Struktury danych	
Kierunek *Informatyczne Systemy Automatyki*	**Termin** *środa TP* $15^{15} - 16^{55}$
Imię, nazwisko, numer albumu *Piotr Brajer 272538*	**Data** *01.05.2024*
Link do Projektu https://github.com/ElemkayZ/strukturyDanych/tree/main/projekt2	

Sprawozdanie – Projekt 2

1) Kopiec Binarny – Binary Heap

Kopiec binarny pozwala na skuteczne operacje usuwania i dodawania elementów o najwyższym priorytecie. Operacje działają na złożoności czasowej O(log n) przez co operacje są wykonywane szybko na dużych ilościach danych.
Kolejka priorytetowa została zaprezentowana na podstawie tablicy w której utrzymywane są właściwości kopca.
Podczas dodawania element jest wstawiony na koniec tablicy a następnie przesuwany w górę wykonując operacje heapify up utrzymując własności kopca. Dla usuwania element o najwyższym priorytecie który jest na początku tablicy jest usuwany i na jego miejsce wstawiany jest ostatni element na którym wykonywana jest operacja heapify down dzięki czemu ustrzymywana jest własność kopca.

2) Sortowana Tablica – Sorted Array

Tablice stanowi solidną podstawę do budowania kolejek priorytetowych w strukturach danych. Organizuje elementy w oparciu o swoje priorytety, w kolejności od najwyższej do najniższej.
Pozycja każdego elementu w tablicy odpowiada jego poziomowi priorytetu, umożliwiając szybki dostęp do elementów o wysokim lub niskim priorytecie. Dodawanie elementów polega na umieszczeniu ich w tablicy zgodnie z ich priorytetem i ewentualnym ponownym dostosowaniu w celu zachowania kolejności.
Usuwanie elementów ma na celu element o najwyższym priorytecie, znajdujący się na początku tablicy. Tablica oferuje szybki dostęp do elementów za pomocą jej indeksu. Jednak przez stały rozmiar może wymagać dostosowania w miarę zmiany kolejki, co wpływa na wydajność i wykorzystanie pamięci. Chociaż tablica skutecznie radzą sobie z dostępem opartym na priorytetach.

3) Średnia Złożoności obliczeniowe struktur:

	OrderedArray	BinaryHeap
Insert	O(n)	O(log n)
Extract	O(n)	O(log n)
Peek	O(1)	O(1)
Modify	O(n)	O(log n)
Size	O(1)	O(1)

4) Założenia projektowe:

a) Wielkość struktur:
   i) Każda struktura została badana na: 30 000, 50 000, 60 000, 70 000, 100 000, 150 000, 200 000, 300 000 ilości elementów

b) Sposób generowania elementów do struktur:
   i) Do wygenerowania pseudo przypadkowych liczb wykorzystałem mt19937, na przedziałach 0-10 000 dla int przechowywanych wewnątrz elementów struktur, a dla priorytetów na przedziale 0-10 000 000

```
mt19937 rng(time(0));
std::uniform_int_distribution<unsigned long> rData(0,10000);
std::uniform_int_distribution<unsigned long> rPriority(0,10000000);
```

```
orderedArray orderedArray;
binaryHeap binaryHeap;
int dataBaseSize = 80000;//number of random elements added to DBs
for (int i = 0; i < dataBaseSize ; ++i) {
    orderedArray.insert(rData(rng), rPriority(rng));
    binaryHeap.insert(rData(rng), rPriority(rng));

}
```

c) Sposób Pomiaru czasu funkcji struktury:
   i) Wygenerowano 10 przypadkowo generowanych zbiorów danych dla każdej struktury
   ii) Każda funkcja została mierzona pojedynczo 20 razy
   iii) W sumie każdą funkcje mierzono 200 razy a średnia pomiarów została podana w tabelach i wykresach

```
for (int i = 0; i < 20; i++)
{
    auto start = std::chrono::high_resolution_clock::now();
    orderedArray.insert(rData(rng), rPriority(rng));
    auto stop = std::chrono::high_resolution_clock::now();
    d = stop - start;
    OAavgInsert += d.count();
}
```

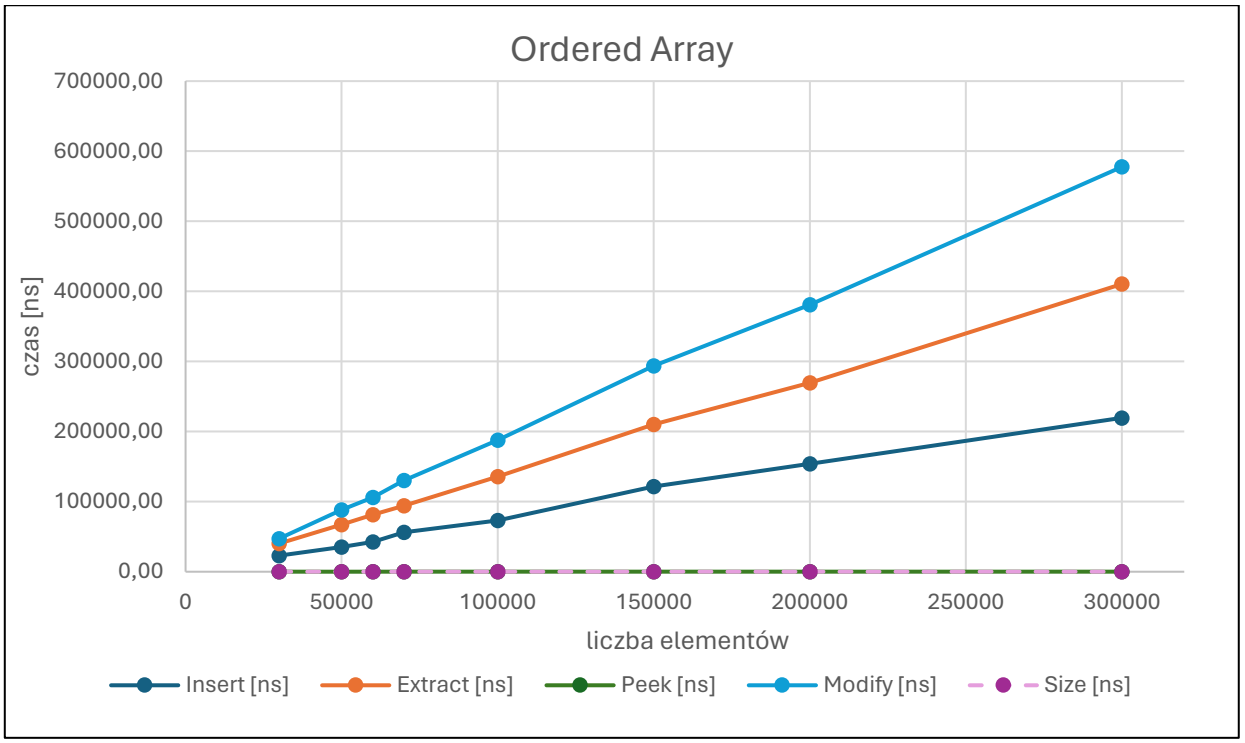
5) Badania:

Tabela 1 : średnie pomiary czasu wykonywania funkcji

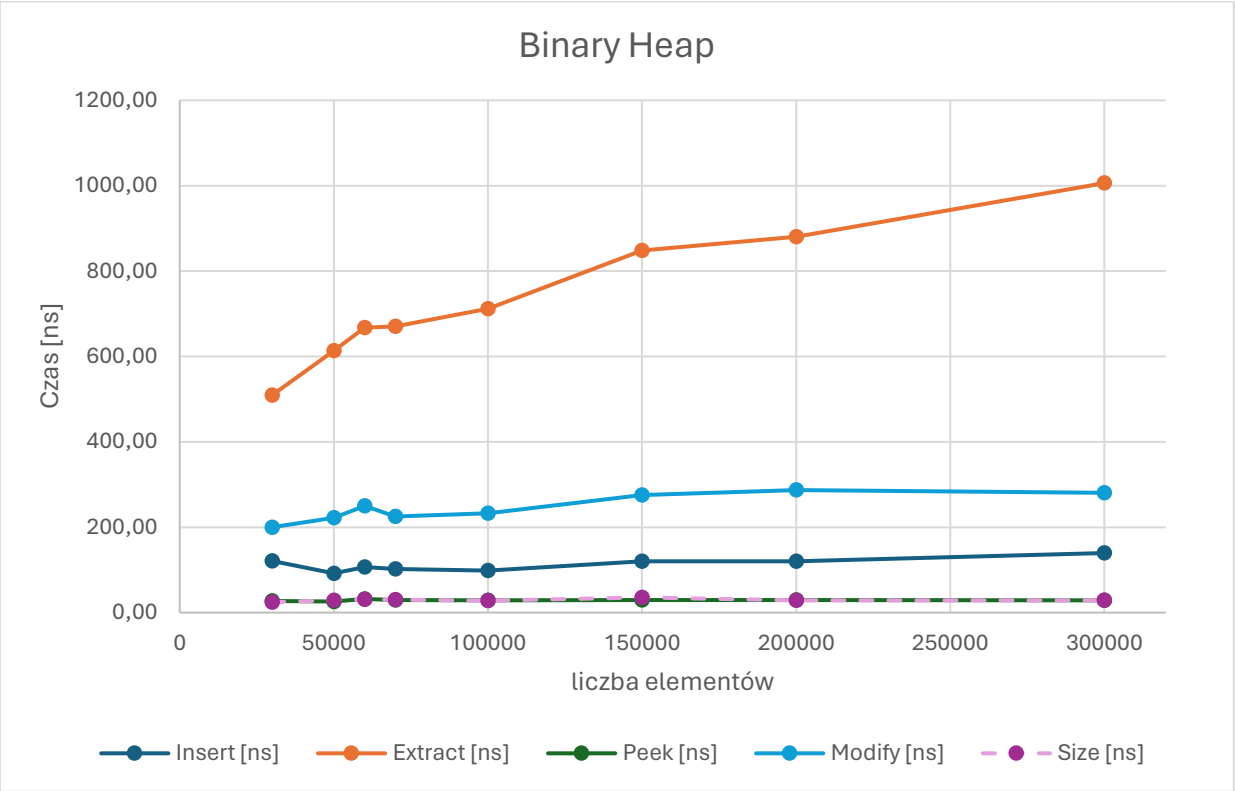
	ilość elementów							
	30000		50000		60000		70000	
DBs ->	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap
Insert [ns]	22988,5	121	35438,5	92	42655,5	107	56372	102
Extract [ns]	40230	509,5	67250,5	613,5	81154,5	667,5	94330,5	670,5
Peek [ns]	29,5	27,5	26	26	34	32	30	29,5
Modify [ns]	47310,5	200	88267	222,5	105966	249,5	130292	225
Size [ns]	28	24,5	28,5	29	28	31	29,5	30

Tabela 2 : średnie pomiary czasu wykonywania funkcji

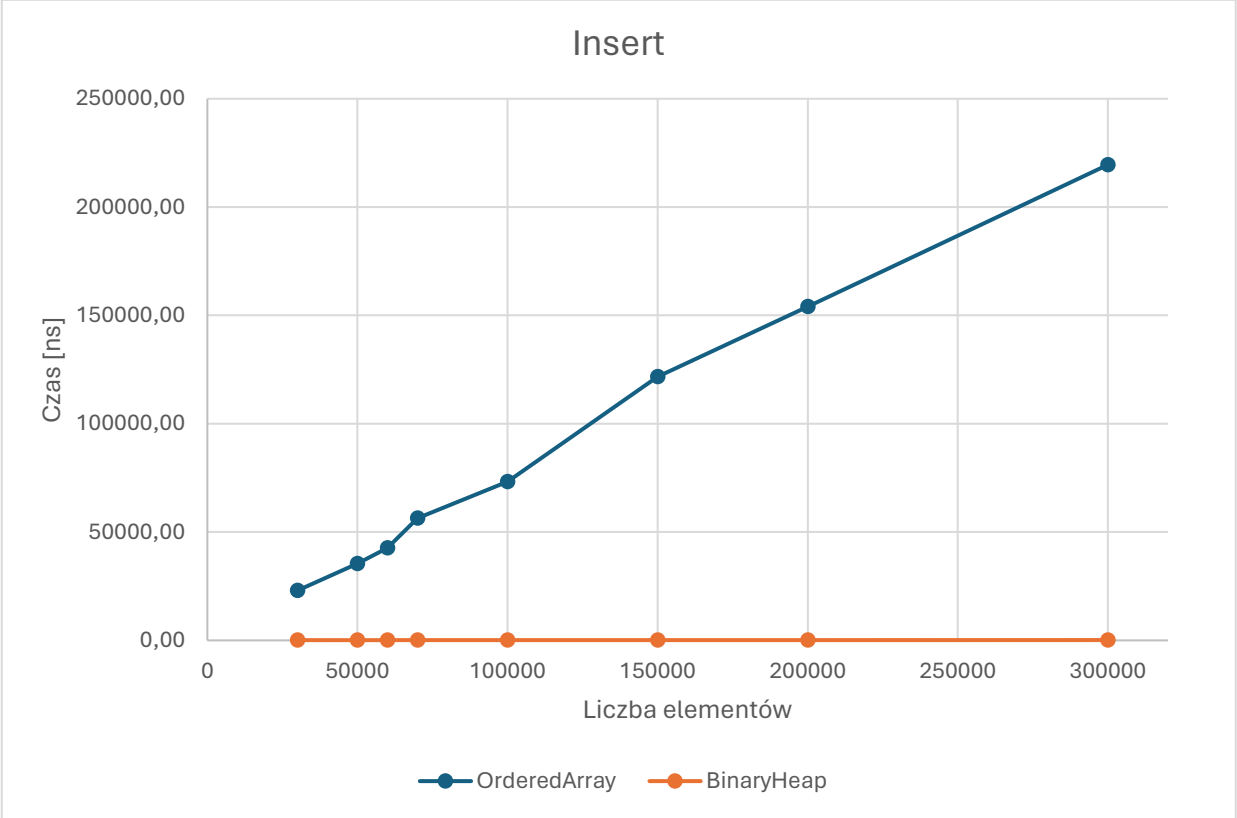
	ilość elementów							
	100000		150000		200000		300000	
DBs ->	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap	OrderedArray	BinaryHeap
Insert [ns]	73188,5	98,5	121755	120,5	154030	120	219523	139,5
Extract [ns]	135559	711,5	209998	848,5	269613	880,5	410449	1006
Peek [ns]	28	29	32,5	29,5	28	29,5	27,5	29
Modify [ns]	187577	232,5	293704	275,5	380958	287	577594	280,5
Size [ns]	30,5	28	28,5	35,5	32,5	28,5	29,5	29



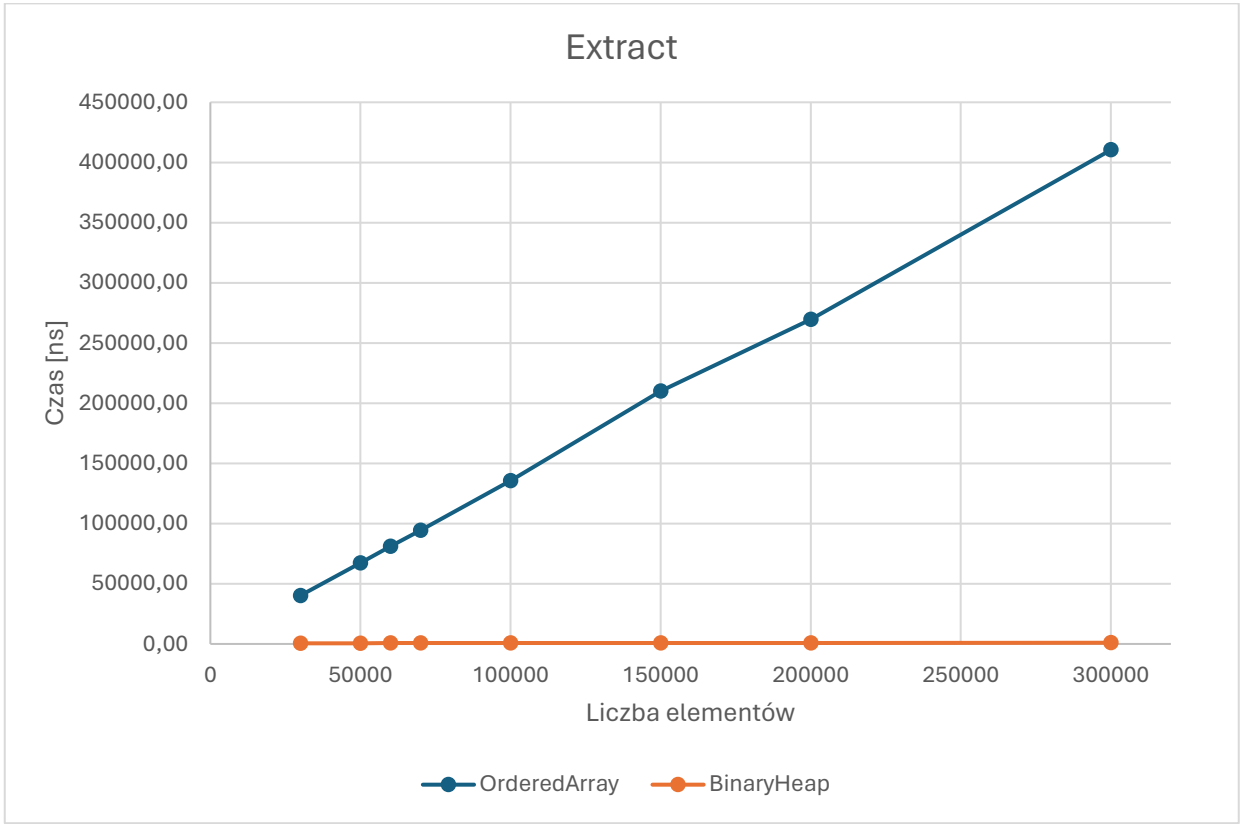
WYKRES 1: ŚREDNIE CZASY WYKONYWANIA FUNKCJI DLA ORDERED ARRAY



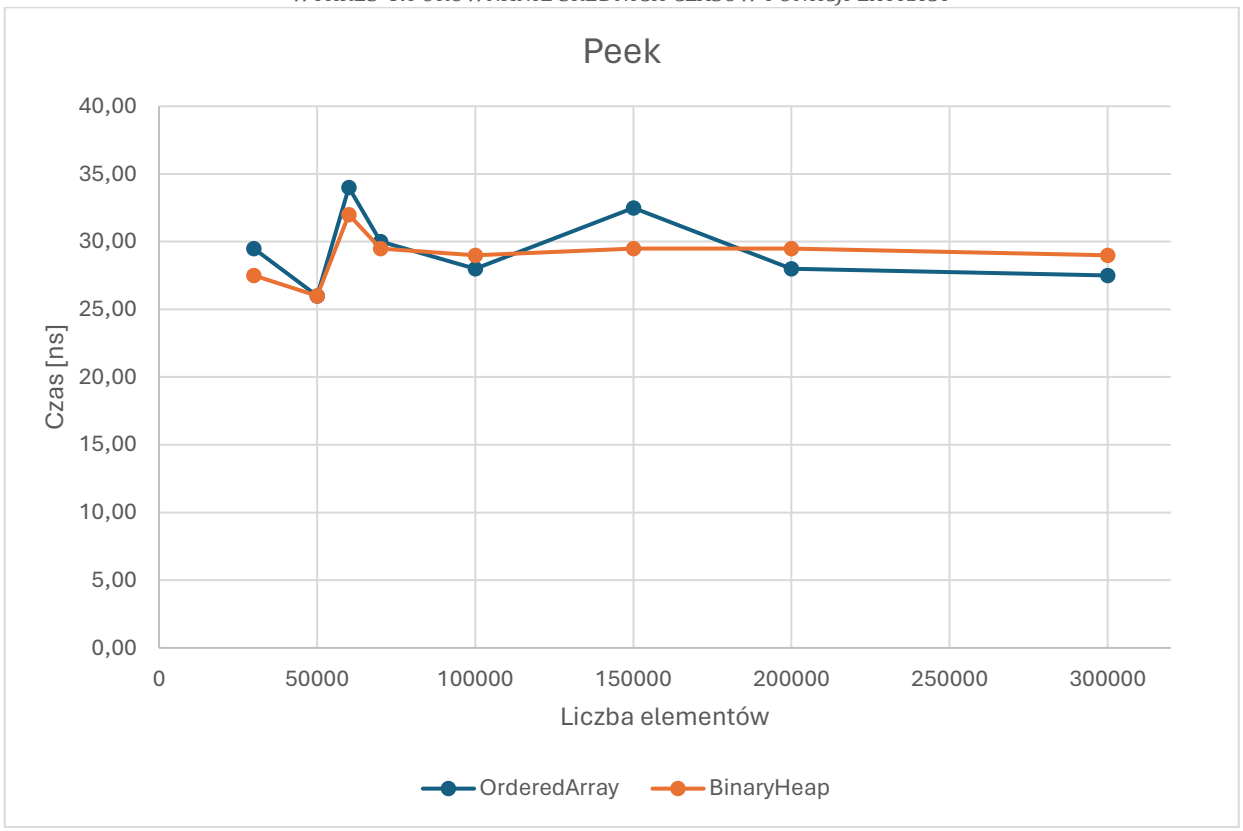
WYKRES 2: ŚREDNIE CZASY WYKONYWANIA FUNKCJI DLA BINARY ARRAY



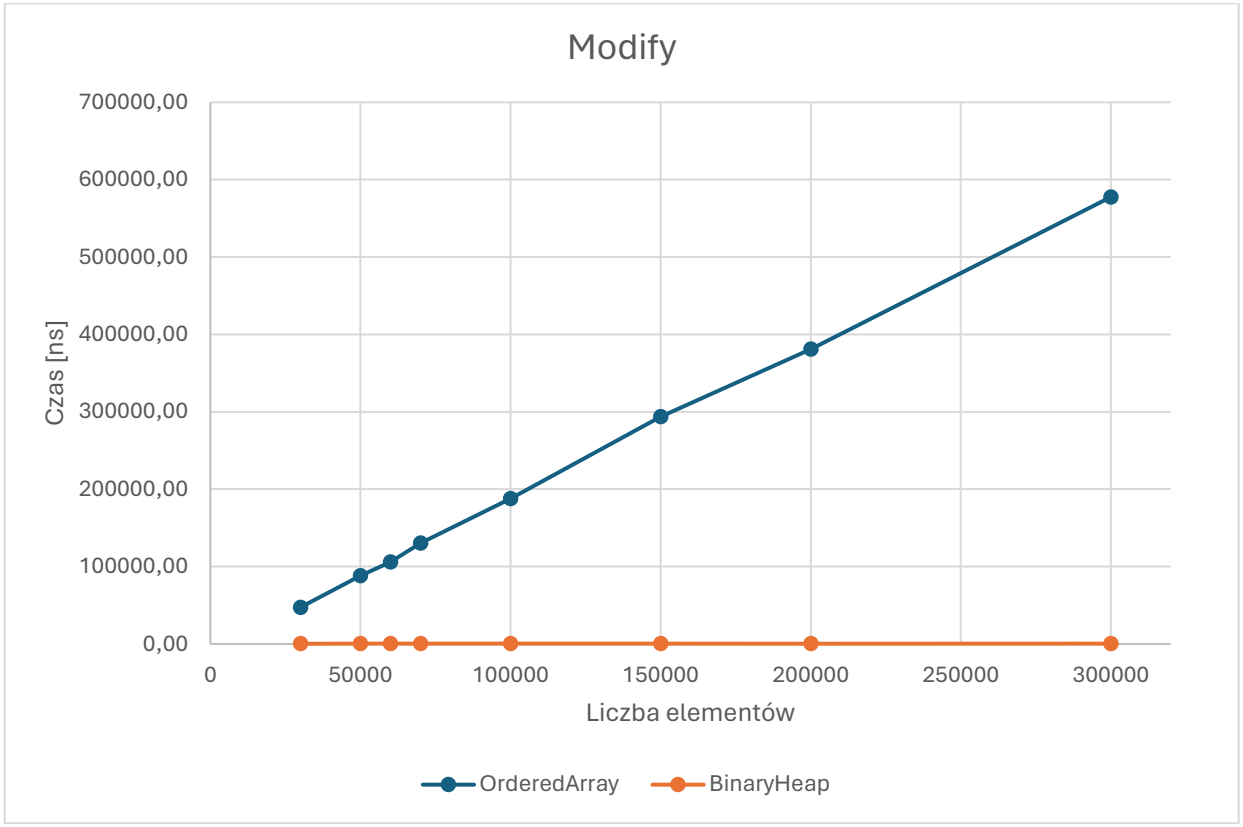
WYKRES 3:PORÓWNANIE ŚREDNICH CZASÓW FUNKCJI INSERT



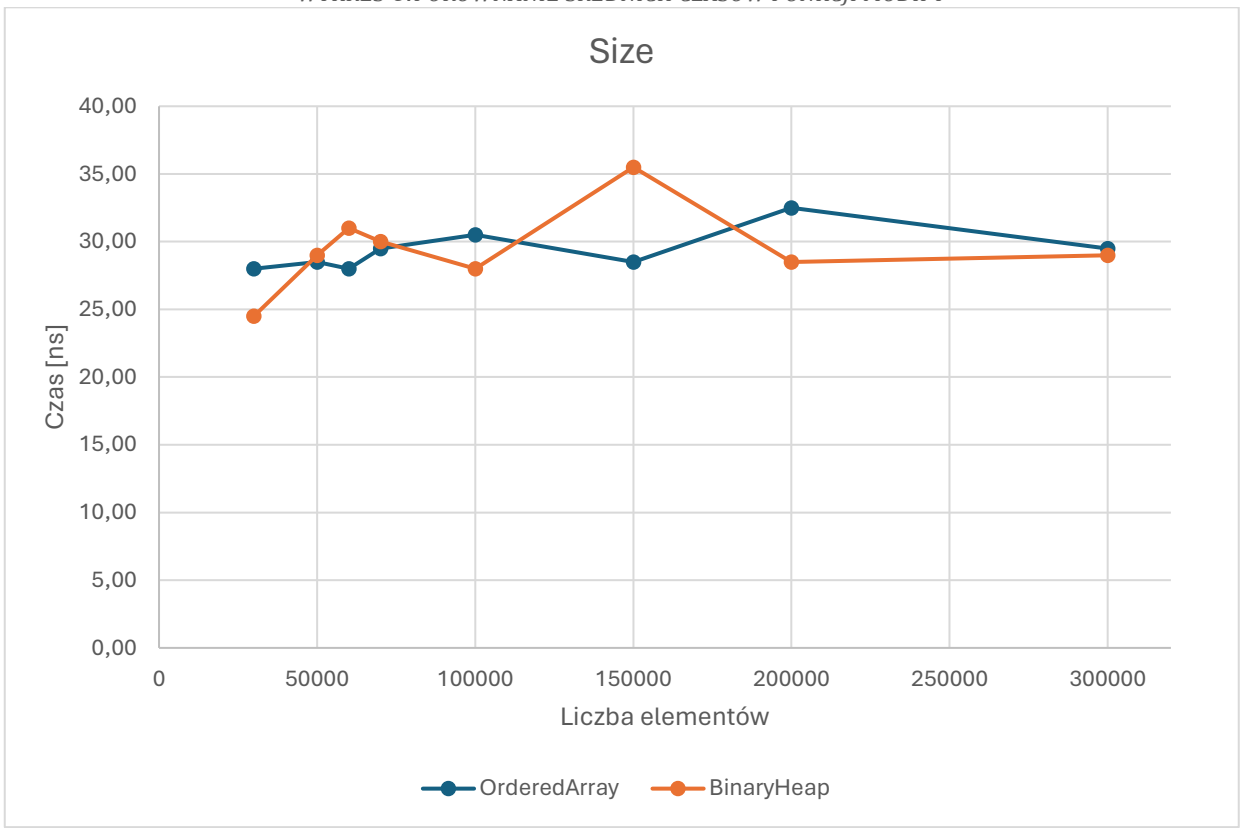
WYKRES 4:PORÓWNANIE ŚREDNICH CZASÓW FUNKCJI EXTRACT



WYKRES 5:PORÓWNANIE ŚREDNICH CZASÓW FUNKCJI PEEK



WYKRES 6:PORÓWNANIE ŚREDNICH CZASÓW FUNKCJI MODIFY



WYKRES 7:PORÓWNANIE ŚREDNICH CZASÓW FUNKCJI SIZE

6) Wnioski:

Kopiec binarny ma dużo lepszą złożoność czasową dla operacji dodawania, usuwania oraz modyfikacji priorytetu w porównaniu z posortowaną tablicą.
Jest to spowodowane wymaganiem przesuwania dużej ilości elementów tablicy podczas reorganizacji struktury dla elementów z wysokim priorytetem, przez co jest łatwo zauważyć ogromną różnicę pomiędzy strukturami dla średnich czasów wykonywania danych funkcji.

Można zaważyć że dla tablicy sortowanej średni czas wykonywania istotnych funkcji wzrasta liniowo a dla kopca binarnego logarytmicznie to potwierdza wcześniej wyliczone średnie złożoności obliczeniowej struktur.

Dzięki wykonywania pomiarów funkcji Peek można zauważyć że podczas wykonywania funkcji Extract samo zdobywanie elementu nie jest pracochłonne a tylko jego usuwanie i reorganizacja struktury.

Porównując ze sobą wykresy 1 i 2 można zauważyć że dla tablicy sortowanej najbardziej pracochłonne jest modyfikacja elementów spowodowane potrzebą:
-usunięcia elementu,
-dodanie nowego elementu ,
-reorganizacja struktury.
Natomiast dla kopca binarnego jest to Extract spowodowane większą ilością złożonych operacji podczas reorganizacji struktury, jednak jest to dalej o wiele szybsze w porównaniu do sortowanej tablicy.