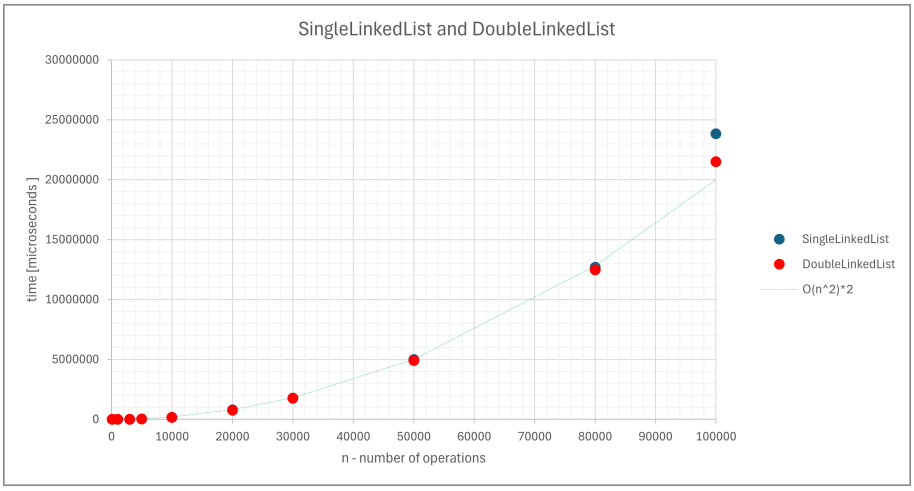
Zdjęcie 9: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



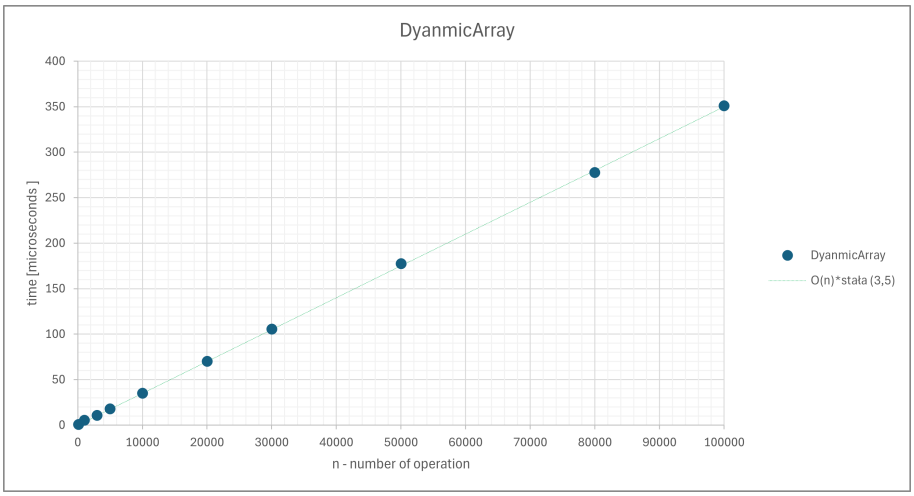
Zdjęcie 10: Tablica dynamiczna

Wyszukiwanie elementów iterując od końca struktury(get$_{12}$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	10.8	13.7	0.6
1000	1133.7	1448.5	5.5
3000	13475.6	13573.9	10.7
5000	38194.4	38208.2	18
10000	164025.9	162353.7	35.2
20000	783639.4	759535.6	70.1
30000	1775310.9	1773991	105.5
50000	4990044.9	4909880.9	177.4
80000	12679194.3	12488928.9	277.7
100000	23822642.7	21501652.3	351.1



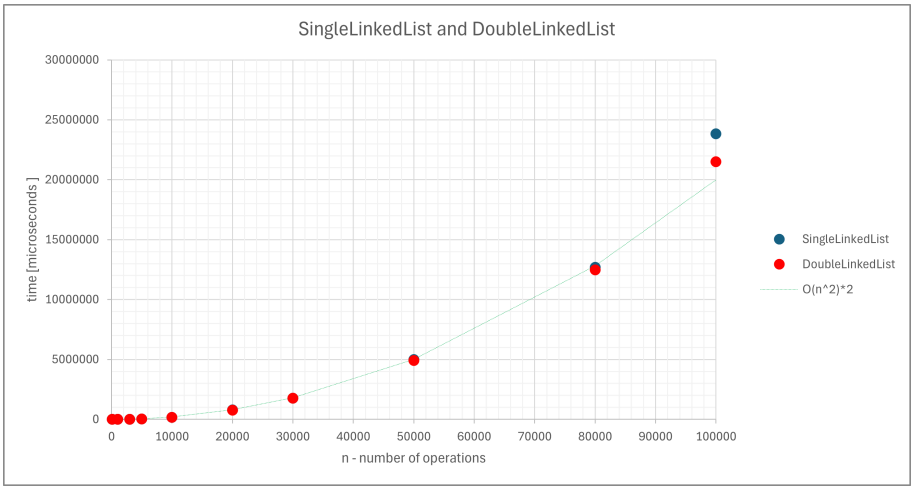
Zdjęcie 11: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



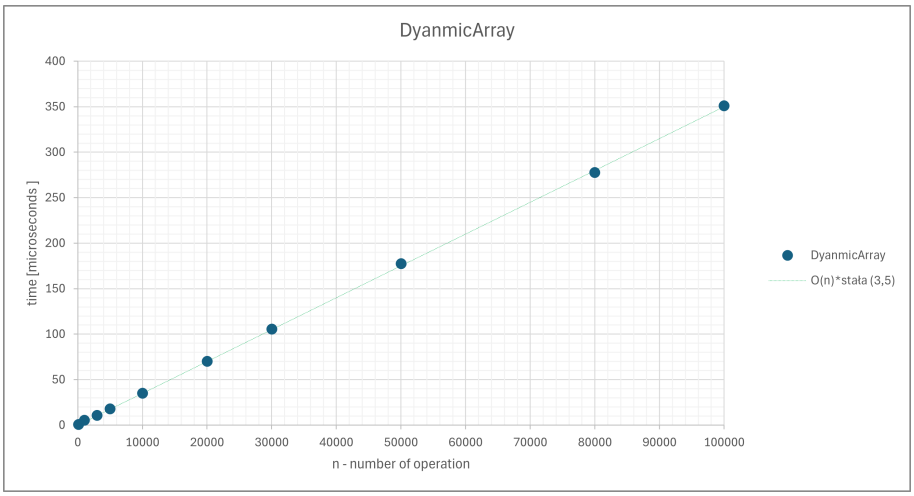
Zdjęcie 12: Tablica dynamiczna

Wyszukiwanie elementu środkowego (getMiddle)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	0.4	0.3	0.1
1000	1.6	1.7	0.1
3000	5.3	4.8	0.1
5000	8.8	8	0.1
10000	21.9	21.5	0.1
20000	62.2	53.9	0.1
30000	91.2	109.6	0.1
50000	194	228.3	0.1
80000	310.6	356	0.1
100000	414.2	407.4	0.1



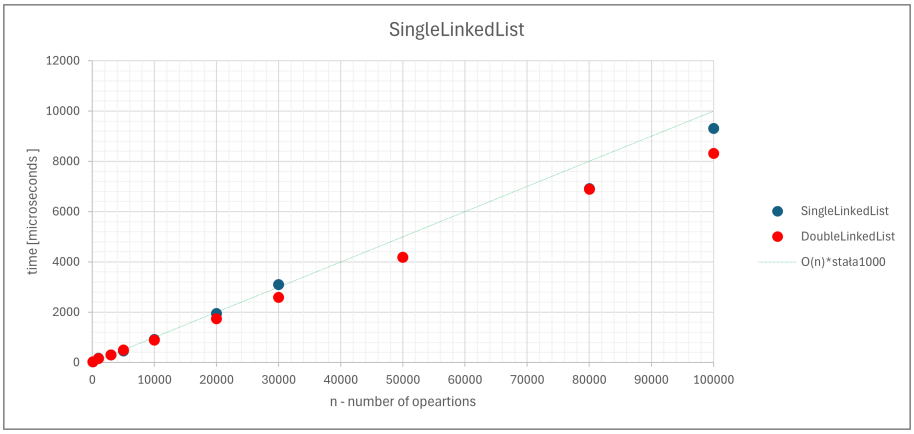
Zdjęcie 13: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



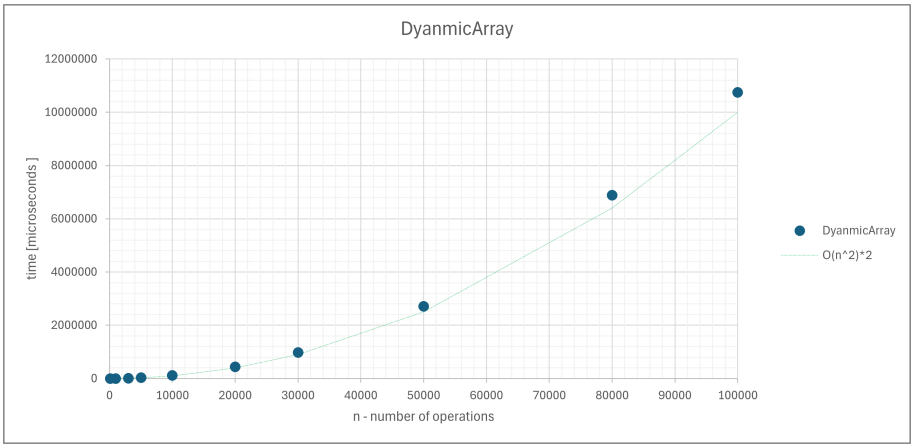
Zdjęcie 14: Tablica dynamiczna

Dodawanie elementów na końcu struktury(addLast$_5$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	24.5	19.8	19.2
1000	163.9	149.8	1299.4
3000	300.6	293.6	11038.9
5000	461	491	26785.2
10000	917.1	888.8	110470.9
20000	1950	1734.7	434228.7
30000	3090.3	2588.3	979794.9
50000	4175.8	4188.1	2708246
80000	6916.6	6895.2	6880817.2
100000	9313.5	8319.5	10746429.8



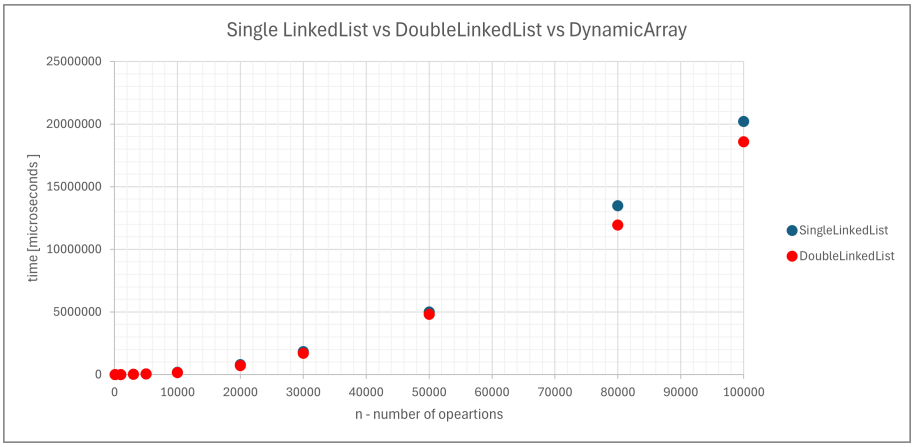
Zdjęcie 15: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



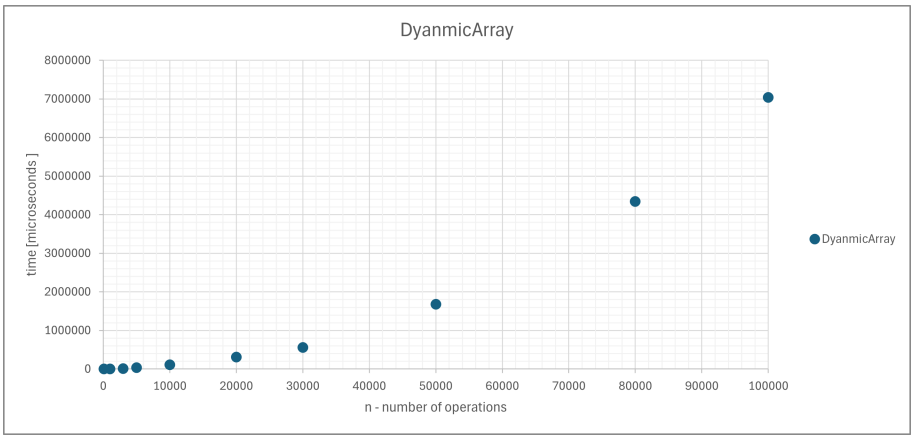
Zdjęcie 16: Tablica dynamiczna

Dodawanie elementów na środek struktury(addMiddle$_6$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	47.2	74.3	15.1
1000	1257.8	1241.8	838
3000	14703.8	14511.1	12348.7
5000	45188.6	42695	34588
10000	174234.4	146659.7	107740.2
20000	793272.2	723681.7	312664.8
30000	1843301.2	1712237	557015.1
50000	4993537.3	4815113.4	1678470.2
80000	13488982.2	11946169.1	4345318.4
100000	20214486.8	18587409.3	7041270



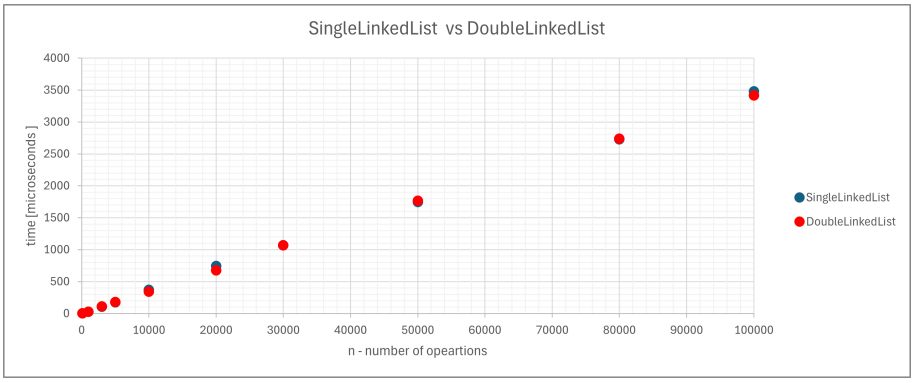
Zdjęcie 17: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



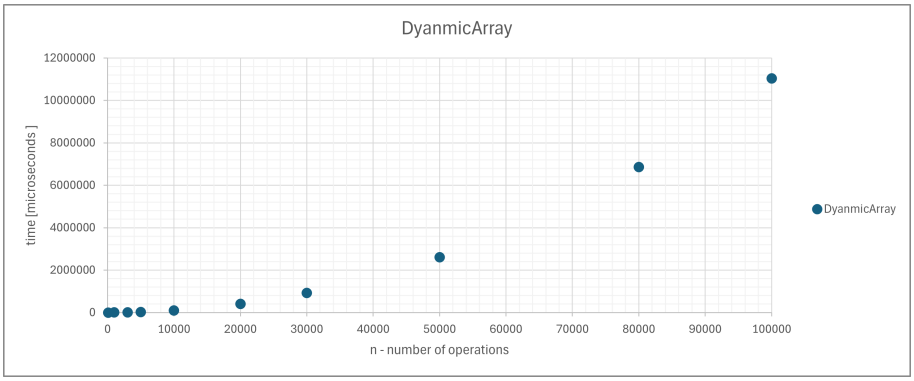
Zdjęcie 18: Tablica dynamiczna

Usuwanie elementów od początku struktury(deleteFromMiddle$_7$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	5.1	4.5	13
1000	29.8	29.9	1194.6
3000	106.9	111.6	9175
5000	176.8	179.3	26744.4
10000	370.7	344.6	101102
20000	746.4	673.8	411985.7
30000	1068.4	1070.3	928219.6
50000	1745.8	1765.7	2612258.2
80000	2727.2	2739	6864635.6
100000	3479	3414.2	11032343.7



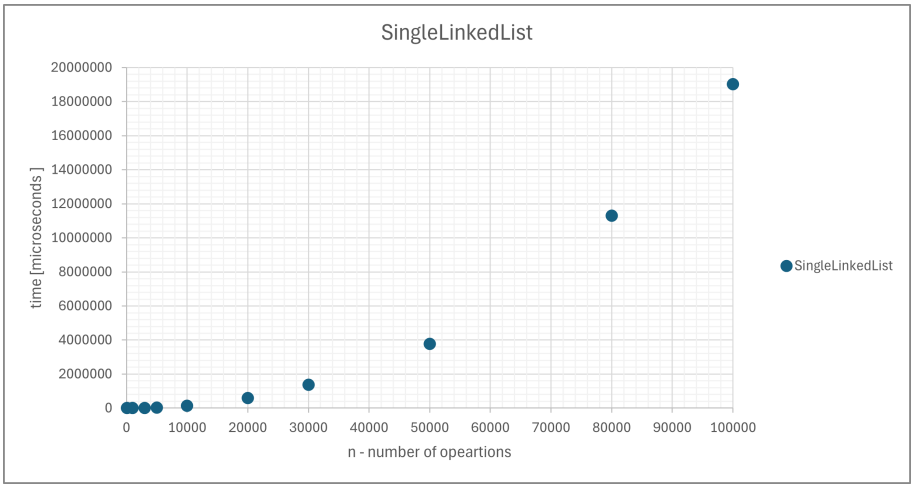
Zdjęcie 19: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana



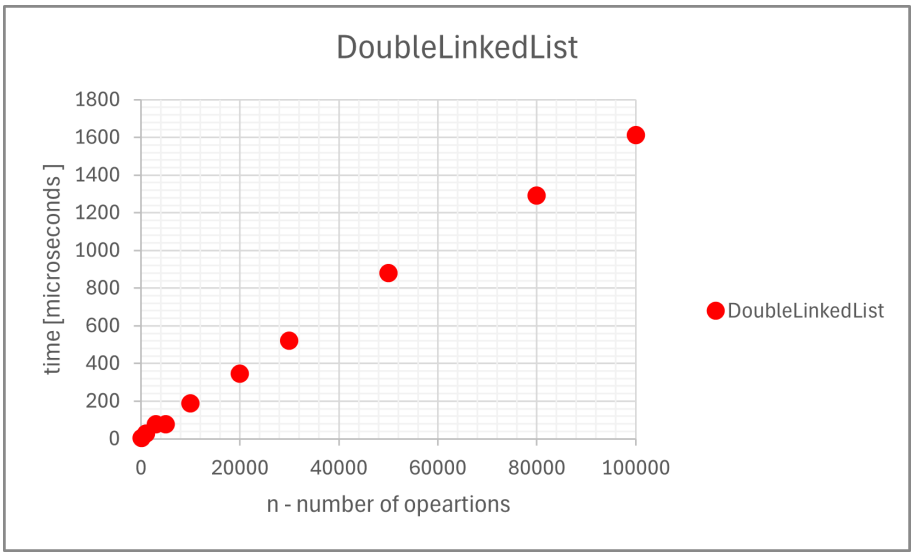
Zdjęcie 20: Tablica dynamiczna

Usuwanie elementów od końca struktury(deleteLast$_8$)

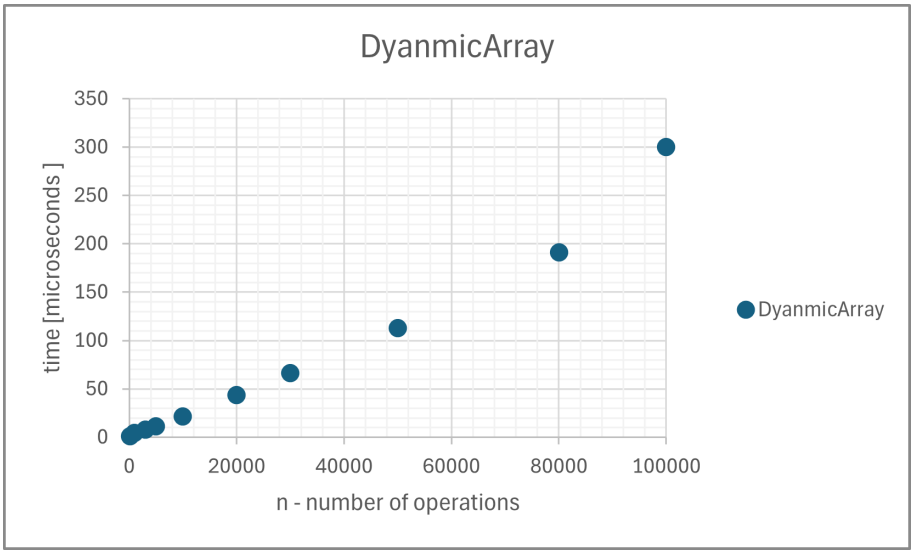
n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	15	3.7	0.8
1000	1169.8	27.7	4.6
3000	11753.1	77.2	7.6
5000	32111.3	76.1	11.3
10000	131337	187.8	21.7
20000	587830.9	345.9	43.5
30000	1381974.5	520	66.5
50000	3764197.3	881.2	112.7
80000	11299327.8	1292.5	191.1
100000	19020744.1	1612.6	299.8



Zdjęcie 21: Jednokierunkowa lista wiązana



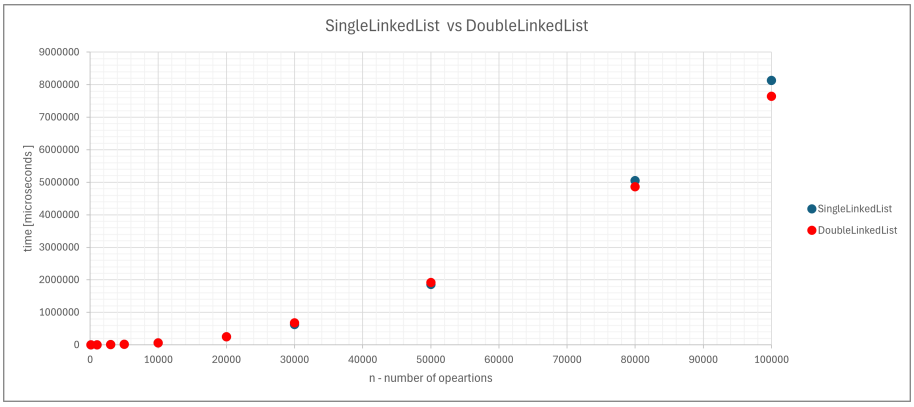
Zdjęcie 22: Dwukierunkowa lista wiązana



Zdjęcie 23: Tablica dynamiczna

Usuwanie elementów od środka struktury(deleteFromMiddle$_9$)

n	SingleLinkedList [µs]	DoubleLinkedList [µs]	DyanmicArray [µs]
100	7.4	7.2	12.2
1000	490	546.3	979.9
3000	5626.9	6236.7	4642.8
5000	15154.2	15010.1	12923.9
10000	57651.9	58897.6	51182.9
20000	252955.3	252063.3	205056.9
30000	625593.7	680811.6	456196.9
50000	1856102.6	1916777.1	1301557.3
80000	5046762.6	4864462.7	3295046.1
100000	8134661.8	7642589.1	5176971.3



Zdjęcie 24: Jednokierunkowa lista wiązana i dwukierunkowa lista wiązana