Struktury danych

Kierunek	Termin
Informatyczne Systemy Automatyki	*środa TN* $15^{15} - 16^{55}$
Imię, nazwisko, numer albumu	**Data**
Iwo Chwiszczuk 280043, Julia Wilkosz 280040	*7 maja 2025*
Link do projektu	
`https://github.com/iwonieevo/Struktury-danych`	

Kolejki priorytetowe

Spis treści

Spis tabel

Spis wykresów

1 Wstęp teoretyczny

1.1 Kolejka priorytetowa (ang. *priority queue*)

Kolejka priorytetowa to abstrakcyjny typ danych służący do przechowywania elementów, z których każdy posiada przypisany priorytet, zwykle reprezentowany w postaci klucza liczbowego. Elementy w kolejce są organizowane w taki sposób, aby możliwy był szybki dostęp do elementu o najwyższym (lub najniższym) priorytecie, w zależności od przyjętej konwencji. Każdy element kolejki stanowi parę klucz-wartość, gdzie klucz określa jego znaczenie względem pozostałych. Struktura ta umożliwia efektywne wykonywanie operacji takich jak wstawianie nowego elementu czy usuwanie elementu o najwyższym priorytecie. W przypadku istnienia wielu elementów o identycznym priorytecie, stosuje się dodatkowe strategie rozstrzygania kolejkowania, np. zasady FIFO (*first-in-first-out*) lub LIFO (*last-in-first-out*).

1.2 Implementacja za pomocą kopca (ang. *heap*)

Jedną z najczęściej stosowanych struktur do implementacji kolejki priorytetowej jest kopiec. Kopiec to nieskierowany, spójny i acykliczny graf (czyli drzewo) spełniający warunek kopca, czyli relację porządku pomiędzy węzłami — dla kopca maksymalnego każdy rodzic ma klucz większy lub równy od swoich dzieci, a dla kopca minimalnego — mniejszy lub równy. Dzięki tej strukturze możliwe jest szybkie (w czasie $O(\log n)$) wstawianie elementów oraz usuwanie elementu o najwyższym (lub najniższym) priorytecie. Kopiec najczęściej implementowany jest w formie tablicy, co umożliwia efektywny dostęp do elementów bez konieczności jawnego przechowywania wskaźników. Jedna z naszych kolejek priorytetowych jest zaimplementowana jako kopiec binarny – drzewo binarne, w którym wszystkie poziomy, z wyjątkiem ostatniego, muszą być całkowicie zapełnione i które dodatkowo spełnia własności kopca.

1.3 Implementacja za pomocą listy jednokierunkowej (ang. *single-linked list*)

Innym sposobem implementacji kolejki priorytetowej jest użycie listy jednokierunkowej. W tym podejściu elementy wstawiane są do listy w odpowiednim miejscu tak, aby utrzymać porządek według priorytetu (sortowanie przy wstawianiu). Wyszukiwanie elementu o najwyższym priorytecie możliwe jest w czasie stałym $O(1)$, jeśli lista jest posortowana malejąco według priorytetów, natomiast operacja wstawiania wymaga przejścia przez listę w celu znalezienia odpowiedniego miejsca — co w najgorszym przypadku zajmuje czas $O(n)$. Alternatywnie, można wstawiać elementy na początku listy (w czasie $O(1)$), a wyszukiwać minimum lub maksimum w czasie liniowym $O(n)$, w zależności od przyjętej strategii.

1.4 Zastosowania poszczególnych implementacji i korzyści z nich wynikające

1.4.1 Implementacja za pomocą kopca

Zastosowania:

- Algorytmy grafowe (np. Dijkstra, Prim).
- Harmonogramowanie zadań w systemach operacyjnych.
- Zarządzanie zdarzeniami w symulacjach dyskretnych.
- Kompresja danych (np. algorytm Huffmana).

Korzyści:

- Złożoność $O(\log n)$ dla operacji wstawiania i usuwania.
- Wysoka wydajność przy dużych zbiorach danych.
- Prosta implementacja w formie tablicy.

1.4.2 Implementacja za pomocą listy jednokierunkowej

Zastosowania:

- Proste systemy kolejkowania o małej liczbie elementów.
- Zadania edukacyjne i prototypowanie.
- Systemy o niskich wymaganiach czasowych (np. buforowanie komunikatów).

Korzyści:

- Bardzo prosta i szybka implementacja.
- Niewielkie zużycie pamięci przy małej liczbie elementów.
- Łatwe sterowanie obsługą elementów o równym priorytecie.

2 Założenia projektowe

W celu przeprowadzenia analizy wydajnościowej poszczególnych implementacji przyjęto następujące założenia projektowe:

2.1 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 10 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 implementacji.

2.2 Sposób generowania danych:

Dane wykorzystane w testach zostały wygenerowane za pomocą funkcji szablonowej `generate_random_integers`, która zapisuje do pliku tekstowego określoną liczbę losowych liczb całkowitych z zadanego przedziału.

Funkcja wykorzystuje następujące mechanizmy języka C++:

- `std::random_device` – dostarcza ziarno losowe (ang. *random seed*),
- `std::mt19937` – generator liczb pseudolosowych oparty na algorytmie Mersenne Twister, zapewniający wysoką jakość losowości,
- `std::uniform_int_distribution<IntegralType>` – generuje liczby całkowite o równomiernym rozkładzie w przedziale $[min, max]$. Typ `IntegralType` jest automatycznie dedukowany na podstawie argumentów `min` i `max`.

Działanie funkcji:

- Sprawdza, czy typ `IntegralType` jest całkowity (w przeciwnym razie wypisuje błąd i kończy działanie),
- Otwiera plik o ścieżce `dest_path` (w przypadku niepowodzenia wypisuje błąd),
- Generuje `n_rows` liczb losowych i zapisuje je w osobnych wierszach pliku wyjściowego.

W trakcie działania programu, tworzono dwa pliki wykorzystując tę funkcję - `temp_keys`, w którym zapisywano klucze oraz `temp_values`, w którym zapisywano wartości. Następnie dane z tych plików były odpowiednio wczytywane do zaimplementowanych struktur.

2.3 Sposób pomiaru czasu:

Czas wykonania operacji na strukturach danych mierzony jest za pomocą uniwersalnej funkcji szablonowej `measure_time`, która wykorzystuje mechanizmy biblioteki `<chrono>` do precyzyjnego pomiaru w nanosekundach.

Działanie funkcji:

- Przyjmuje trzy argumenty:
  - wskaźnik na obiekt (`object`),
  - wskaźnik na metodę klasy (`method`),
  - parametry metody (`params...`).
- Mierzy czas wykonania metody za pomocą `std::chrono::steady_clock`, który gwarantuje monotoniczność pomiaru (odporność na zmiany czasu systemowego).
- Wynik zwracany jest w nanosekundach jako `unsigned long long`.

Zastosowanie: Funkcja jest zaprojektowana jako narzędzie ogólnego przeznaczenia (ang. *generic*), dzięki czemu może mierzyć czas dowolnej metody dowolnej klasy. Wykorzystuje:

- `std::invoke` do wywołania metody z przekazanymi parametrami,
- perfect forwarding (`std::forward`) dla efektywnego przekazania argumentów.

2.4 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.

2.5 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

W badaniach porównano wydajność następujących operacji na kolejkach priorytetowych:

- `insert`:
  - Wariant A: wstawienie elementu o **nowym maksymalnym** priorytecie (większym niż dotychczasowe maksimum) (`insert_highest`)
  - Wariant B: wstawienie elementu o **minimalnym** priorytecie (równym dotychczasowemu minimum) (`insert_lowest`)
- `extract_max`: usunięcie i zwrócenie elementu o maksymalnym priorytecie
- `find_max`: zwrócenie wartości elementu o maksymalnym priorytecie (bez usuwania)
- `modify_key`:
  - Wariant A: zmiana priorytetu elementu z **maksymalnego na minimalny** (`decrease_key`)
  - Wariant B: zmiana priorytetu elementu z **minimalnego na większy niż dotychczasowy maksymalny** (`increase_key`)
- `return_size`: zwrócenie liczby elementów w kolejce

Dla każdej operacji przygotowano:

- Tabelę porównawczą czasów wykonania (w nanosekundach) w zależności od:
  - rozmiaru danych,
  - implementacji kolejki.
- Dwa typy wykresów:
  - W skali liniowej (dla bezpośredniej interpretacji wyników),
  - W skali logarytmicznej ($\log_{10}$ na obu osiach) – uwydatniający różnice przy dużych rozmiarach danych.

Tabela 1: Czas wykonania `insert_highest` [ns]

SIZE	BinaryHeap	LinkedList
5000	460	340
10000	410	380
15000	450	480
20000	460	400
25000	480	360
30000	490	430
35000	590	360
40000	630	3790
45000	670	400
50000	580	410
55000	630	410
60000	510	390
65000	580	400
70000	680	480
75000	770	350
80000	760	440
85000	670	380
90000	770	340
95000	670	400
100000	700	420

Tabela 2: Czas wykonania `insert_lowest` [ns]

SIZE	BinaryHeap	LinkedList
5000	240	9530
10000	230	23750
15000	240	47550
20000	210	52970
25000	310	86850
30000	240	100150
35000	230	134340
40000	230	143860
45000	270	144810
50000	220	168880
55000	270	262720
60000	210	228270
65000	200	264330
70000	270	318880
75000	390	328460
80000	340	315980
85000	250	310540
90000	270	277950
95000	230	461530
100000	270	419950

Tabela 3: Czas wykonania `extract_max` [ns]

SIZE	BinaryHeap	LinkedList
5000	360	280
10000	370	290
15000	390	300
20000	370	420
25000	450	310
30000	450	340
35000	580	410
40000	480	400
45000	540	410
50000	540	410
55000	660	370
60000	600	400
65000	510	410
70000	920	380
75000	820	480
80000	800	410
85000	940	410
90000	650	460
95000	1030	360
100000	900	400

Tabela 4: Czas wykonania `find_max` [ns]

SIZE	BinaryHeap	LinkedList
5000	200	210
10000	160	220
15000	240	260
20000	180	260
25000	210	270
30000	220	250
35000	260	260
40000	210	290
45000	240	240
50000	230	230
55000	260	260
60000	240	290
65000	250	320
70000	270	250
75000	270	280
80000	240	330
85000	240	320
90000	300	310
95000	300	330
100000	280	330

Tabela 5: Czas wykonania `decrease_key` [ns]

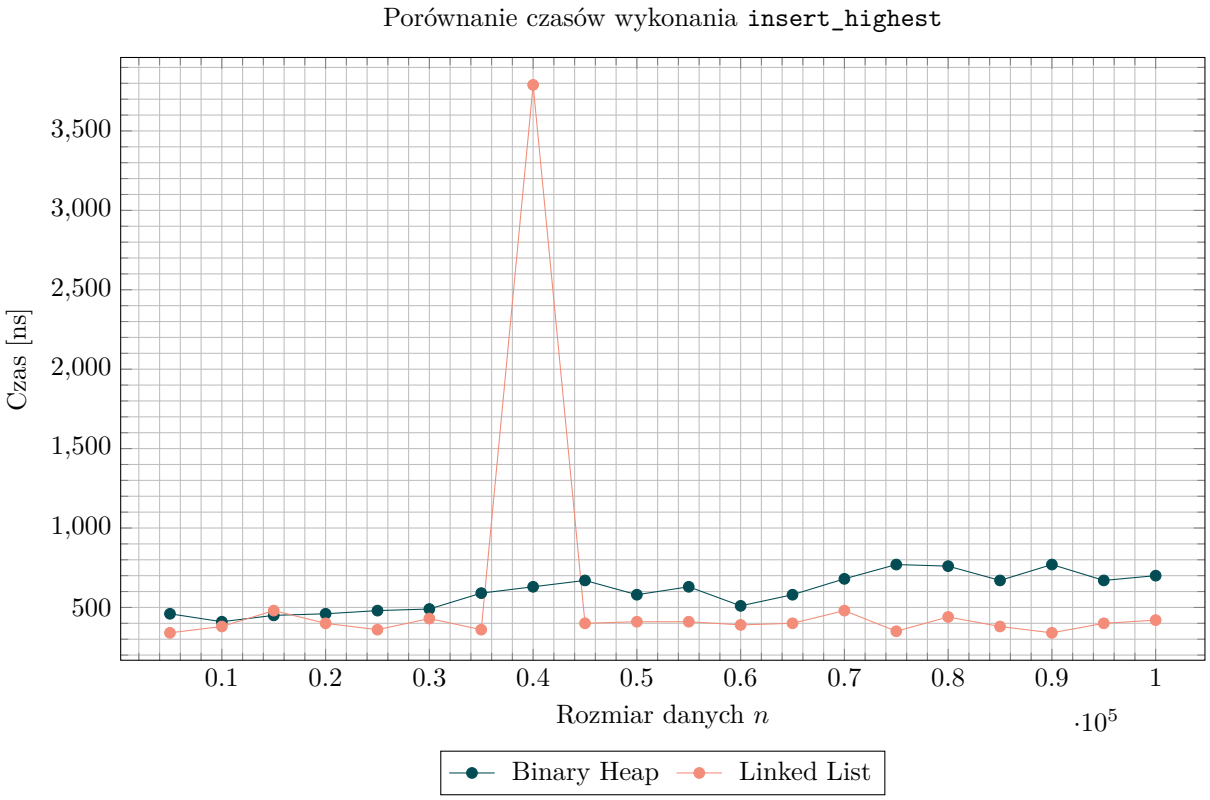
SIZE	BinaryHeap	LinkedList
5000	280	9270
10000	300	21030
15000	380	45170
20000	370	67990
25000	420	86810
30000	370	96010
35000	390	112600
40000	370	150020
45000	410	122220
50000	560	135680
55000	410	207640
60000	550	228950
65000	370	224090
70000	710	270830
75000	540	287520
80000	590	231980
85000	550	221770
90000	580	265000
95000	610	386940
100000	720	418090

Tabela 6: Czas wykonania `increase_key` [ns]

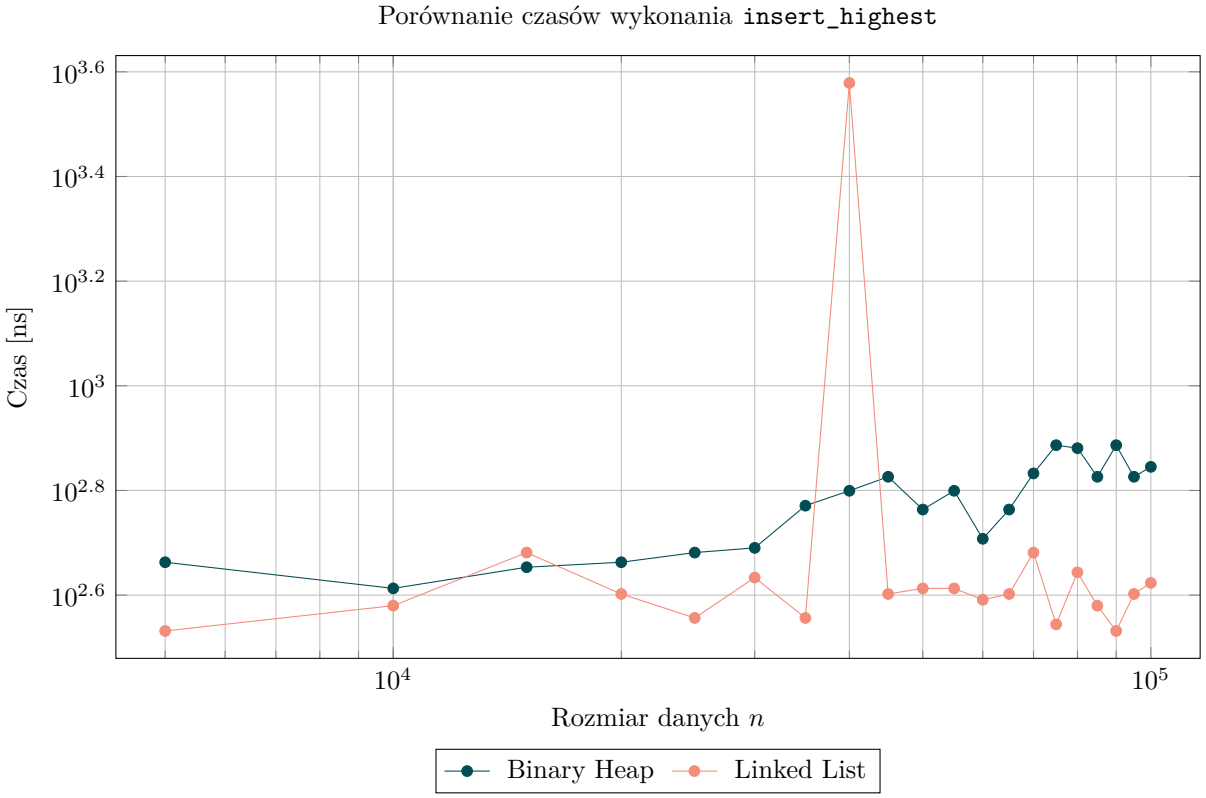
SIZE	BinaryHeap	LinkedList
5000	5070	9500
10000	9120	25840
15000	13520	43920
20000	22020	52030
25000	26970	95080
30000	26490	98190
35000	33040	125170
40000	37010	136250
45000	42180	175680
50000	49850	140620
55000	53030	155920
60000	56720	208430
65000	59800	241510
70000	74200	270600
75000	70820	363700
80000	76390	335070
85000	83570	242390
90000	85880	366960
95000	85990	491580
100000	103290	446450

Tabela 7: Czas wykonania `return_size` [ns]

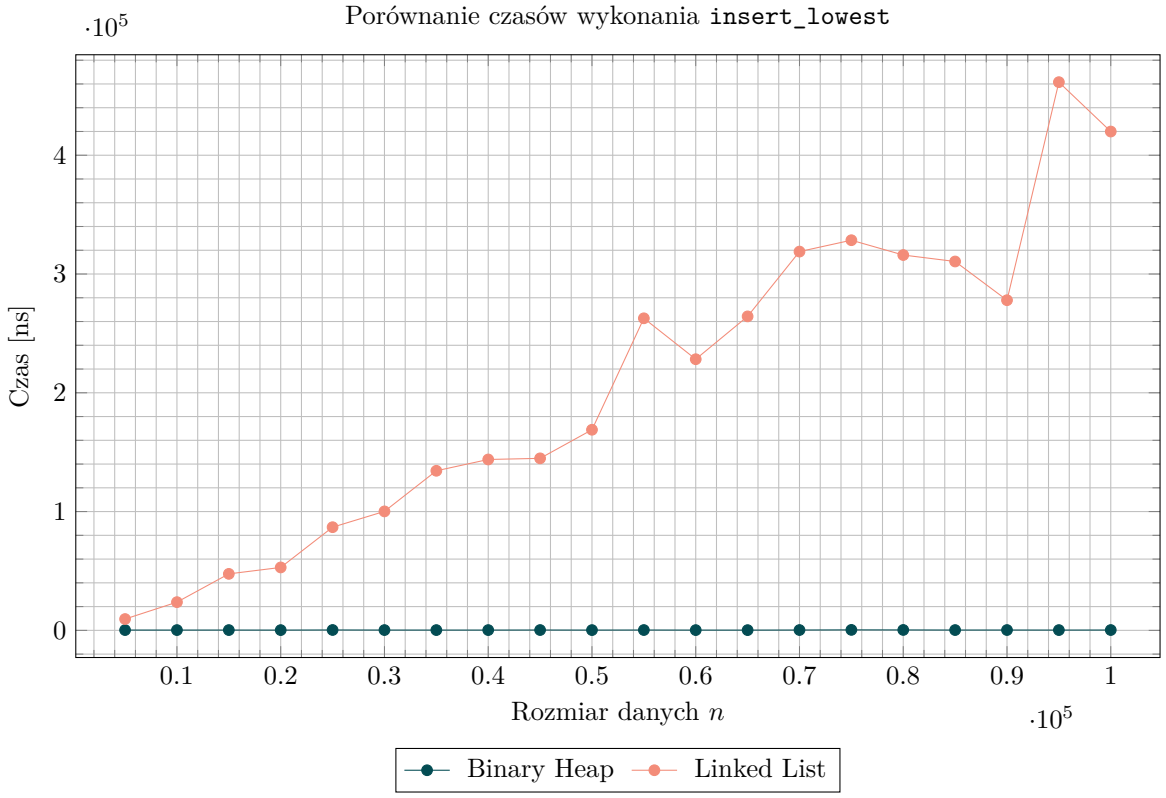
SIZE	BinaryHeap	LinkedList
5000	80	90
10000	70	110
15000	40	140
20000	120	120
25000	110	140
30000	100	130
35000	110	180
40000	100	150
45000	110	160
50000	120	120
55000	140	160
60000	130	130
65000	120	1570
70000	220	250
75000	140	230
80000	160	170
85000	210	180
90000	130	160
95000	160	170
100000	110	160



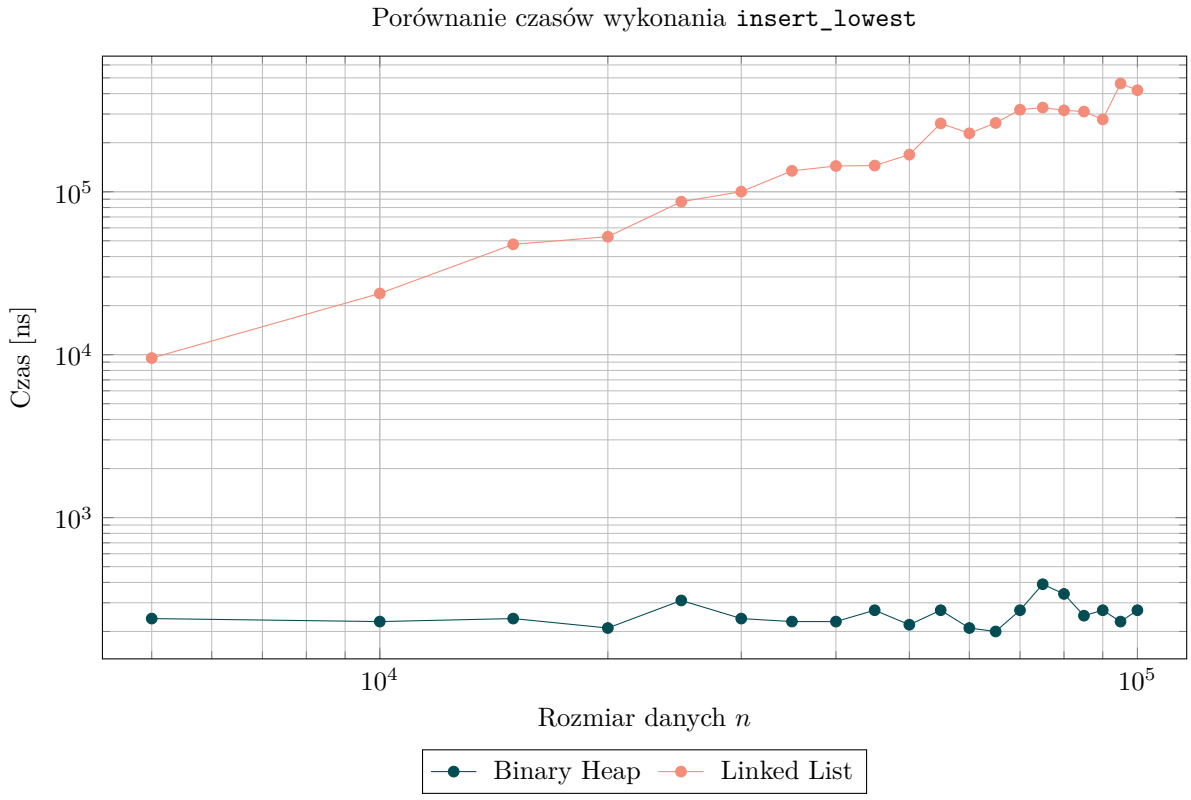
Wykres 1: `insert_highest`



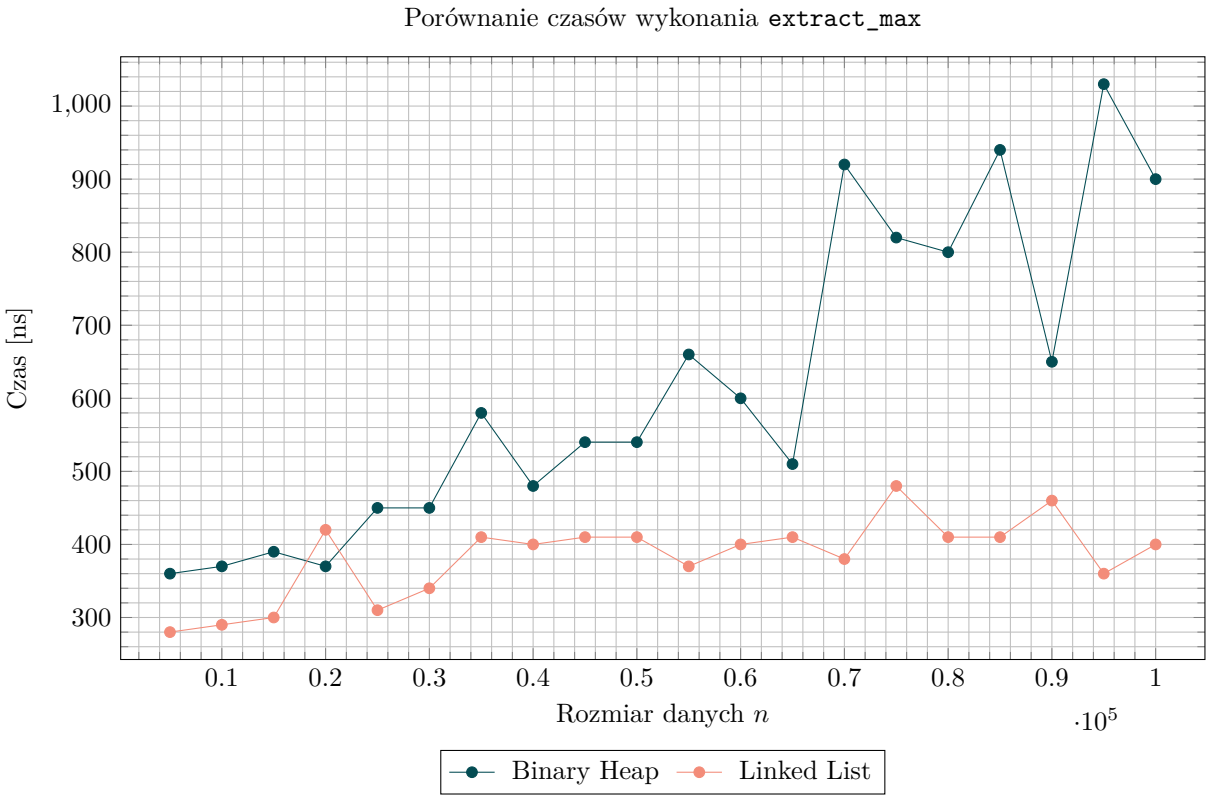
Wykres 2: `insert_highest` (w skali $\log x$-$\log y$)



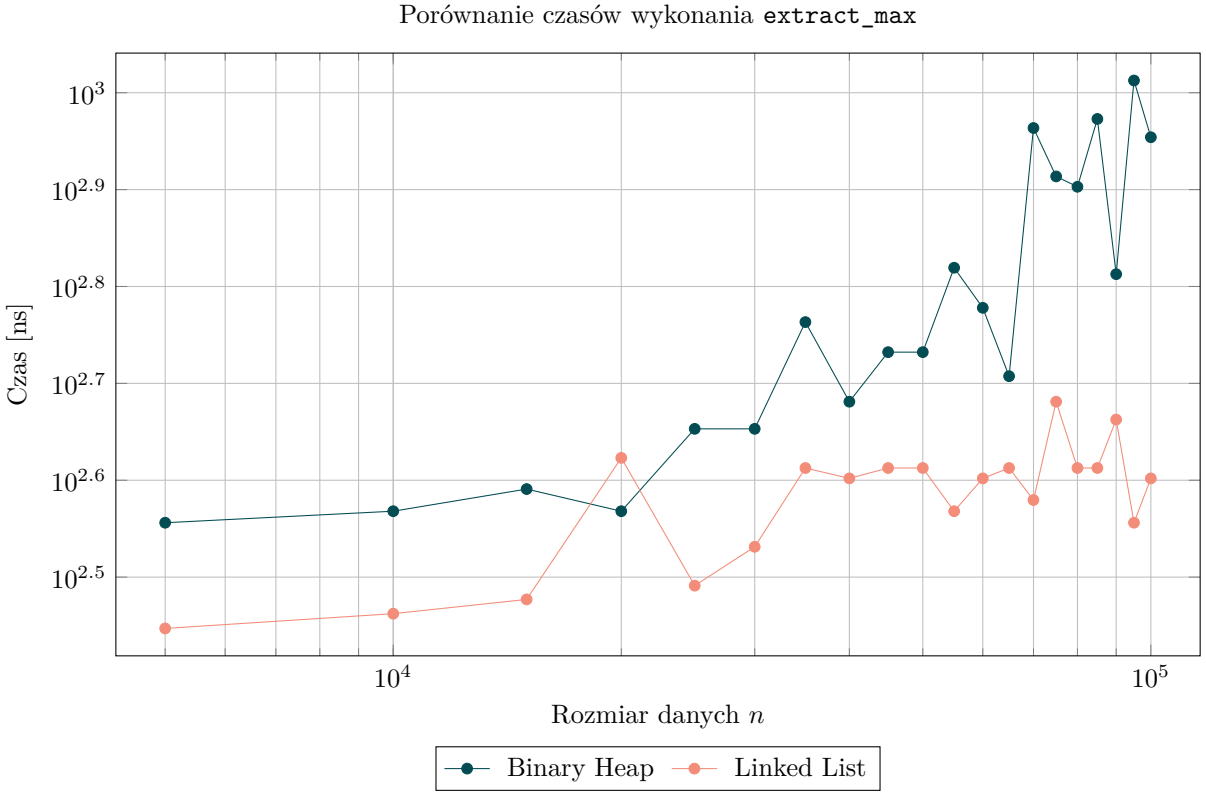
Wykres 3: `insert_lowest`



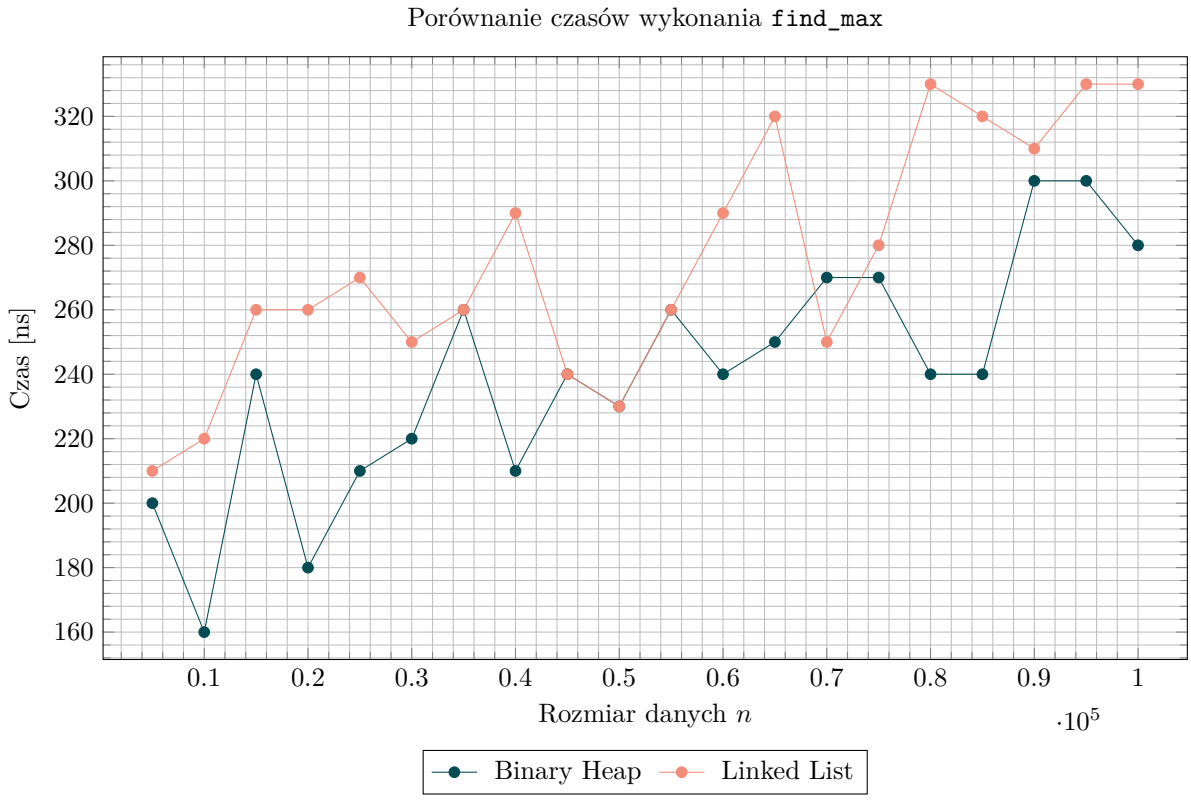
Wykres 4: `insert_lowest` (w skali $\log x$-$\log y$)



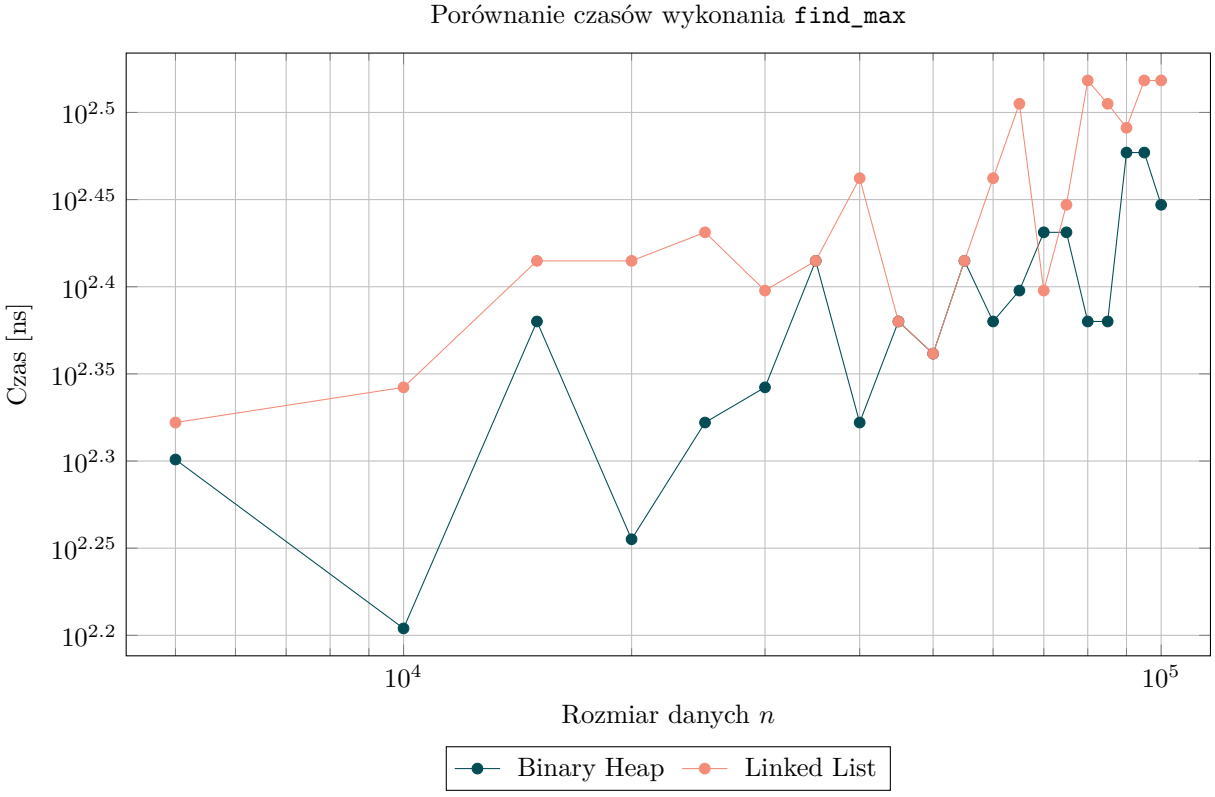
Wykres 5: `extract_max`



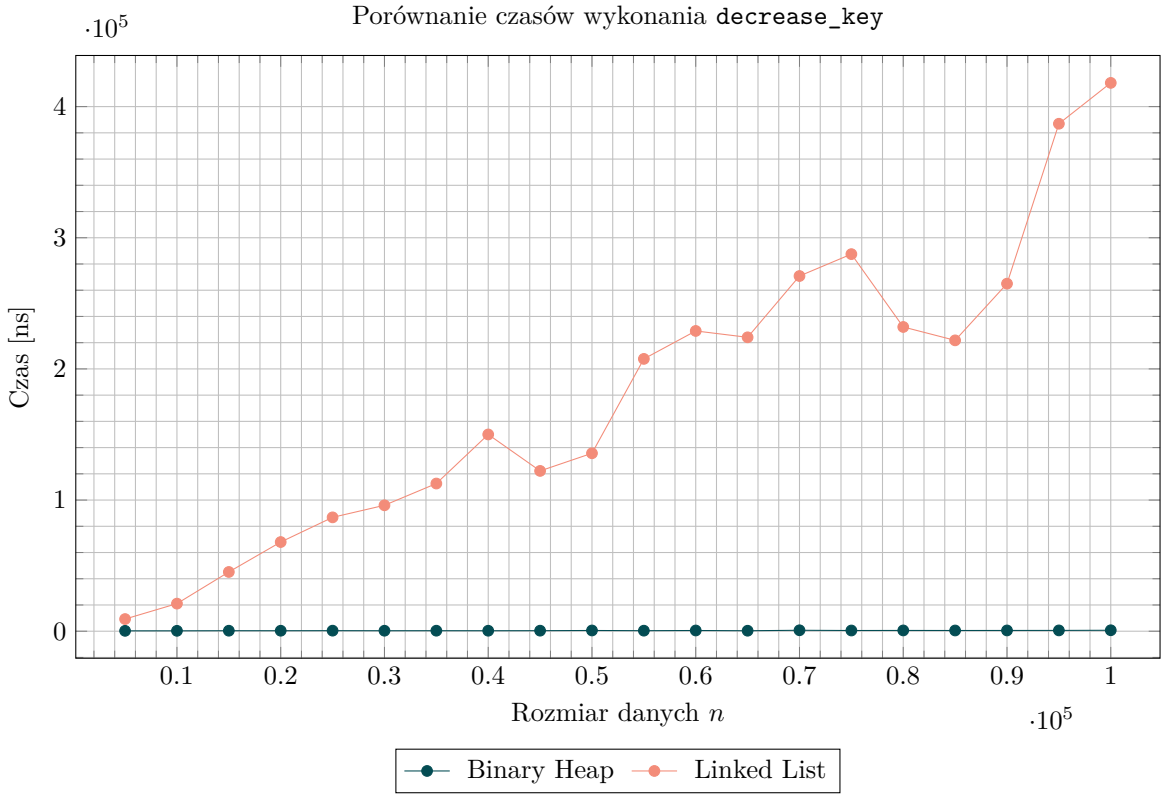
Wykres 6: `extract_max` (w skali $\log x$-$\log y$)



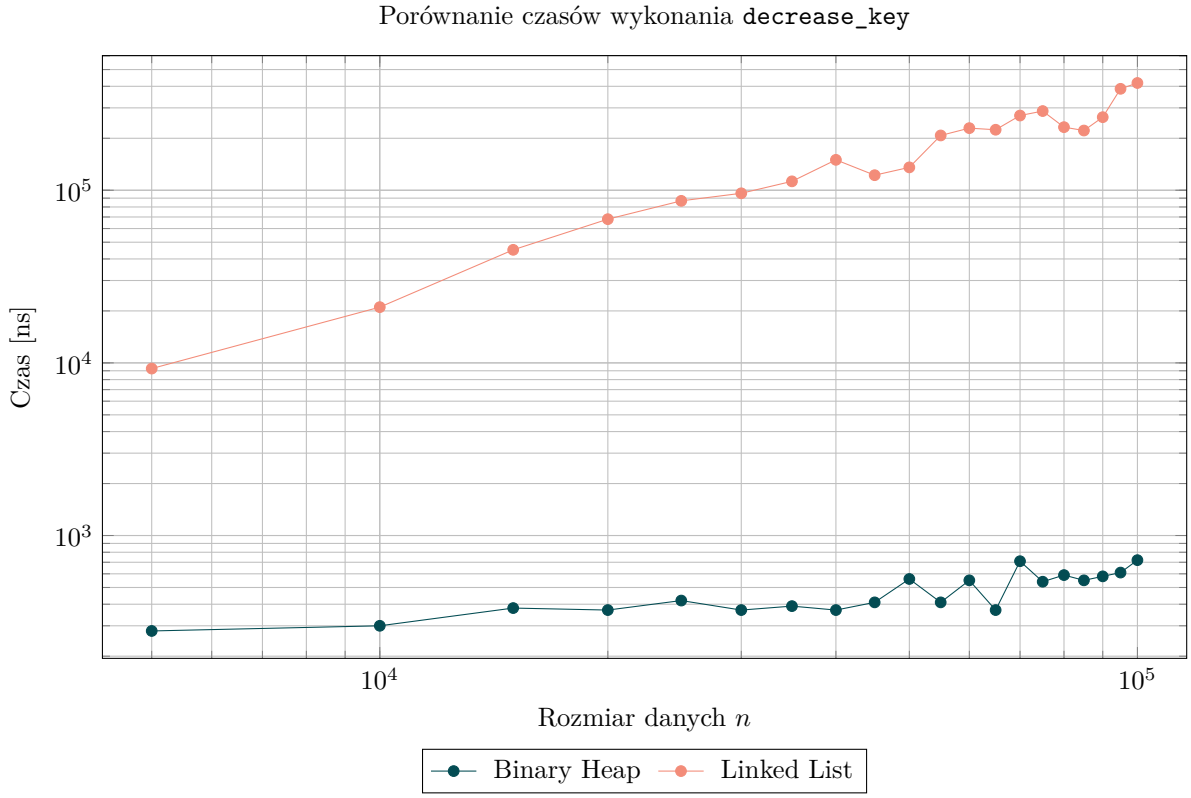
Wykres 7: `find_max`



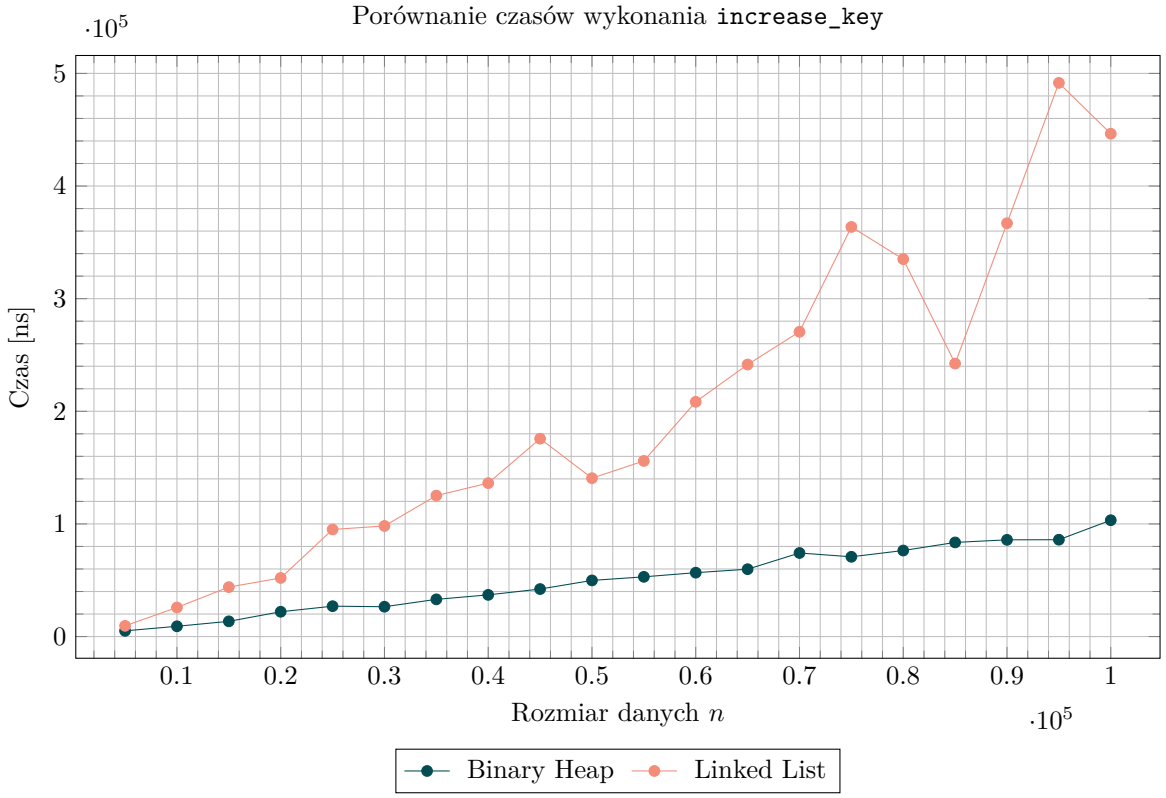
Wykres 8: `find_max` (w skali $\log x$-$\log y$)



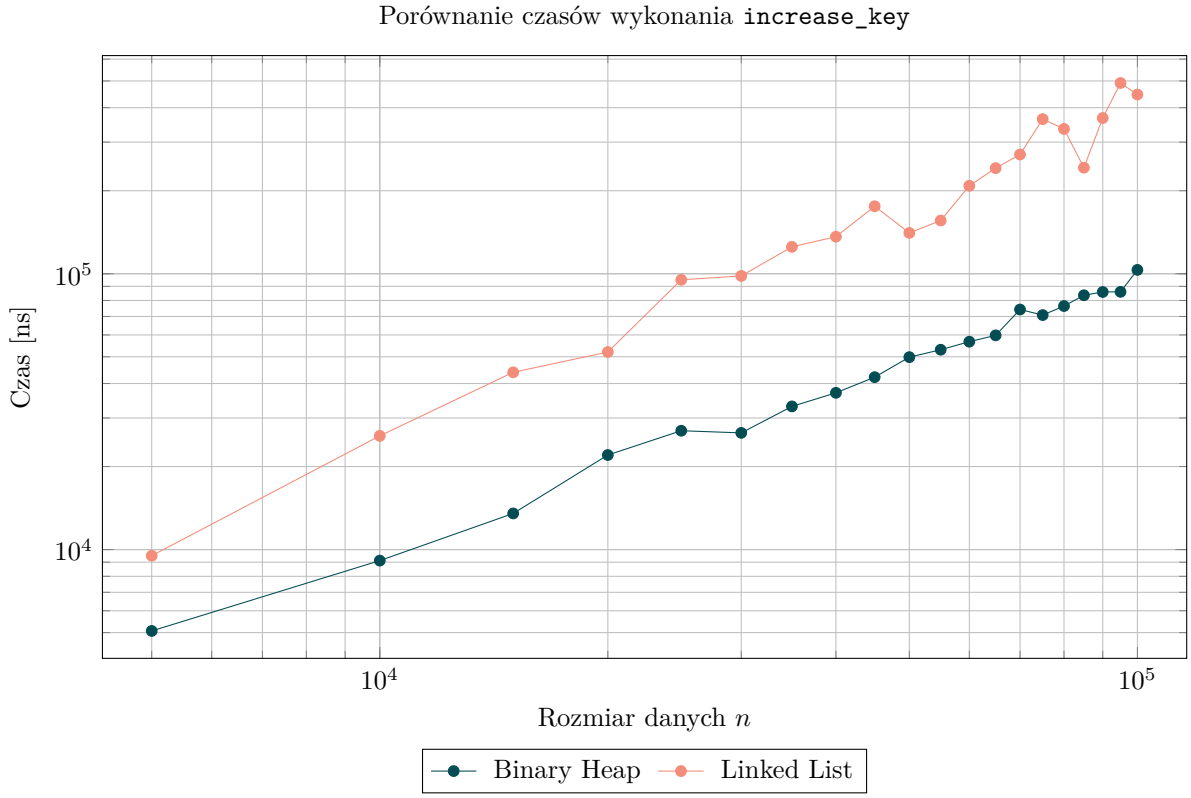
Wykres 9: `decrease_key`



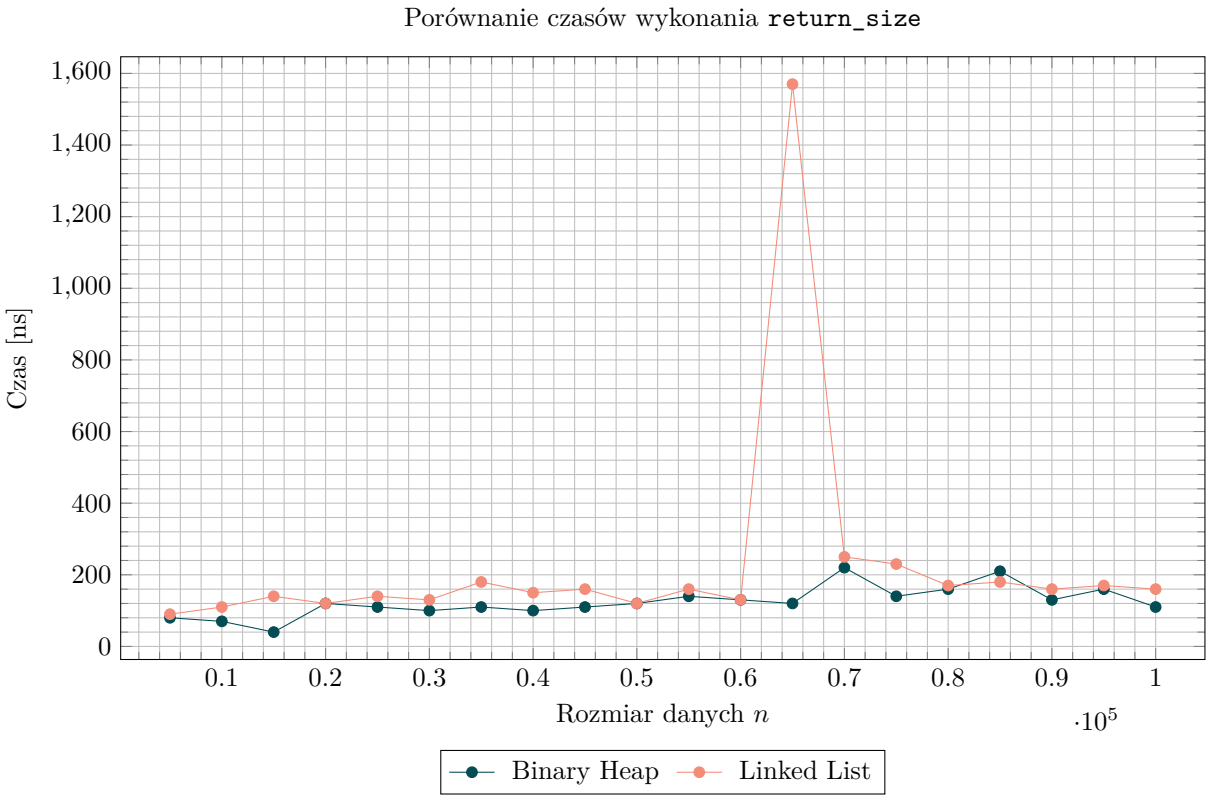
Wykres 10: `decrease_key` (w skali $\log x$-$\log y$)



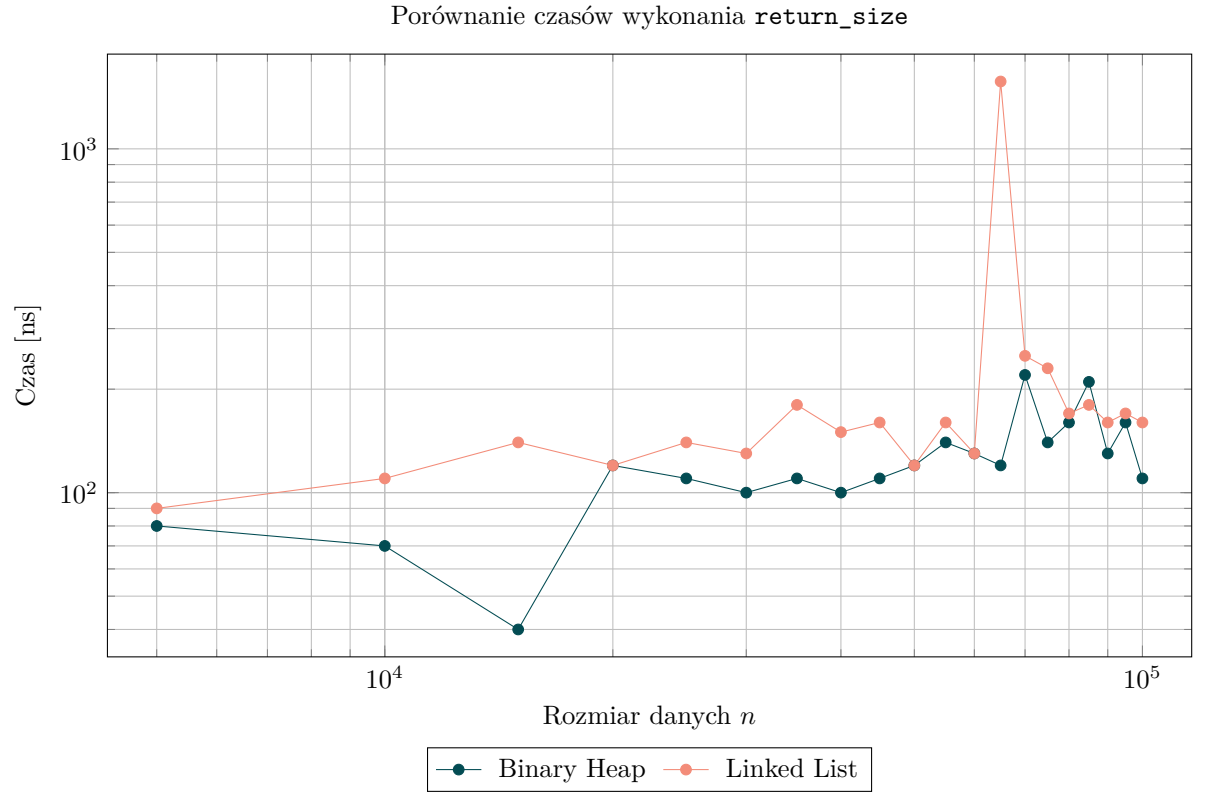
Wykres 11: `increase_key`



Wykres 12: `increase_key` (w skali $\log x$-$\log y$)



Wykres 13: `return_size`



Wykres 14: `return_size` (w skali $\log x$-$\log y$)

4 Analiza wyników

Na podstawie przeprowadzonych pomiarów można wyciągnąć następujące wnioski dotyczące porównania implementacji kolejki priorytetowej przy użyciu kopca binarnego (`BinaryHeap`) oraz listy wiązanej (`LinkedList`):

- **Operacje podstawowe**
  - `return_size`:
    - `BinaryHeap`: Stały czas wykonania 40-220 ns ($O(1)$ teoretycznie i praktycznie)
    - `LinkedList`: Zazwyczaj stały czas 90-180 ns, ale z anomaliami do 1570 ns (teoretycznie $O(1)$, anomalie sugerują problemy z alokacją pamięci)
  - `find_max`:
    - `BinaryHeap`: 160-300 ns ($O(1)$ - dostęp do korzenia)
    - `LinkedList`: 210-330 ns ($O(1)$ - dostęp do głowy listy)
- **Operacje wstawiania `insert`**
  - **Wstawianie minimum `insert_lowest`**:
    - `BinaryHeap`: Stabilny czas 200-390 ns ($O(\log n)$ - zgodne z teorią)
    - `LinkedList`: Czas rośnie liniowo od 9530 ns do 419950 ns ($O(n)$ - konieczność przejścia całej listy)
  - **Wstawianie maximum `insert_highest`**:
    - `BinaryHeap`: 410-770 ns ($O(\log n)$ - naprawa kopca w górę)
    - `LinkedList`: 340-430 ns ($O(1)$ - wstawienie na początek) + anomalia 3790 ns przy n=40000)
- **Operacje zmiany priorytetu `modify_key`**
  - **Zwiększenie priorytetu `increase_key`**:
    - `BinaryHeap`: 5070-103290 ns ($O(n)$ dla wyszukania + $O(\log n)$ dla naprawy)
    - `LinkedList`: 9500-491580 ns ($O(n)$ - wyszukanie + usunięcie + wstawienie)
  - **Zmniejszenie priorytetu `decrease_key`**:
    - `BinaryHeap`: 280-720 ns ($O(n)$ dla wyszukania + $O(\log n)$ dla naprawy)
    - `LinkedList`: 9270-418090 ns ($O(n)$ - pełne przeszukanie listy)
- **Operacja `extract_max`**
  - `BinaryHeap`: 360-1030 ns ($O(\log n)$ - zgodne z teorią)
  - `LinkedList`: 280-480 ns ($O(1)$ - usunięcie głowy listy)

5 Podsumowanie złożoności

Tabela 8: Porównanie złożoności teoretycznych i praktycznych

Operacja	**BinaryHeap**		**LinkedList**	
	Teoria	Praktyka	Teoria	Praktyka
`insert_lowest`	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
`insert_highest`	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
`increase_key`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`decrease_key`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`find_max`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`extract_max`	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$

6 Wnioski

- Kopiec binarny przeważa dla operacji modyfikujących strukturę przy dużych rozmiarach danych
- Lista wiązana może być lepsza dla specjalizowanych przypadków (częste `extract_max` + rzadkie modyfikacje)
- Anomalie w LinkedList sugerują problemy z zarządzaniem pamięcią lub lokalnością pamięci
- Implementacja BinaryHeap mogłaby mieć teorytyczną złożoność $O(\log n)$ dla `modify_key` przy zastosowaniu hashmapy, w której mapowane byłyby indeksy elementów

Źródła

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms* (3rd Edition), MIT Press, 2009

[2] Robert Sedgewick, Kevin Wayne, *Algorithms* (4th Edition), Addison-Wesley Professional, 2011

[3] Wikipedia, *Priority queue*, dostęp: 7 maja 2025, `https://en.wikipedia.org/wiki/Priority_queue`

[4] Wikipedia, *Binary heap*, dostęp: 7 maja 2025, `https://en.wikipedia.org/wiki/Binary_heap`