Struktury danych	
Kierunek *Informatyczne Systemy Automatyki*	**Termin** *środa TN* $15^{15} - 16^{55}$
Imię, nazwisko, numer albumu *Iwo Chwiszczuk 280043, Julia Wilkosz 280040*	**Data** *09/04/2025*
Link do projektu `https://github.com/iwonieevo/Struktury-danych`	

Sprawozdanie – miniprojekt nr 1

Spis treści

1 Wstęp teoretyczny

1.1 Tablica dynamiczna (ang. *Array List*)

Tablica dynamiczna to struktura danych, która umożliwia przechowywanie elementów w sposób ciągły w pamięci i automatyczne zarządzanie rozmiarem. W przeciwieństwie do zwykłej tablicy statycznej, dynamiczna może się rozszerzać (zwykle przez alokację nowej tablicy o większym rozmiarze i skopiowanie danych).

Tabela 1: Teoretyczna złożoność obliczeniowa tablicy dynamicznej[1]

Operacja	Optymistyczna	Średnia	Pesymistyczna
`append`	O(1)	O(1) (amort.)	O(1) (amort.)
`prepend`	O(n)	O(n)	O(n)
`insert`	O(1)	O(n)	O(n)
`pop front`	O(n)	O(n)	O(n)
`pop back`	O(1)	O(1)	O(1)
`remove`	O(1)	O(n)	O(n)
`search`	O(1)	O(n)	O(n)

[1] Źródła: Jarosław Rudy, *listy, tablice dynamiczne, listy wiązane*, `www.bigocheatsheet.com`, `www.geeksforgeeks.org/dynamic-array-in-c/`

Zastosowania struktury i korzyści z niej wynikające

Tablice dynamiczne są szeroko wykorzystywane w aplikacjach, gdzie:

- potrzebna jest szybka indeksacja (O(1) dostęp do elementu)
- rozmiar danych nie jest znany z góry
- dominuje operacja `append` (np. buforowanie danych, logi, kolekcje danych wejściowych)

Przykładowe zastosowania:

- implementacja stosu, kolejki
- bufory tekstowe w edytorach (np. lista linii w pliku)
- przechowywanie danych wejściowych w programach
- w strukturach takich jak hash mapy (jako łańcuchy kolizji)

Korzyści korzystania z listy dynamicznej:

- prosta implementacja i dostęp do elementów
- wydajność w scenariuszach z częstymi dodaniami na koniec
- elastyczność rozmiaru przy niskim koszcie kopiowania (rzadko wykonywana realokacja dzięki strategii podwajania rozmiaru)

1.2 Lista jednokierunkowa (ang. *Singly Linked List*)

Lista jednokierunkowa to dynamiczna struktura danych składająca się z węzłów. Każdy węzeł zawiera dane oraz wskaźnik do następnego elementu. W przeciwieństwie do tablicy dynamicznej, elementy nie są przechowywane w sposób ciągły w pamięci, co pozwala na łatwe wstawianie i usuwanie elementów bez konieczności przesuwania pozostałych.

Tabela 2: Teoretyczna złożoność obliczeniowa listy jednokierunkowej (implementacja z *tail*)[2]

Operacja	Optymistyczna	Średnia	Pesymistyczna
`append`	O(1)	O(1)	O(1)
`prepend`	O(1)	O(1)	O(1)
`insert`	O(1)	O(n)	O(n)
`pop front`	O(1)	O(1)	O(1)
`pop back`	O(n)	O(n)	O(n)
`remove`	O(1)	O(n)	O(n)
`search`	O(1)	O(n)	O(n)

Zastosowania struktury i korzyści z niej wynikające

Zastosowania:

- kolejki, listy FIFO, stosy
- alokatory pamięci i systemy operacyjne (zarządzanie wolnymi blokami)
- przechowywanie danych o nieznanym z góry rozmiarze, z częstymi wstawieniami/usuwaniami

Korzyści:

- bardzo efektywne dodawanie/usuwanie elementów z przodu listy (O(1))
- brak potrzeby przesuwania elementów (w odróżnieniu od tablic)
- dynamiczny rozmiar — nie trzeba rezerwować dużych bloków pamięci

[2] Źródła: Jarosław Rudy, *listy, tablice dynamiczne, listy wiązane*, `www.bigocheatsheet.com`, `www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list/`

1.3 Lista dwukierunkowa (ang. *Doubly Linked List*)

Lista dwukierunkowa to dynamiczna struktura danych, w której każdy węzeł zawiera dane oraz dwa wskaźniki: do poprzedniego i do następnego elementu. Pozwala to na bardziej elastyczne poruszanie się po liście i wydajniejsze wstawianie oraz usuwanie elementów z obu końców oraz „ze środka" listy.

Tabela 3: Teoretyczna złożoność obliczeniowa listy dwukierunkowej[3]

Operacja	Optymistyczna	Średnia	Pesymistyczna
`append`	O(1)	O(1)	O(1)
`prepend`	O(1)	O(1)	O(1)
`insert`	O(1)	O(n)	O(n)
`pop front`	O(1)	O(1)	O(1)
`pop back`	O(1)	O(1)	O(1)
`remove`	O(1)	O(n)	O(n)
`search`	O(1)	O(n)	O(n)

Zastosowania struktury i korzyści z niej wynikające

Zastosowania:

- implementacja deque (double-ended queue)
- edytory tekstu — lista linii, swobodne poruszanie się kursorem
- systemy operacyjne — np. zarządzanie procesami (lista procesów)
- nawigacja w historii przeglądarki
- algorytmy z usuwaniem elementów ze środka (np. LRU cache)

Korzyści:

- wydajne wstawianie i usuwanie elementów z obu końców listy
- możliwość poruszania się w obu kierunkach
- usuwanie węzła w czasie O(1) (jeśli mamy wskaźnik)

2 Założenia projektowe

W celu przeprowadzenia analizy wydajnościowej struktur danych przyjęto następujące założenia projektowe:

- **Wielkości struktur:** W ramach przeprowadzonych testów wydajnościowych wykorzystano różne rozmiary struktur danych, które były określone w tablicy:

  ```
  unsigned int SIZES[] = {5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000,
  45000, 50000, 55000, 60000, 65000, 70000, 75000, 80000, 85000, 90000, 95000, 100000};
  ```

  Wszystkie testy były powtarzane 50 razy w celu uzyskania średnich wyników i redukcji wpływu ewentualnych błędów losowych. Dla każdej wielkości struktury, dane zostały generowane zgodnie z opisanym niżej sposobem, a następnie wczytywane do struktur danych (tablicy dynamicznej, listy jednokierunkowej i listy dwukierunkowej).

- **Sposób generowania danych:** Dane wykorzystywane w testach zostały wygenerowane przy użyciu mechanizmów dostępnych w języku C++. Funkcja `generate_random_integers` zapisuje do pliku tekstowego określoną liczbę liczb całkowitych losowanych z zadanego przedziału.

[3] Źródła: Jarosław Rudy, *listy, tablice dynamiczne, listy wiązane*, `www.bigocheatsheet.com`, `www.geeksforgeeks.org/doubly-linked-list/`

Do generowania liczb losowych wykorzystano następujące mechanizmy:

- `std::random_device` jako źródło ziarna losowego,
- `std::mt19937` — silnik Mersenne Twister zapewniający wysoką jakość losowości,
- `std::uniform_int_distribution<int>` do losowania wartości z równomiernym rozkładem z przedziału $[min, max]$.

Dla każdej wartości wygenerowanej przez dystrybutor, liczba zapisywana była w osobnym wierszu pliku tekstowego wskazanego przez parametr `dest_path`. Liczba wygenerowanych wartości była zgodna z parametrem `n_rows`.

W wyniku wywołania tej funkcji powstawał plik tekstowy zawierający `n_rows` liczb całkowitych z przedziału [-10000, 10000], gotowy do wczytania i użycia w strukturach danych. W trakcie działania programu plik był wielokrotnie nadpisywany, a na koniec został on usunięty.

- **Sposób pomiaru czasu:** Pomiar czasu wykonania operacji na strukturach danych został przeprowadzony przy użyciu funkcji `measure_time`, która mierzy czas wykonania wybranej metody na strukturze danych. Pomiar realizowany był za pomocą biblioteki `std::chrono`, która umożliwia precyzyjne śledzenie czasu w nanosekundach.

  Funkcja `measure_time` została zaprojektowana w sposób ogólny, aby mogła mierzyć czas dowolnej metody zdefiniowanej w klasie `List`. Przyjmuje ona wskaźnik na metodę oraz jej parametry, a następnie mierzy czas wykonania operacji.

  Funkcja ta pozwala na dokładny pomiar czasu wykonania metod operujących na strukturach danych, co jest szczególnie przydatne w testach wydajnościowych, gdzie mierzenie czasu dla różnych rozmiarów struktur ma kluczowe znaczenie.

- **Parametry sprzętu:** Testy przeprowadzono na komputerze wyposażonym w procesor AMD Ryzen 9 5900X 12-core o taktowaniu 3.70 GHz, z pamięcią RAM o pojemności 32 GB.

- **Oprogramowanie:** Program został zaimplementowany w języku C++ w standardzie C++23. Kompilacja została przeprowadzona za pomocą kompilatora g++ 14.2.0 w edytorze VSCode. Środowisko testowe stanowił system operacyjny Windows 11.

3 Badania

Dla każdej tabeli zastosowano skróty:

- TD - tablica dynamiczna
- L1 - lista jednokierunkowa
- L2 - lista dwukierunkowa

Tabela 4: Wyniki badań dla metody `append` w ns

rozmiar	TD	TD max	L1	L1 max	L2	L2 max
5000	18014	29400	172	300	160	300
10000	17952	31400	194	1200	170	400
15000	17288	66200	168	1200	184	1400
20000	13444	73800	178	1200	182	1400
25000	10688	73800	184	1200	186	1400
30000	14060	73800	178	1200	174	1400
35000	13538	73800	192	1500	172	1400
40000	18554	73800	186	1500	182	1400
45000	20794	73800	182	1500	190	1400
50000	22720	73800	174	1500	178	1400
55000	26752	73800	198	1500	184	1400
60000	26324	73800	184	1500	216	2500
65000	30642	93400	212	1800	174	2500
70000	30444	93400	180	1800	250	2700
75000	31046	93400	176	1800	176	2700
80000	30738	93400	170	1800	168	2700
85000	32486	93400	186	1800	178	2700
90000	32638	93400	220	1800	184	2700
95000	35736	93400	178	1800	174	2700
100000	39852	93400	180	1800	192	2700

Tabela 5: Wyniki badań dla metody `prepend` w ns

rozmiar	TD	TD max	L1	L1 max	L2	L2 max
5000	19556	34500	174	1100	150	400
10000	24942	67500	164	1100	176	400
15000	23444	71200	150	1100	186	400
20000	29712	71200	174	1100	192	1200
25000	39990	80200	170	1100	212	1200
30000	53182	80200	190	1100	214	1200
35000	55092	80200	174	1100	206	1200
40000	66004	87600	190	1100	228	1200
45000	74310	104900	166	1100	220	1400
50000	79966	111700	188	1100	210	1400
55000	88040	122500	220	1300	218	1400
60000	95458	136400	180	1300	218	1400
65000	108456	142900	192	1300	232	1400
70000	116864	308900	192	1300	216	1400
75000	112892	308900	184	1300	222	1400
80000	117272	308900	182	1300	230	1400
85000	124884	308900	184	1300	204	1400
90000	130574	308900	198	1300	244	1400
95000	140692	308900	180	1300	210	1400
100000	151008	308900	188	1300	214	1400

Tabela 6: Wyniki badań dla metody `insert` w ns

rozmiar	**TD**	**TD max**	**L1**	**L1 max**	**L2**	**L2 max**
5000	16582	29400	8878	13700	8818	15900
10000	18992	44500	18534	58300	19902	47900
15000	14908	44500	38970	111100	30074	95400
20000	18638	44500	45024	111100	44458	121400
25000	24908	47400	50626	136100	49528	157000
30000	31314	47400	63514	192000	59072	168400
35000	38518	96800	64160	244500	60924	339800
40000	40896	96800	84362	258200	78988	339800
45000	42944	96800	85262	334900	76248	339800
50000	48924	96800	85924	334900	83110	339800
55000	57502	96800	106106	337100	106674	565800
60000	59936	105200	104800	337100	98178	565800
65000	67486	107400	136052	386300	144908	565800
70000	69586	107400	194210	386300	173884	565800
75000	69198	107400	202122	635400	178162	565800
80000	77158	125600	201082	635400	191354	565800
85000	77762	125600	228296	635400	204610	565800
90000	84078	144500	223640	635400	212876	565800
95000	85624	144500	215024	635400	226652	565800
100000	92250	144500	294384	635400	265950	565800

Tabela 7: Wyniki badań dla metody `pop_front` w ns

rozmiar	**TD**	**TD max**	**L1**	**L1 max**	**L2**	**L2 max**
5000	7226	9200	224	600	202	300
10000	14196	21100	246	800	244	800
15000	21282	41900	252	800	290	1000
20000	27622	41900	290	800	310	1000
25000	36676	57300	322	1000	366	1500
30000	47710	119700	364	1000	360	1500
35000	54756	119700	280	1000	336	1500
40000	61794	119700	338	1000	328	1500
45000	70558	119700	316	1000	288	1500
50000	77854	119700	296	1000	308	1500
55000	84728	121700	330	1000	340	1500
60000	93260	139100	304	1000	426	2300
65000	101146	139100	350	1200	408	2300
70000	105834	152100	402	1200	416	2300
75000	110200	152100	350	1200	428	2300
80000	114104	152500	360	1200	464	2300
85000	125920	162000	448	1200	474	2300
90000	129354	166400	406	1200	456	2300
95000	133956	168900	332	1200	366	2300
100000	144256	205300	372	1200	368	2300

Tabela 8: Wyniki badań dla metody `pop_back` w ns

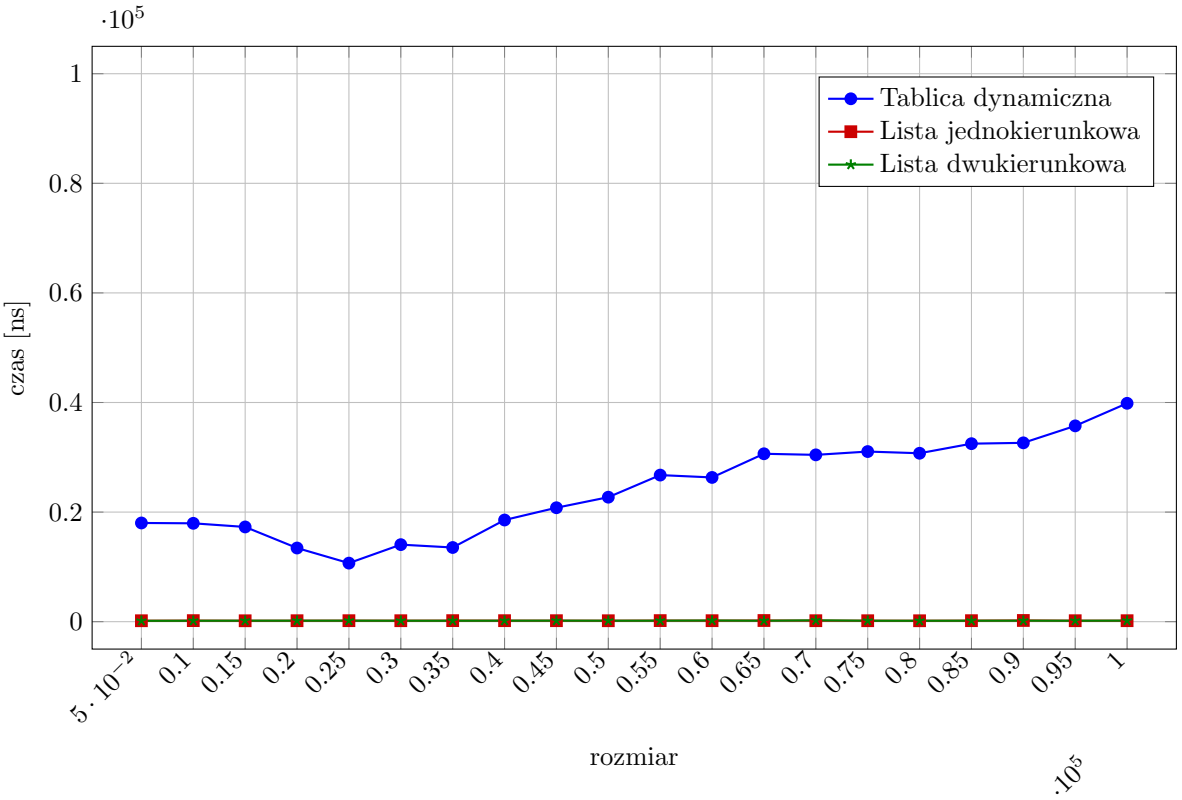
rozmiar	**TD**	**TD max**	**L1**	**L1 max**	**L2**	**L2 max**
5000	70	100	19452	76000	140	200
10000	76	500	35608	157900	156	300
15000	80	500	54184	215100	144	300
20000	72	500	65052	250000	170	600
25000	86	500	78994	250000	166	600
30000	92	500	106856	392400	166	600
35000	90	500	107636	392400	174	700
40000	84	500	146972	401100	184	800
45000	90	500	140206	434700	170	800
50000	86	500	164162	448200	168	800
55000	92	500	177418	448200	168	800
60000	90	500	217174	603900	176	800
65000	92	500	277774	688400	186	800
70000	90	500	331774	1089200	196	800
75000	100	500	344544	1089200	194	800
80000	90	500	340304	1089200	198	800
85000	92	500	390024	1089200	202	800
90000	86	500	362292	1089200	168	800
95000	96	600	436714	1089200	166	800
100000	80	600	482264	1457800	194	800

Tabela 9: Wyniki badań dla metody `remove` w ns

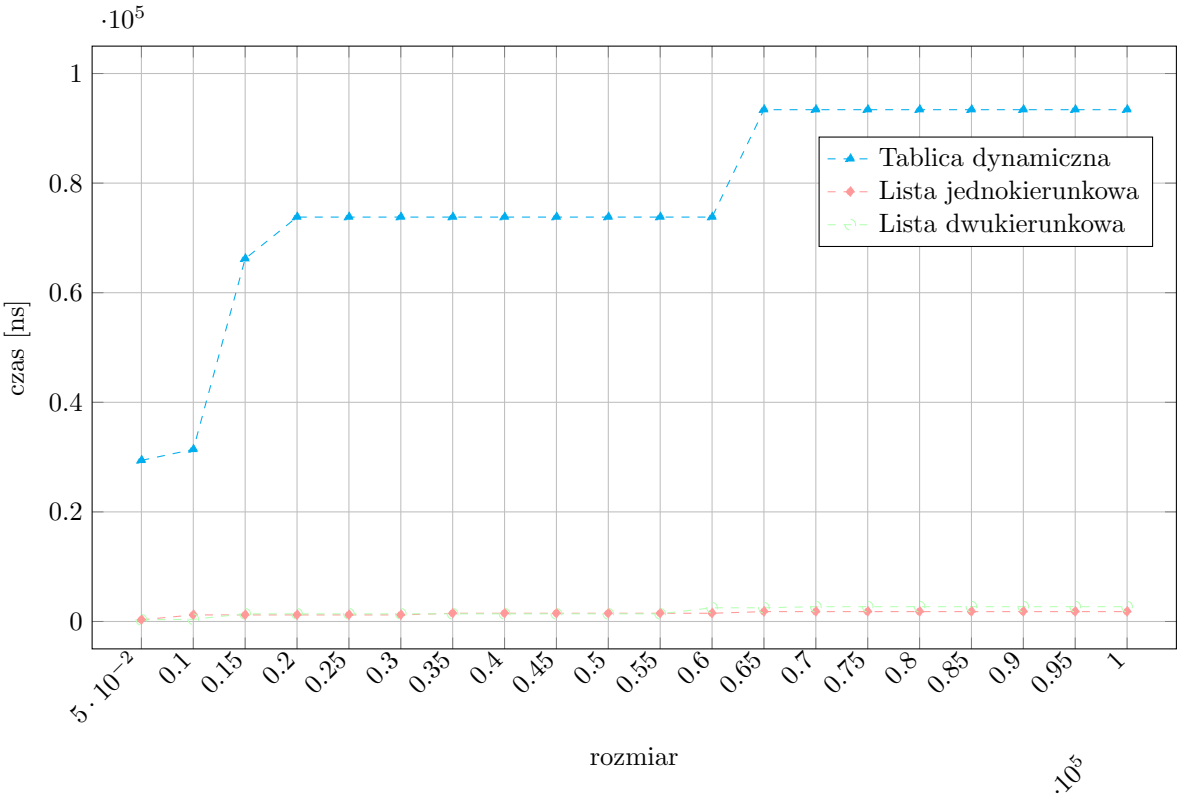
rozmiar	**TD**	**TD max**	**L1**	**L1 max**	**L2**	**L2 max**
5000	3658	5900	10004	27000	8174	13500
10000	7314	11000	19582	71200	15282	35400
15000	10528	13900	33554	87300	22754	66600
20000	14184	17600	39038	125100	29568	116400
25000	18284	22300	44452	145200	34690	116400
30000	24418	109400	64572	306700	41738	116400
35000	25034	109400	62600	306700	59294	268500
40000	31466	109400	82996	306700	57108	268500
45000	32722	109400	78552	306700	62036	268500
50000	37612	109400	74864	306700	70670	268500
55000	40796	109400	113490	472600	87846	279500
60000	43112	109400	105910	472600	90732	401100
65000	47660	109400	153466	572700	107384	401100
70000	51284	109400	156542	572700	125466	401100
75000	53996	109400	184174	572700	134810	429900
80000	56226	109400	191290	572700	127590	429900
85000	58638	109400	219502	572700	156360	429900
90000	64582	109400	223050	572700	148042	429900
95000	66858	109400	246542	572700	158742	429900
100000	69966	109400	293968	572700	181406	494300

Tabela 10: Wyniki badań dla metody `search` w ns

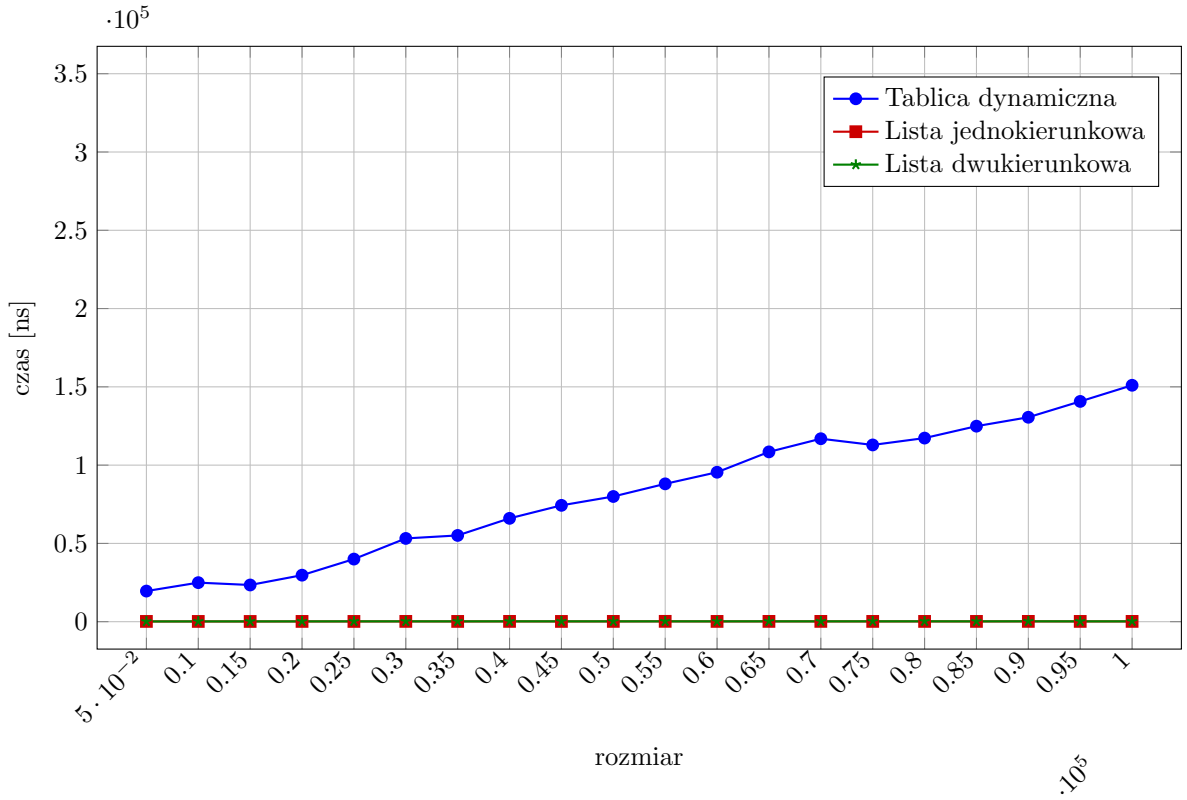
rozmiar	**TD**	**TD max**	**L1**	**L1 max**	**L2**	**L2 max**
5000	4638	7000	19420	61800	19066	44800
10000	9310	15000	29656	61800	37254	106700
15000	13700	20700	56588	183400	57678	128700
20000	17996	29000	67080	193600	72972	196200
25000	24196	54100	67014	193600	94666	241400
30000	28430	54600	95228	265600	131700	387000
35000	34470	60600	99734	284600	133770	387000
40000	38618	61300	134980	483300	177376	569400
45000	45584	89300	153638	483300	191012	569400
50000	45866	89300	124434	483300	201332	569400
55000	50734	89300	163450	550800	263390	695400
60000	56340	89300	176224	550800	259684	695400
65000	62462	107600	226474	645300	344408	846600
70000	66500	130000	271034	680900	347924	846600
75000	65398	130000	322292	680900	352520	846600
80000	71148	130000	342664	744400	385258	1306400
85000	74012	130000	401734	1000200	519974	1306400
90000	79684	130000	380018	1110800	445984	1306400
95000	83748	131700	395842	1110800	490736	1306400
100000	89508	131700	461816	1110800	619912	1306600



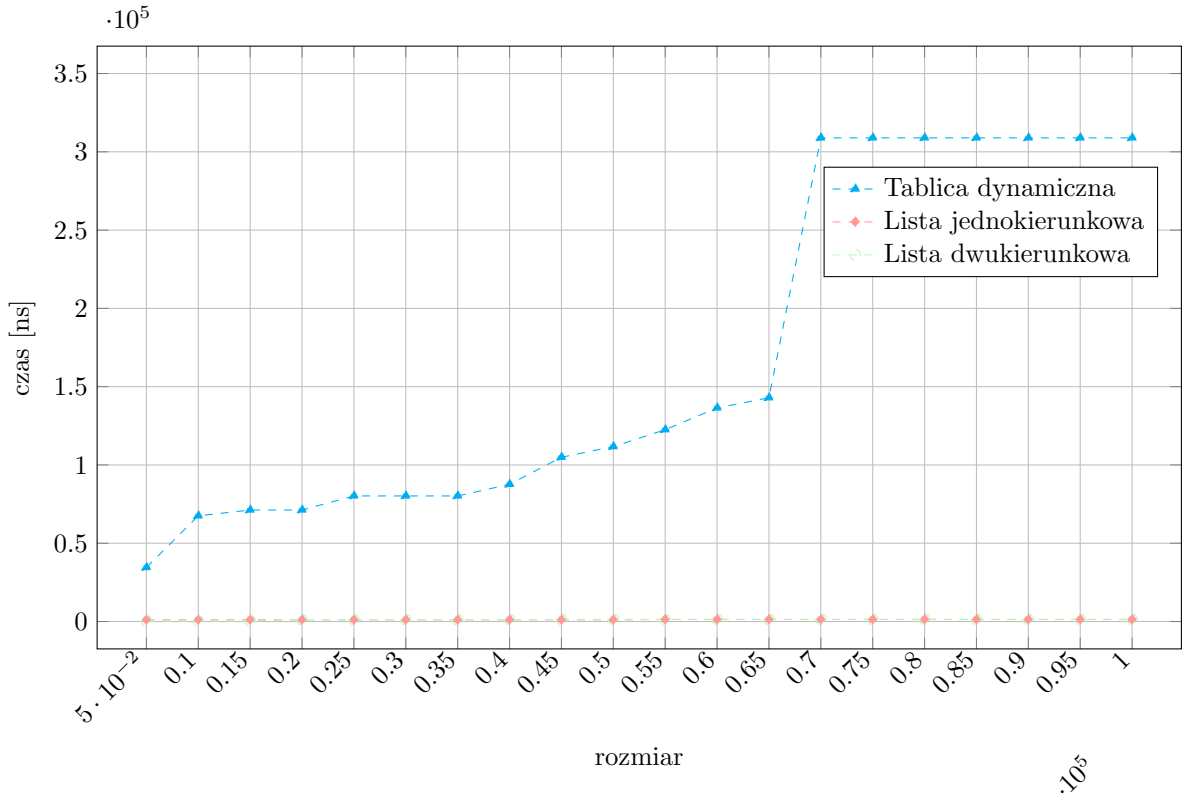
Rysunek 1: złożoność czasowa metody `append`



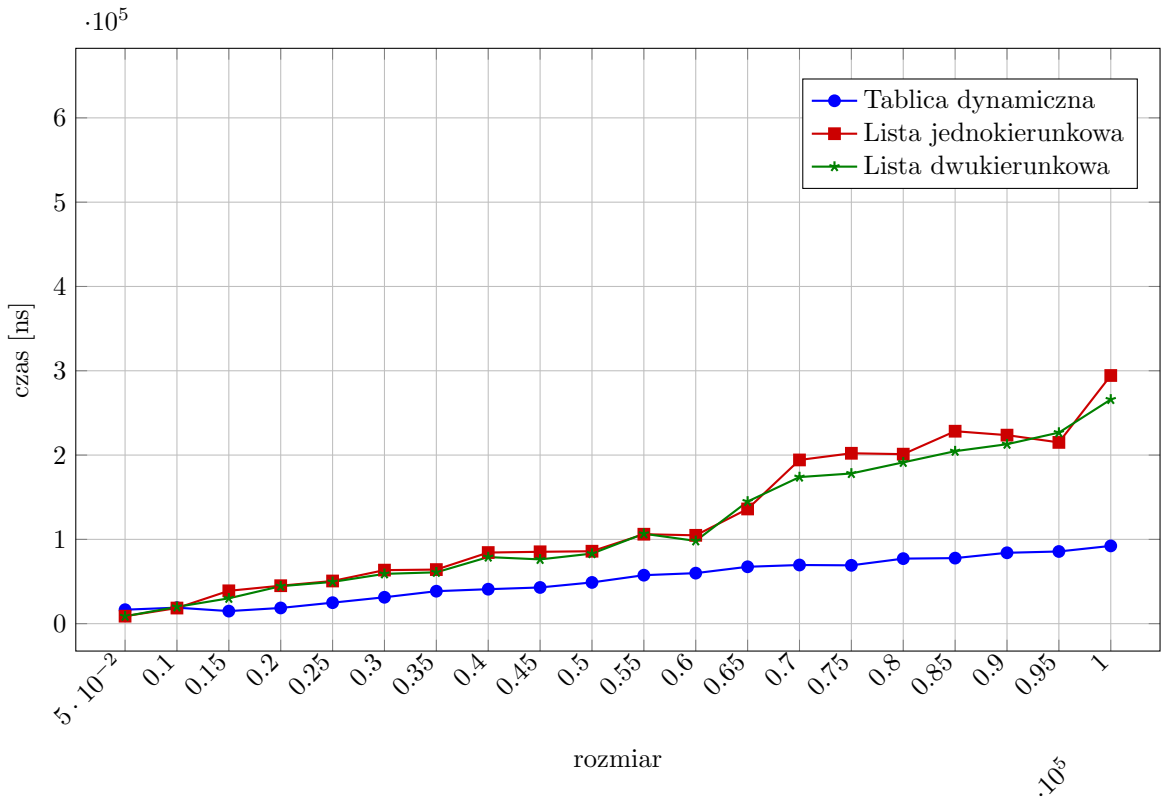
Rysunek 2: maksymalna złożoność czasowa metody `append`



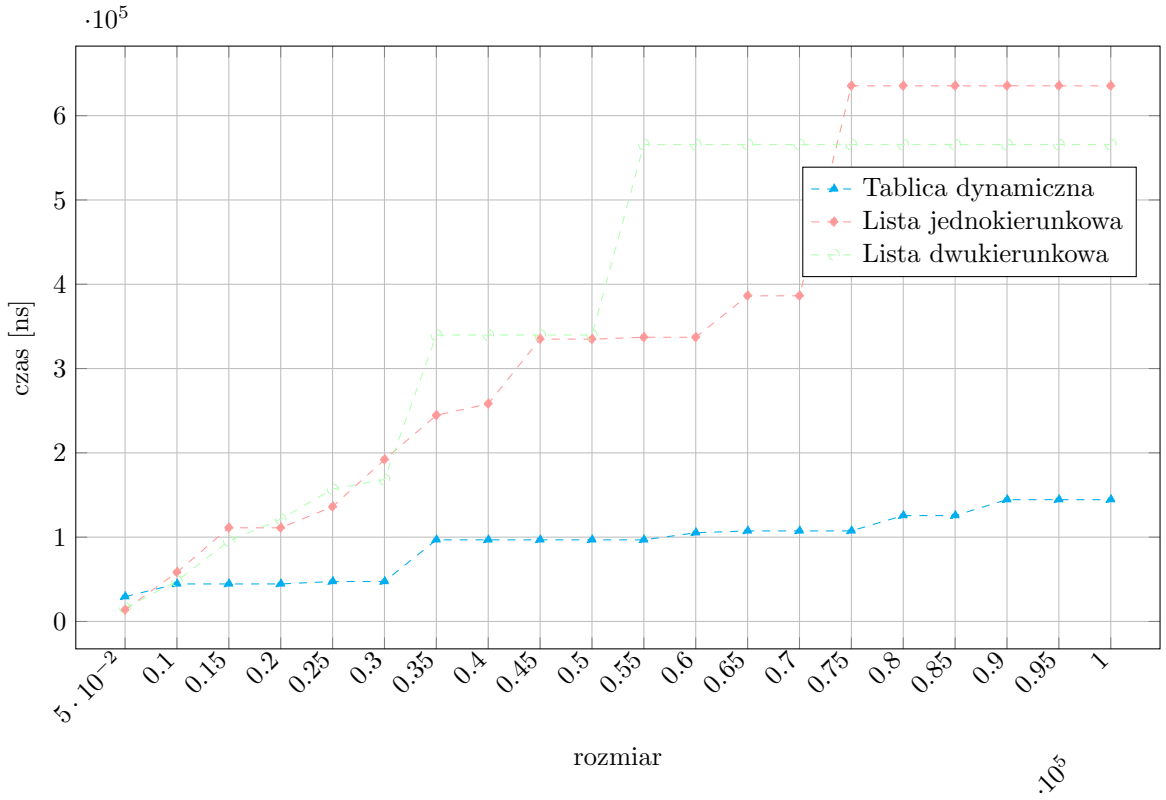
Rysunek 3: złożoność czasowa metody `prepend`



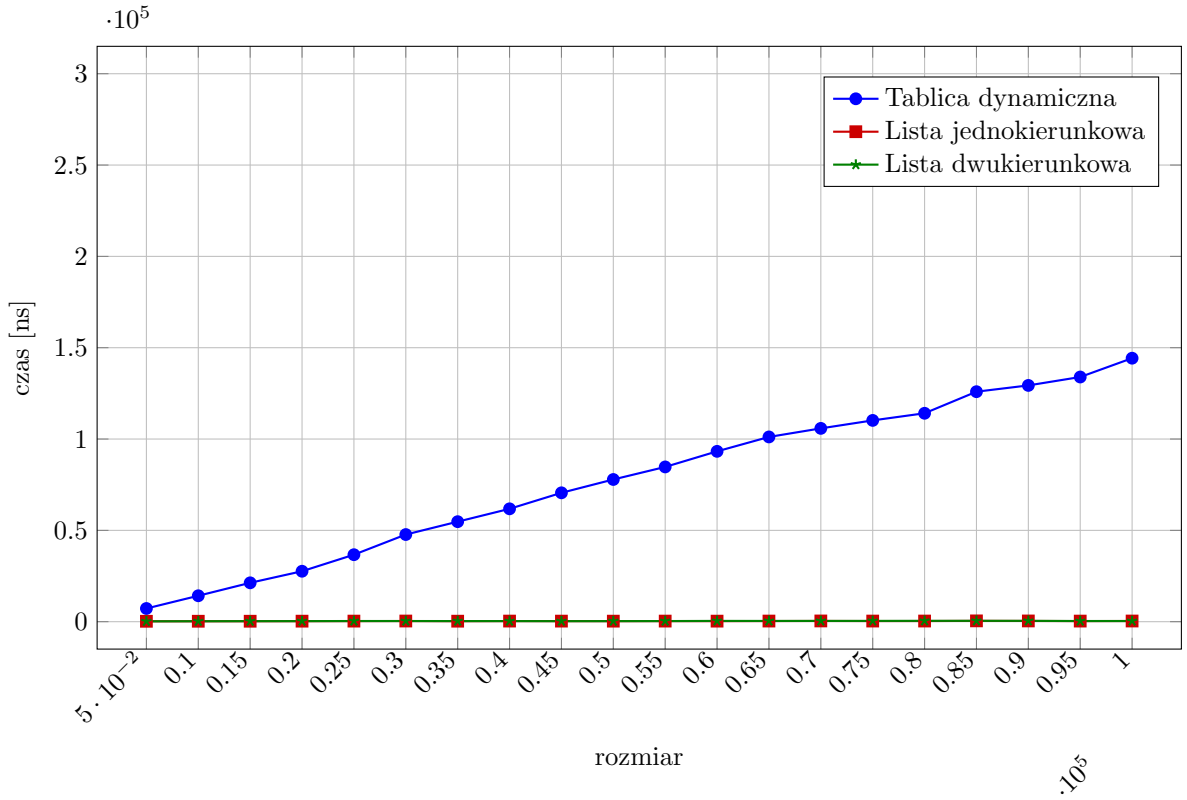
Rysunek 4: maksymalna złożoność czasowa metody `prepend`



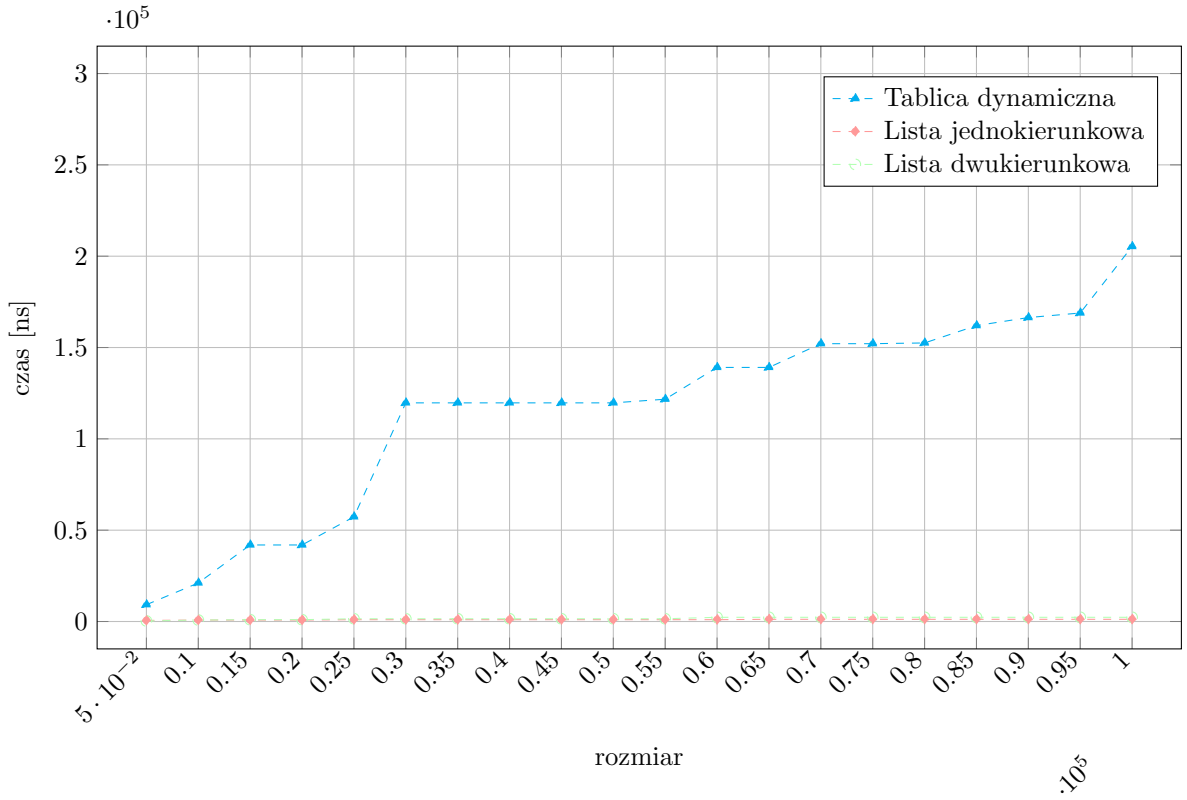
Rysunek 5: złożoność czasowa metody `insert`



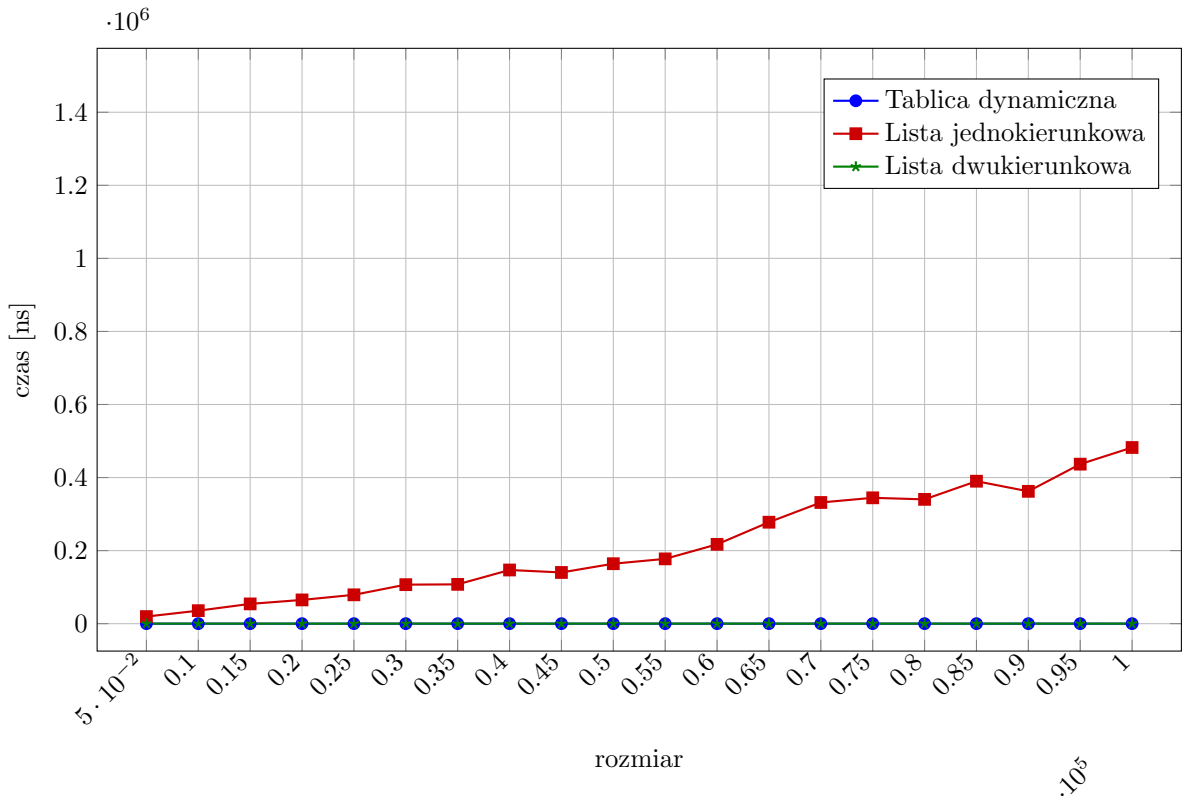
Rysunek 6: maksymalna złożoność czasowa metody `insert`



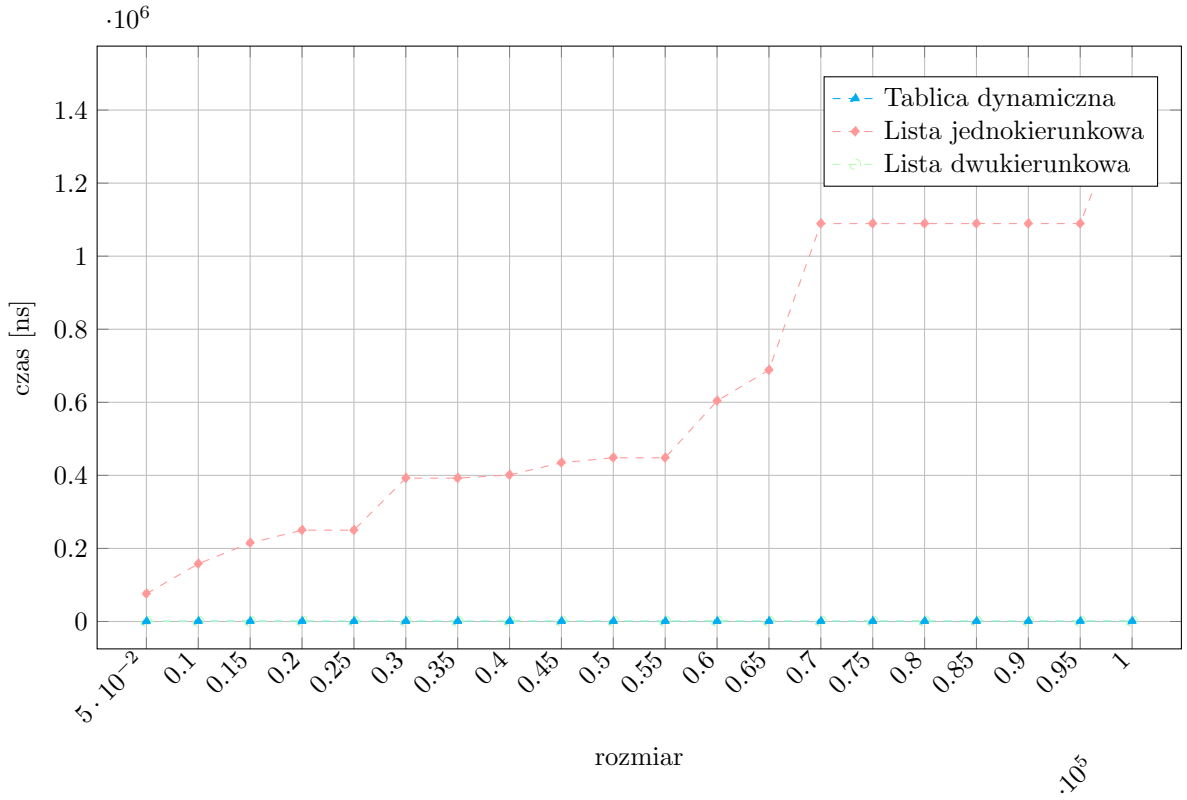
Rysunek 7: złożoność czasowa metody `pop_front`



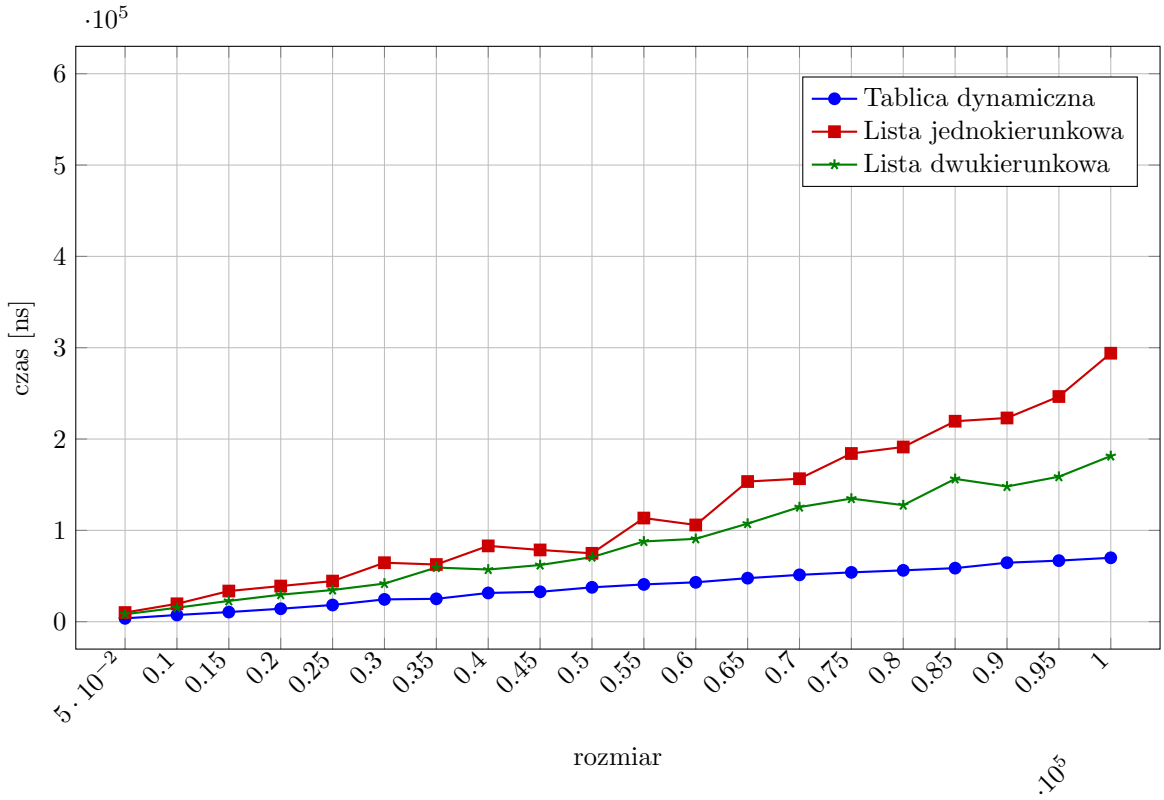
Rysunek 8: maksymalna złożoność czasowa metody `pop_front`



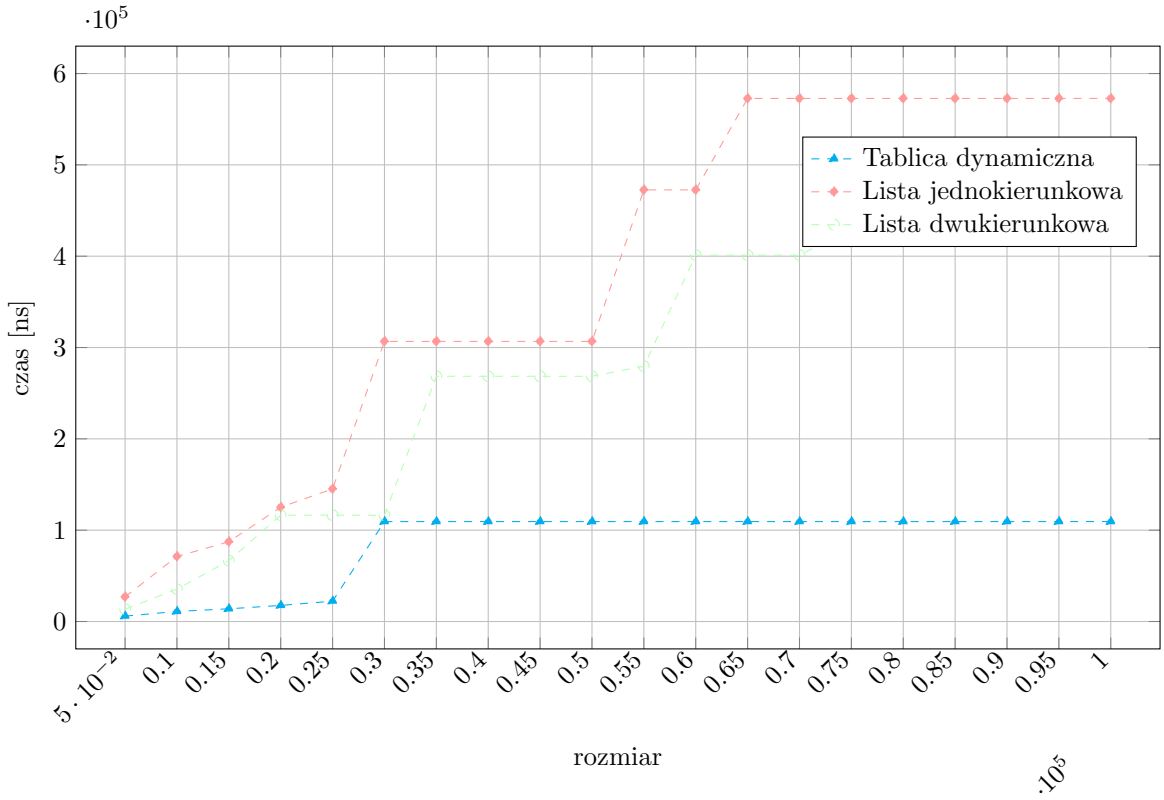
Rysunek 9: złożoność czasowa metody `pop_back`



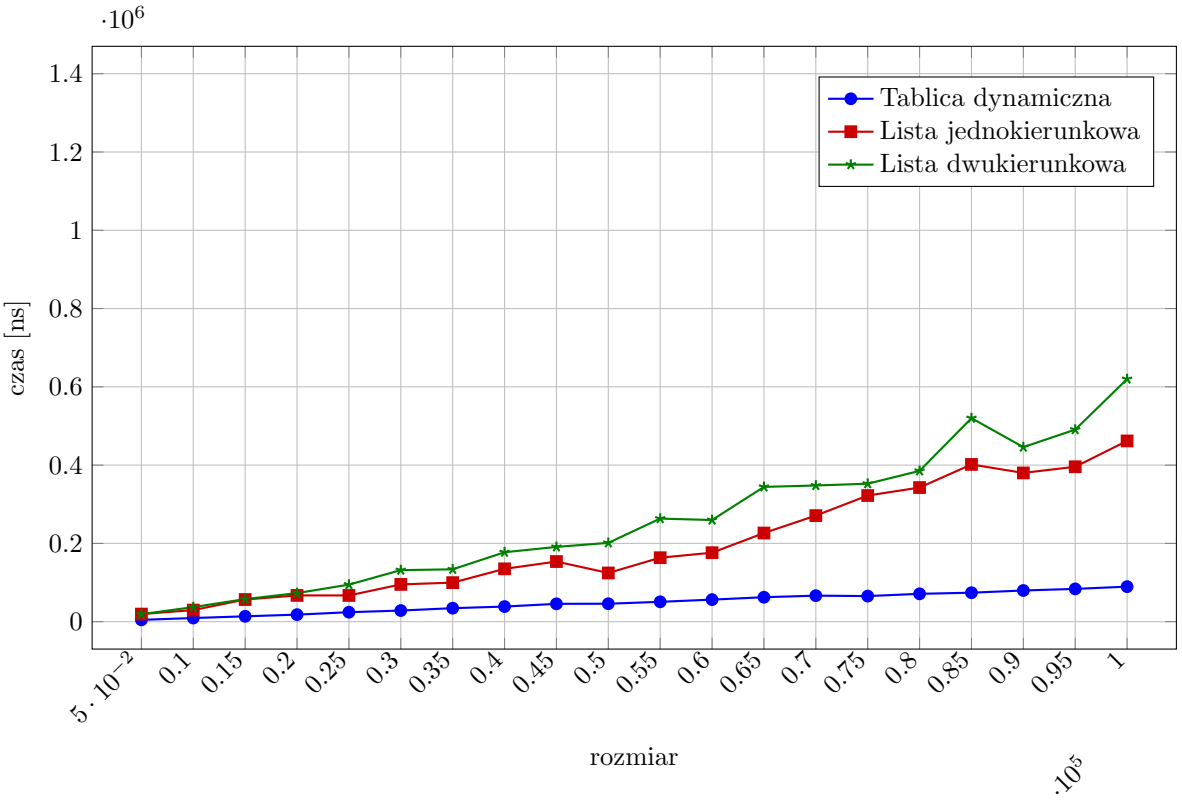
Rysunek 10: maksymalna złożoność czasowa metody `pop_back`



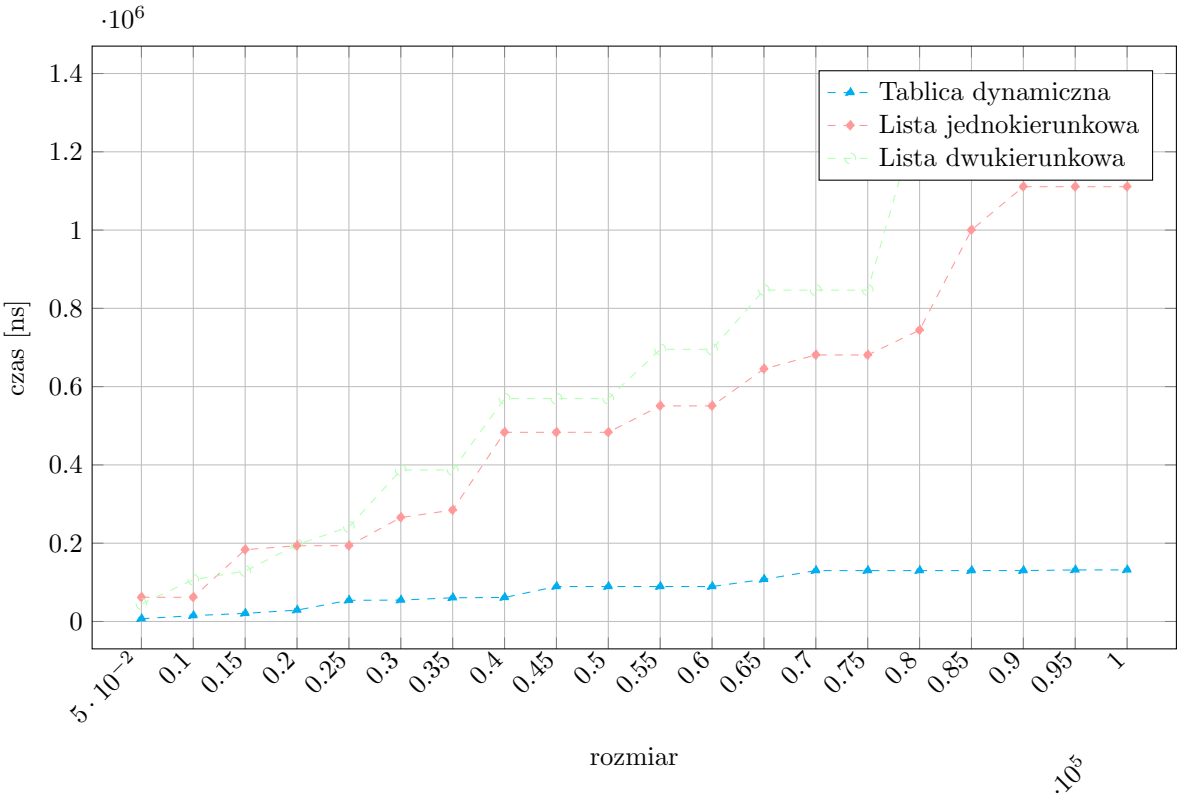
Rysunek 11: złożoność czasowa metody `remove`



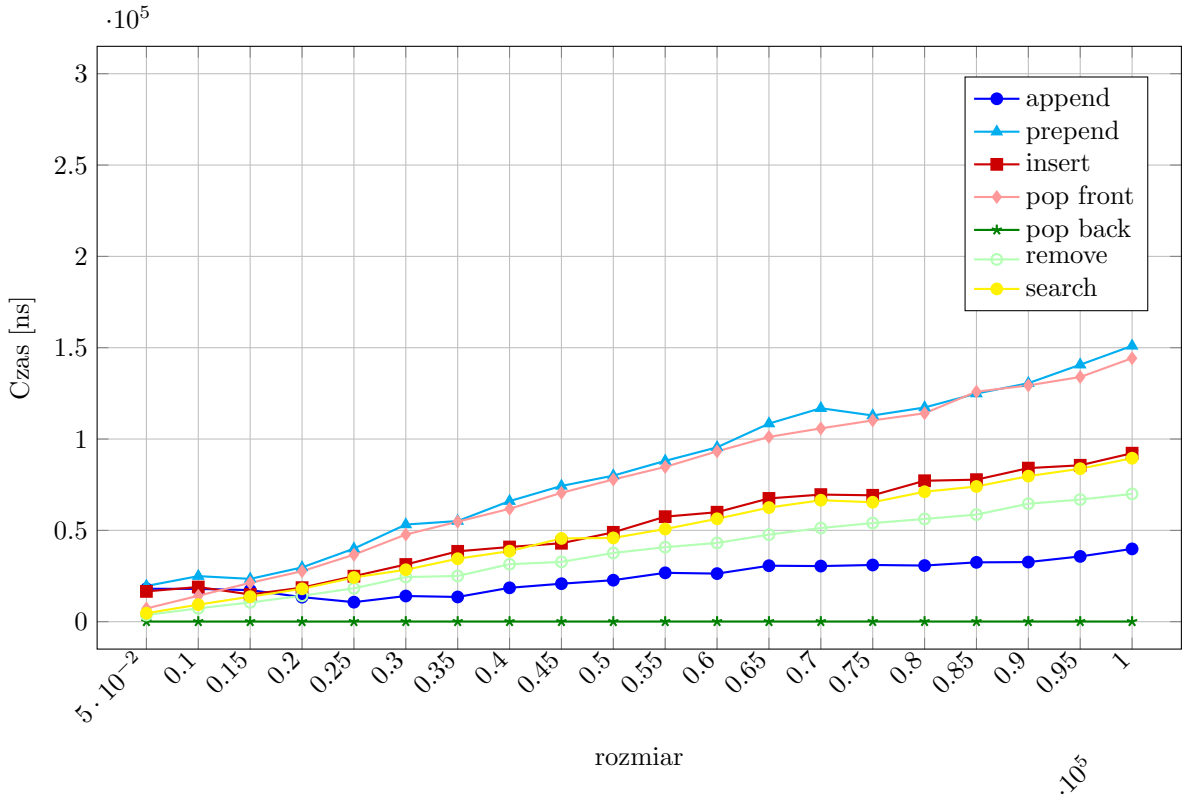
Rysunek 12: maksymalna złożoność czasowa metody `remove`



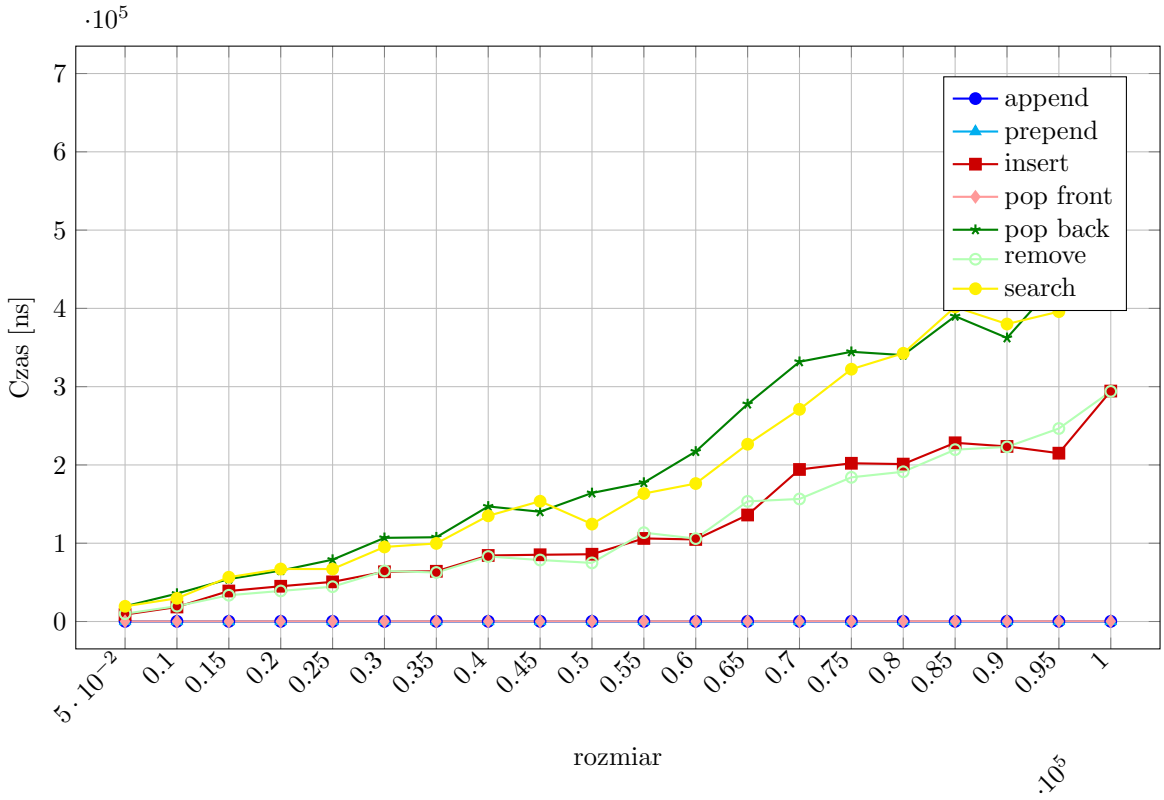
Rysunek 13: złożoność czasowa metody `search`



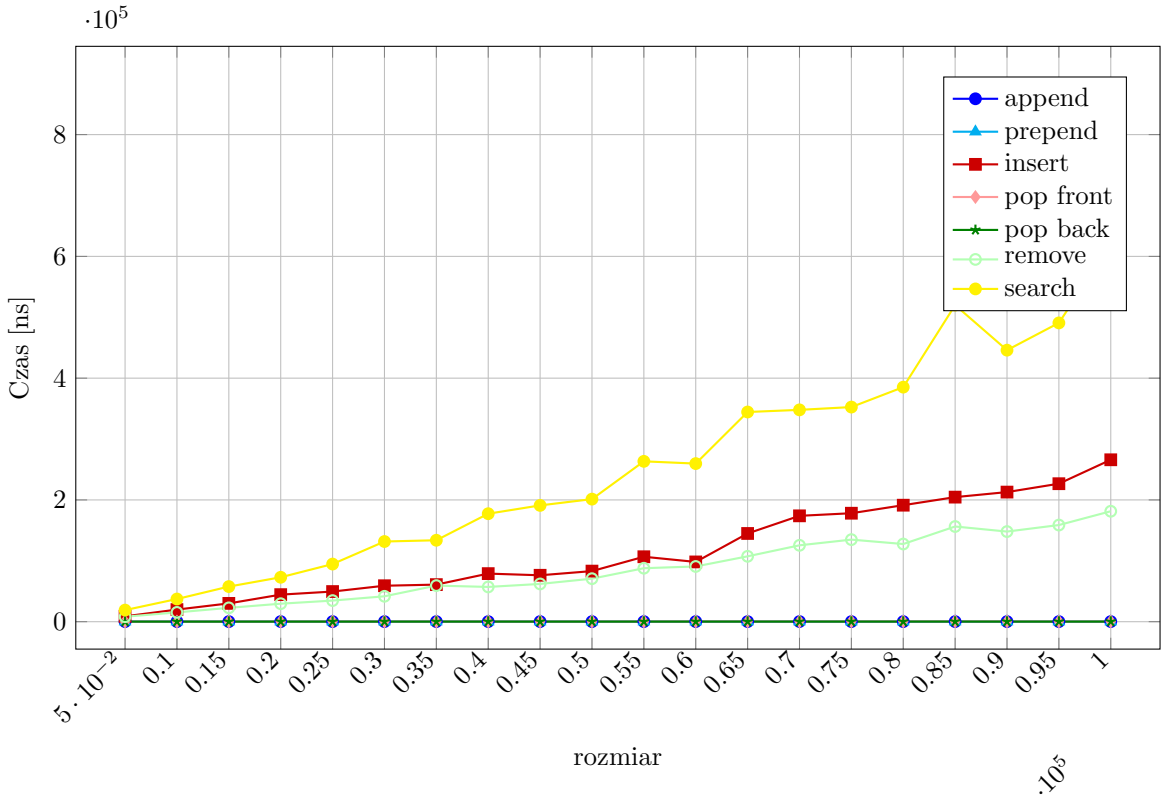
Rysunek 14: maksymalna złożoność czasowa metody `search`



Rysunek 15: Złożoność czasowa poszczególnych metod dla różnych wielkości tablicy dynamicznej



Rysunek 16: Złożoność czasowa poszczególnych metod dla różnych wielkości listy jednokierunkowej



Rysunek 17: Złożoność czasowa poszczególnych metod dla różnych wielkości listy dwukierunkowej

W ramach przeprowadzonych testów porównano wydajność trzech implementacji list: tablicy dynamicznej (**TD**), listy jednokierunkowej (**L1**) oraz listy dwukierunkowej (**L2**). Analizie poddano czasy wykonania podstawowych operacji: dodawania na koniec (*append*), dodawania na początek (*prepend*), wstawiania w środku (*insert*), usuwania z początku (*pop_front*), usuwania z końca (*pop_back*), usuwania konkretnego elementu (*remove*) oraz wyszukiwania (*search*). Pomiar obejmował średni oraz maksymalny czas wykonania dla różnych rozmiarów struktur: 5-100 tys. elementów (z krokiem 5 tys.). Warto zaznaczyć, że dla tablicy dynamicznej zasymulowano najgorszy przypadek dodawania elementu - była ona tworzona przed dodaniem w taki sposób, by wymagane było powiększenie rozmiaru tablicy ponad limit, a więc wywołanie *resize* co skutkowało większą złożonością czasową.

W operacji ***append***, struktury listowe (L1 i L2) wykazały bardzo niskie i stabilne czasy, niezależnie od rozmiaru. Tablica dynamiczna radziła sobie dobrze, choć wraz ze wzrostem rozmiaru czas dodawania nieznacznie rósł. Dla ***prepend***, TD wykazuje znaczny wzrost czasu w zależności od rozmiaru, co wiąże się z koniecznością przesuwania elementów. Listy wiązane natomiast utrzymują bardzo dobrą wydajność.

Operacja ***insert*** w środku tablicy dynamicznej miała liniowy wzrost czasu, jednak znacznie niższy niż w przypadku list wiązanych, które wymagały przejścia przez wiele elementów, co skutkowało bardzo wysokimi czasami wykonania.

W przypadku ***pop_front***, TD ponownie wypada niekorzystnie, z czasami rosnącymi nawet do ponad 140 tys. ns, co również wynika z konieczności przesuwania danych. Zarówno L1, jak i L2 zapewniły stabilne, szybkie wykonanie tej operacji.

Operacja ***pop_back*** ukazała największą przewagę tablicy dynamicznej i listy dwukierunkowej – czasy były bardzo niskie i stabilne. Lista jednokierunkowa z kolei wykazała bardzo słabe wyniki, szczególnie dla większych rozmiarów, co jest konsekwencją braku dostępu do poprzedniego elementu i konieczności iteracji.

Operacja ***remove*** była kosztowna we wszystkich strukturach, choć w listach wiązanych rosła znacznie szybciej, co może wynikać z dodatkowego zarządzania wskaźnikami. TD również wymagała przesunięć, co wpływało negatywnie na wydajność.

Wyszukiwanie (***search***) wykazało liniowy wzrost czasu wykonania w każdej strukturze. TD nieznacznie wyróżniała się lepszymi czasami przy mniejszych rozmiarach, jednak dla większych danych różnice pomiędzy strukturami zacierały się.

4 Wnioski

Analiza wyników potwierdza, że wybór struktury danych ma istotne znaczenie dla efektywności operacji. Każda z badanych struktur charakteryzuje się innymi właściwościami wydajnościowymi, co czyni je bardziej lub mniej odpowiednimi w zależności od konkretnego przypadku użycia.

Tablica dynamiczna (TD) najlepiej sprawdza się przy operacjach na końcu (*append*, *pop_back*), oferując niskie czasy wykonania. Operacje na początku (*prepend*, *pop_front*) są dla niej kosztowne ze względu na konieczność przesuwania elementów.

Lista jednokierunkowa (L1) charakteryzuje się wysoką wydajnością dla operacji związanych z początkiem struktury, ale bardzo słabo radzi sobie z usuwaniem z końca oraz wstawianiem w środku, gdzie brak możliwości cofania się po elementach znacząco pogarsza wyniki.

Lista dwukierunkowa (L2) łączy zalety obu pozostałych struktur, oferując dobrą wydajność zarówno dla operacji na początku, jak i na końcu. W większości przypadków zapewnia stabilne czasy wykonania i może być traktowana jako struktura uniwersalna, szczególnie w przypadkach wymagających częstego dostępu w obu kierunkach.

W zależności od oczekiwanego wzorca operacji, każda z przedstawionych struktur może być optymalnym wyborem. W zastosowaniach, w których dominują operacje dodawania lub usuwania na końcu, najlepszym wyborem będzie tablica dynamiczna. Dla operacji na początku – lista jednokierunkowa. W przypadku konieczności wsparcia szerokiego wachlarza operacji o względnie dobrych parametrach czasowych, najbardziej wszechstronnym rozwiązaniem pozostaje lista dwukierunkowa.