Politechnika Wrocławska

Struktury danych

Projekt 2

Freddy-Ms
ArtsyKimiko

8 maja 2024

1 Wstęp teoretyczny

Kolejka priorytetowa to struktura danych, która umożliwia przechowywanie elementów w sposób uporządkowany według określonego kryterium priorytetowego. Innymi słowy, elementy w kolejce są sortowane na podstawie ich priorytetów, co oznacza, że elementy o wyższym priorytecie są obsługiwane przed elementami o niższym priorytecie.

Istnieje wiele sposobów implementacji kolejki priorytetowej, z których dwa wykorzystane to lista dwukierunkowa oraz kopiec binarny. Oba te podejścia mają swoje zalety i zastosowania, ale również różnice, które mogą wpłynąć na wydajność i złożoność operacji wykonywanych na kolejce.

Lista dwukierunkowa jest elastyczną strukturą danych, która pozwala na szybkie dodawanie i usuwanie elementów zarówno na początku, jak i na końcu kolejki. Dzięki temu, implementacja kolejki priorytetowej na liście dwukierunkowej może być stosunkowo prosta i intuicyjna. Każdy element w kolejce przechowuje swoją wartość oraz referencje do poprzedniego i następnego elementu, co umożliwia szybki dostęp do sąsiednich elementów.

Z drugiej strony, kopiec binarny jest strukturą drzewiastą, w której każdy węzeł ma wartość większą lub równą wartości jego dzieci. Kopiec binarny może być zaimplementowany za pomocą tablicy, co pozwala na efektywne wykorzystanie pamięci oraz szybkie operacje wstawiania, usuwania i znajdowania elementów o najwyższym priorytecie.

2 Inicjalizacja

Implementację kolejki priorytetowej na liście dwukierunkowej nazwano `ListQueue`, natomiast kolejkę na kopcu - `BinaryQueue`. Nazwy te wykorzystywane są w dalszej części sprawozdania.

W programie badającym dwie implementacje kolejki priorytetowej wykorzystano poniższe funkcje:

- ReadFile - wczytuje dane z pliku tekstowego
- GetSize - wyświetla ilość elementów
- AddHighest - dodaje element o najwyższym priorytecie
- AddLowest - dodaje element o najniższym priorytecie
- GetHighest - usuwa element o najwyższym priorytecie
- GetLowest - usuwa element o najniższym priorytecie
- Peek - podgląda element bez usuwania
- ModifyFirst - zmienia priorytet pierwszego napotkanego elementu o podanej wartości
- Modify - zmienia priorytet każdego napotkanego elementu o podanej wartości

Tabela 1: Złożoności obliczeniowe zaimplementowanych funkcji

Funkcja	Złożoność	
	ListQueue	BinaryQueue
ReadFile	$O(n^2)$	$O(n \cdot log(n))$
GetSize	$O(1)$	$O(1)$
AddHighest	$O(1)$	$O(log(n))$
AddLowest	$O(n)$	-
GetHighest	$O(1)$	$O(log(n))$
GetLowest	$O(1)$	-
Peek	$O(1)$	$O(1)$
ModifyFirst	$O(n)$	$O(log(n))$
Modify	$O(x \cdot n)$	$O(x \cdot log(n))$

Gdzie `x` oznacza liczbę znalezionych elementów o podanej wartości, a `n` oznacza liczbę wszystkich elementów.

2.1 Lista dwukierunkowa

Do realizacji kolejki priorytetowej tą metodą zaimplementowano klasę `ListQueue` oraz szablon `T`.

- **Szablon**:
  - `template <typename T>`: Deklaruje klasę szablonu, gdzie T jest zmienną zastępczą dla typu danych.
- **Struktura Node**:
  - Prywatna struktura `Node` reprezentuje element w kolejce. Każdy węzeł przechowuje dane, priorytet oraz wskaźniki na poprzedni i następny węzeł. Konstruktor tej struktury inicjuje wartości danych, priorytetu oraz wskaźników na poprzedni i następny węzeł.

```
struct Node{
    T data;
    int priority;
    Node* next;
    Node* prev;
    // Constructor initializes data, priority, and pointers to previous and
        next nodes
    Node(T data, int priority, Node* next = nullptr, Node* prev = nullptr) :
        data(data), priority(priority), next(next), prev(prev) {}
};
```

- **Metody prywatne**:
  - `PushToBack(Node* start)`: Przesuwa element w tył kolejki na podstawie jego priorytetu.
  - `PushToFront(Node* start)`: Przesuwa element do przodu kolejki na podstawie jego priorytetu.
- **Metody publiczne**:
  - `Add(T element, int priority)`: Dodaje element o określonym priorytecie do kolejki.

```
virtual void Add(T element, int priority) override{
    Node* node = new Node(element, priority,head,nullptr);
    if(head == nullptr){
        head = node;
        tail = node;
    }else{ // Update the head and link the previous head to the new node
        head->prev = node;
        head = node;
    }
    PushToBack(head); // Push the newly added node to maintain priority order
    actualsize++;
}
```

  - `PeekAt(int index)`: Wyświetla dane elementu o określonym indeksie w kolejce.

```
void PeekAt(int index){
    if(index == 1)
        return Peek();
    if(index == actualsize)
        return PeekLast();
    Node* temp = head;
    for(int i = 1; i < index; i++)
        temp = temp->next;
    cout << "Element at index " << index << " is: " << temp->data << endl;
}
```

– `Peek()`: Wyświetla pierwszy element w kolejce bez usuwania go.
– `PeekLast()`: Wyświetla dane ostatniego elementu w kolejce.
– `GetSize()`: Zwraca aktualną liczbę elementów w kolejce.
– `GetHighest()`: Usuwa i zwraca element o najwyższym priorytecie z kolejki.
– `GetLast()`: Usuwa i zwraca element o najniższym priorytecie z tyłu kolejki.
– `GetAt(int index)`: Usuwa i zwraca element o określonym indeksie w kolejce.

```
T GetAt(int index){
    if(index == 1)
        return GetHighest();
    if(index == actualsize)
        return GetLast();
    Node* toDelete = head;
    for(int i = 1; i < index; i++)
        toDelete = toDelete->next;
    // Adjust the pointers to remove the node from the queue
    toDelete->prev->next = toDelete->next;
    toDelete->next->prev = toDelete->prev;
    T data = toDelete->data;
    delete toDelete;
    actualsize--;
    return data;
}
```

– `PrintData()`: Wyświetla wartości wszystkich elementów kolejki.
– `PrintPriority()`: Wyświetla priorytety wszystkich elementów kolejki.
– `ReadFile(string filename, int size)`: Wczytuje dane z pliku i dodaje je do kolejki.
– `Modify(T element, int newPriority)`: Modyfikuje priorytet określonego elementu w kolejce.

```
virtual void Modify(T element, int newPriority) override{
    bool found = false;
    Node* node = head;
    while(node != nullptr){
        if(node->data == element){
            found = true;
            // If higher, move element to back of queue
            if(node->priority < newPriority){
                node->priority = newPriority;
                Node* temp = node;
                PushToBack(temp);
            }else if(node->priority > newPriority){
            // If lower, move element to front of queue
                node->priority = newPriority;
                Node* temp = node;
                PushToFront(temp);
            }
        }
        node = node->next;
    }
    if(!found) cout << "Element not found" << endl;
}
```

– `ModifyFirst(T element, int newPriority)`: Modyfikuje priorytet pierwszego napotkanego elementu o danej wartości.

2.2 Kopiec binarny

W celu zrealizowania tego rozwiązania zdefiniowano klasę `BinaryHeap`. poniżej zostały przedstawione funkcje tej klasy wraz z krótkimi opisami oraz cytowanymi fragmentami kodu.

- **Struktura Node**:
  - Prywatna struktura `Node` reprezentuje element w kopcu. Każdy element składa się z danych (typu `T`) i priorytetu (typu `int`).
- **Metody prywatne**:
  - `HeapifyUp(int index)`: Naprawia strukturę kopca w górę po dodaniu nowego elementu.

```
void HeapifyUp(int index){
    int parent = (index - 1) / 2;
    while(index >= 0 && HeapArr[parent].priority > HeapArr[index].priority){
        swap(HeapArr[parent], HeapArr[index]);
        index = parent;
        parent = (index - 1) / 2;
    }
}
```

  - `HeapifyDown(int index)`: Naprawia strukturę kopca w dół po usunięciu elementu.

```
void HeapifyDown(int index){
    while(1){
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int smallest = index;
        // Find the smallest child
        if(left<actualsize && HeapArr[left].priority<HeapArr[index].priority)
            smallest = left;
        if(right<actualsize && HeapArr[right].priority<HeapArr[smallest].
            priority)
            smallest = right;
        if(smallest != index){
            swap(HeapArr[index], HeapArr[smallest]);
            index = smallest;
        }else break;
    }
}
```

  - `IsFull()`: Sprawdza, czy kopiec jest pełny.
  - `IsEmpty()`: Sprawdza, czy kopiec jest pusty.
  - `Resize()`: Zmienia rozmiar tablicy kopca w razie potrzeby.
- **Metody publiczne**:
  - `BinaryHeap(int Capacity)`: Konstruktor z argumentem, tworzy kopiec o określonej pojemności.
  - `~BinaryHeap()`: Destruktor, zwalnia pamięć zaalokowaną dla tablicy kopca.
  - `Add(T element, int priority)`: Dodaje element o określonym priorytecie do kopca.

```
virtual void Add(T element, int priority) override{
    if(IsFull()) Resize();
    HeapArr[actualsize]= {element, priority};
    HeapifyUp(actualsize); // Restore property
    actualsize++;
}
```

– `GetHighest()`: Zwraca element o najwyższym priorytecie z kopca.

```
virtual T GetHighest() override{
    if(IsEmpty()) throw "Queue is empty";
    T temp = HeapArr[0].data;
    HeapArr[0] = HeapArr[actualsize - 1];
    actualsize--;
    HeapifyDown(0); // Restore property
    return temp;
}
```

– `GetSize()`: Wyświetla aktualną liczbę elementów w kopcu.

– `Peek()`: Wyświetla pierwszy element w kolejce priorytetowej.

– `PrintData()`: Wyświetla wartości wszystkich elementów w kolejności z kopca.

– `PrintPriority()`: Wyświetla priorytety wszystkich elementów w kolejności z kopca.

– `ReadFile(string filename, int size)`: Odczytuje dane z pliku i dodaje je do kopca.

– `Modify(T element, int newPriority)`: Modyfikuje priorytet określonego elementu w kopcu.

```
virtual void Modify(T element, int newPriority) override{
    bool found = false;
    for(int i = 0; i < actualsize; i++){
        // If element is found, updates its priority and restores the heap
        if(HeapArr[i].data == element){
            found = true;
            if(newPriority > HeapArr[i].priority){
                HeapArr[i].priority = newPriority;
                HeapifyDown(i); // Restore property
            }else{
                HeapArr[i].priority = newPriority;
                HeapifyUp(i); // Restore property
            }
        }
    }
    if(!found)
        cout << "Element not found" << endl;
}
```

– `ModifyFirst(T element, int newPriority)`: Modyfikuje priorytet pierwszego napotkanego elementu o danej wartości.

3 Badania

W celu przeprowadzenia eksperymentu weryfikującego efektywność zaimplementowanych algorytmów, opracowaliśmy procedurę generującą losowe dane składające się z milionów elementów wraz z przypisanymi im priorytetami, a następnie zapisującą je do pliku tekstowego. Warto zauważyć, że priorytety były wygenerowane losowo z zakresu od 2 do 1 000 000. Takie podejście gwarantuje, że zarówno dla struktury danych opartej na kolejce binarnej, jak i dla struktury danych opartej na liście, możliwe było równorzędne wczytywanie danych, jednocześnie zapewniając, że element o najwyższym priorytecie (1) zawsze będzie pierwszy.

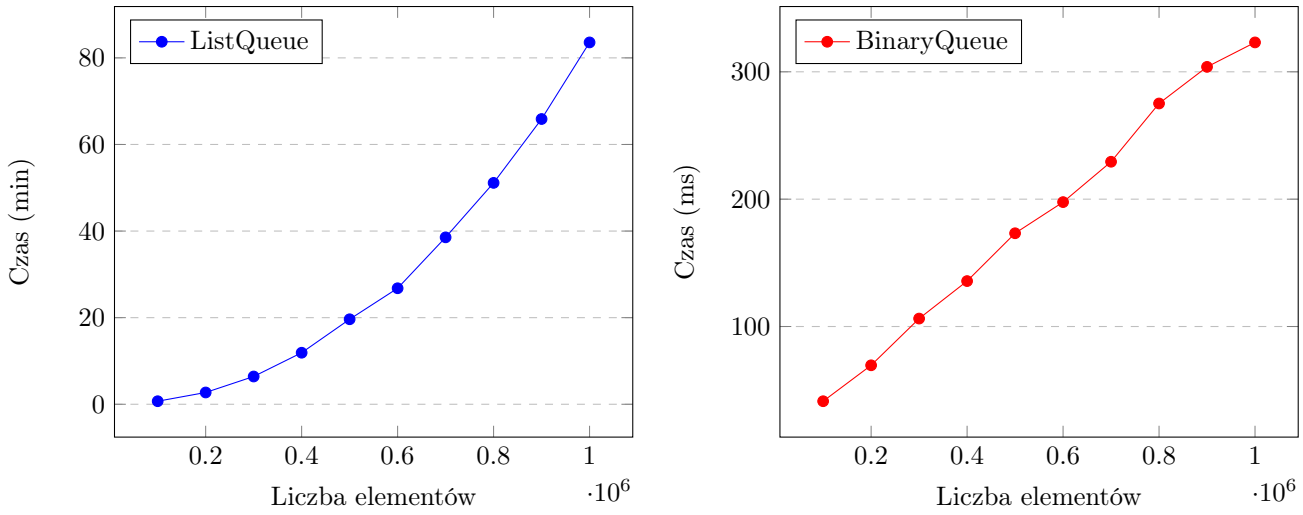
Poniższe badania zostały przeprowadzone na komputerze z procesorem Intel(R) Core(TM) i3-8130U CPU przy 8 GB pamięci RAM.

3.1 Wczytywanie danych

W przypadku struktury danych opartej na liście, dodanie pojedynczego elementu ma złożoność obliczeniową O(n), gdzie n jest liczbą elementów w liście. Jednakże, załadowanie całej struktury z pliku skutkuje złożonością czasową $O(n^2)$, ponieważ dla każdego elementu konieczne jest wywołanie funkcji dodawania, co wymaga n operacji o złożoności O(n). W przypadku struktury danych opartej na kolejce priorytetowej na kopcu, procedura jest analogiczna; wykonujemy n operacji, z których każda ma złożoność O(log n), co prowadzi do całkowitej złożoności czasowej O(n log n).

Tabela 2: Czas wczytywania elementów (funkcja ReadFile)

Liczba elementów	ListQueue [min]	BinaryQueue [ms]
100 000	0,70	41,34
200 000	2,72	69,55
300 000	6,41	106,24
400 000	11,90	135,68
500 000	19,63	173,23
600 000	26,79	197,69
700 000	38,55	229,38
800 000	51,13	275.13
900 000	65,88	303,87
1 000 000	83,57	323,08



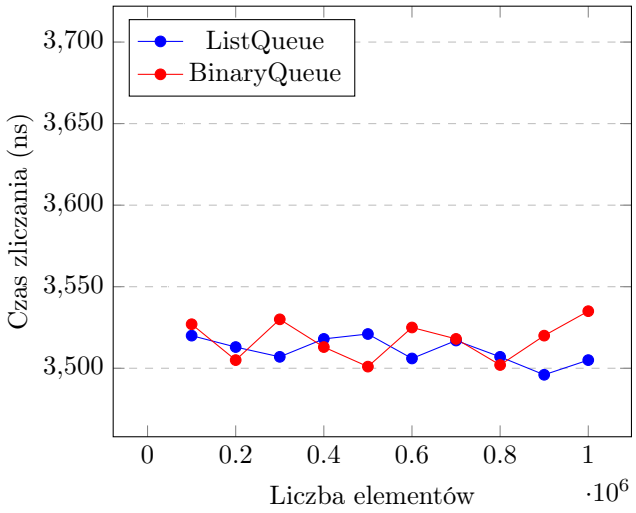
Rysunek 1: Czas wczytywania danych

3.2 Czas wyświetlenia liczby elementów

Z uwagi na dynamiczną naturę kolejki, gdzie jej rozmiar jest aktualizowany natychmiastowo wraz z dodawaniem lub usuwaniem elementów, funkcja zwracająca aktualny rozmiar kolejki charakteryzuje się stałą złożonością czasową O(1). Jest to wynik tego, że funkcja po prostu zwraca wartość zmiennej, która przechowuje aktualny rozmiar kolejki, oznaczoną jako `actualsize`.

Tabela 3: Czas potrzebny na wyświetlenie liczby elementów (funkcja `GetSize`)

Liczba elementów	ListQueue [ns]	BinaryQueue [ns]
100 000	3520	3527
200 000	3513	3505
300 000	3507	3530
400 000	3518	3513
500 000	3521	3501
600 000	3506	3525
700 000	3517	3518
800 000	3500	3502
900 000	3496	3520
1 000 000	3505	3535



Rysunek 2: Porównanie czasów potrzebnych na wyświetlenie liczby elementów

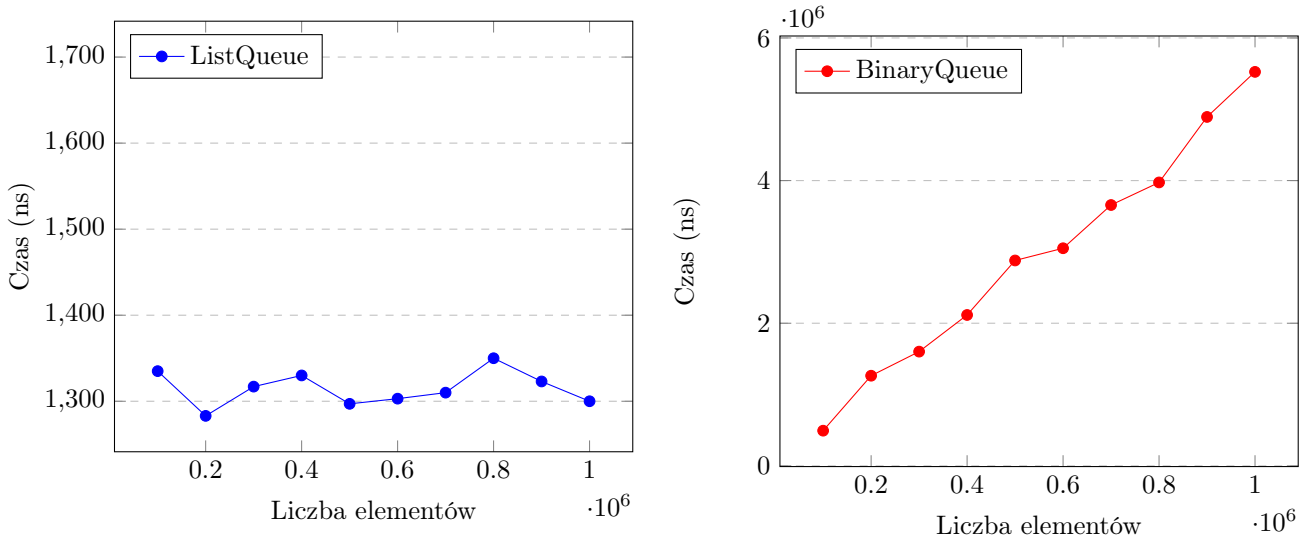
3.3 Dodawanie elementów

Dodawanie elementu do kopca binarnego polega na wstawieniu go na ostatnią pozycję w kopcu, a następnie naprawieniu struktury kopca za pomocą operacji `HeapifyUp`. Złożoność czasowa operacji wstawiania wynosi O(log n), gdzie n to liczba elementów w kopcu. Jest to spowodowane faktem, że wysokość kopca binarnego jest ograniczona przez logarytm dziesiętny liczby elementów.

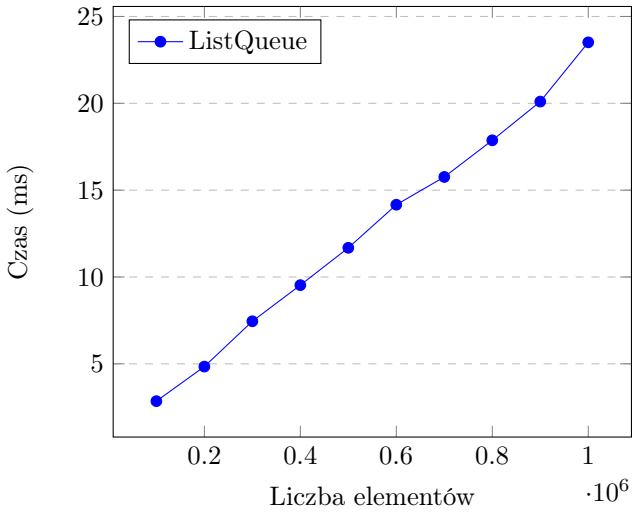
W przypadku kolejki priorytetowej opartej na liście, dodawanie elementu ma złożoność czasową O(n), ponieważ musimy znaleźć odpowiednie miejsce dla nowego elementu w liście. Jednakże, w przypadku dodawania elementu o najwyższym priorytecie (1), złożoność czasowa wynosi O(1), gdyż nowy element jest dodawany na początek listy i nie wymaga przesuwania innych elementów.

Tabela 4: Porównanie czasów dodawania elementów

Liczba elementów	AddHighest [ns]		AddLowest [ms]
	ListQueue	BinaryQueue	ListQueue
100000	1335	493400	2,85
200000	1283	1264100	4,84
300000	1317	1601900	7,45
400000	1330	2115700	9,53
500000	1297	2879800	11,68
600000	1303	3051400	14,16
700000	1310	3657500	15,76
800000	1350	3974100	17,87
900000	1323	4892400	20,10
1000000	1300	5524200	23,51



Rysunek 3: Porównanie czasów dodawania elementu o najwyższym priorytecie



Rysunek 4: Czas dodawania elementu o najniższym priorytecie

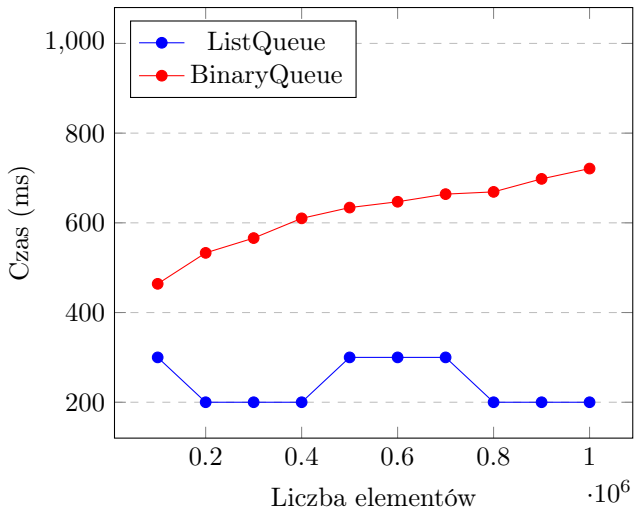
3.4 Usuwanie elementów

Usuwanie elementów w przypadku kopca binarnego rzeczywiście ma taką samą złożoność, jak dodawanie elementu. Polega ono na zamianie pierwszego elementu z ostatnim, usunięciu pierwszego elementu, a następnie naprawieniu struktury kopca w dół przy pomocy operacji `HeapifyDown`. Złożoność czasowa tego procesu również wynosi O(log n), ponieważ zależy od wysokości kopca.

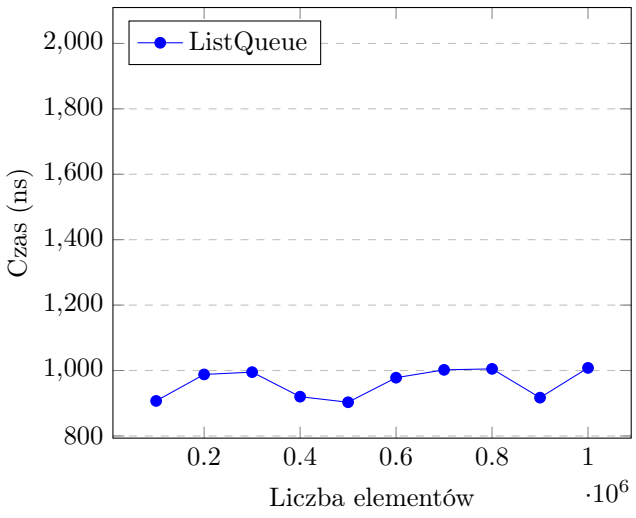
W przypadku kolejki priorytetowej opartej na liście, usunięcie elementów jest zróżnicowane. Usunięcie pierwszego elementu, czyli elementu o najwyższym priorytecie, ma złożoność czasową O(1), ponieważ wystarczy usunąć ten element i przesunąć wskaźnik głowy na następny. Analogicznie, usunięcie ostatniego elementu, czyli elementu o najniższym priorytecie, również ma złożoność O(1), ponieważ wystarczy usunąć ten element i przesunąć wskaźnik ogona na poprzedni. Jednakże usunięcie elementu znajdującego się w dowolnej innej pozycji w kolejce może wymagać dostania się do odpowiedniego elementu, co wymaga przeglądania listy i ma złożoność O(n).

Tabela 5: Porównanie czasów usuwania elementów

Liczba elementów	GetHighest [ns]		GetLowest [ns]
	ListQueue	BinaryQueue	ListQueue
100 000	300	464	907
200 000	200	533	988
300 000	200	566	995
400 000	200	610	930
500 000	300	634	903
600 000	300	647	978
700 000	300	664	1002
800 000	200	669	1005
900 000	200	698	917
1 000 000	200	721	1008



Rysunek 5: Porównanie czasów usuwania elementu o najwyższym priorytecie



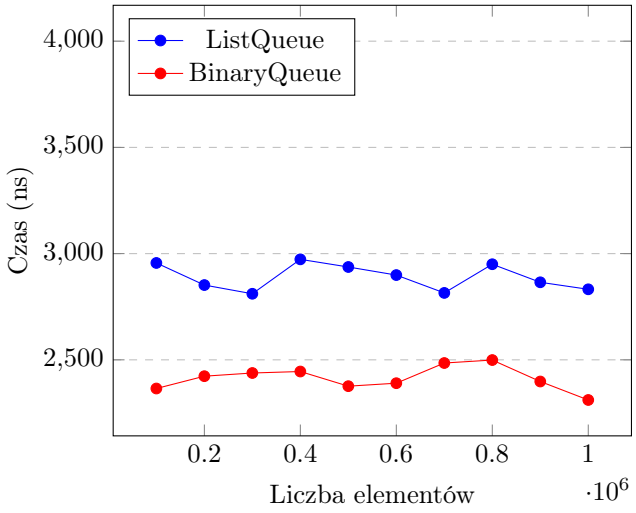
Rysunek 6: Czas usuwania elementu o najniższym priorytecie

3.5 Peek

W przypadku zarówno kolejki opartej na kopcu binarnym, jak i na liście, złożoność czasowa funkcji peek wynosi O(1), ponieważ wystarczy zwrócić pierwszy element kolejki, który zawiera element o najwyższym priorytecie.

Tabela 6: Peek (funkcja Peek)

Liczba elementów	ListQueue [ns]	BinaryQueue [ns]
100000	2956	2365
200000	2852	2423
300000	2811	2438
400000	2973	2445
500000	2937	2376
600000	2899	2390
700000	2815	2485
800000	2950	2499
900000	2865	2398
1000000	2832	2311



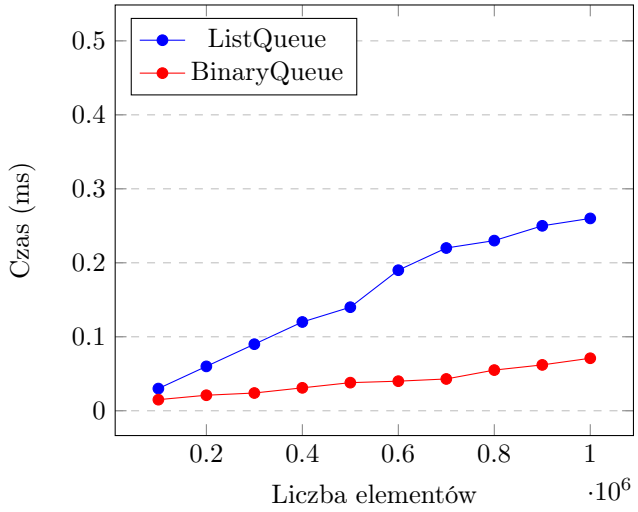
Rysunek 7: Porównanie czasów operacji peek

3.6 Modyfikacja elementu

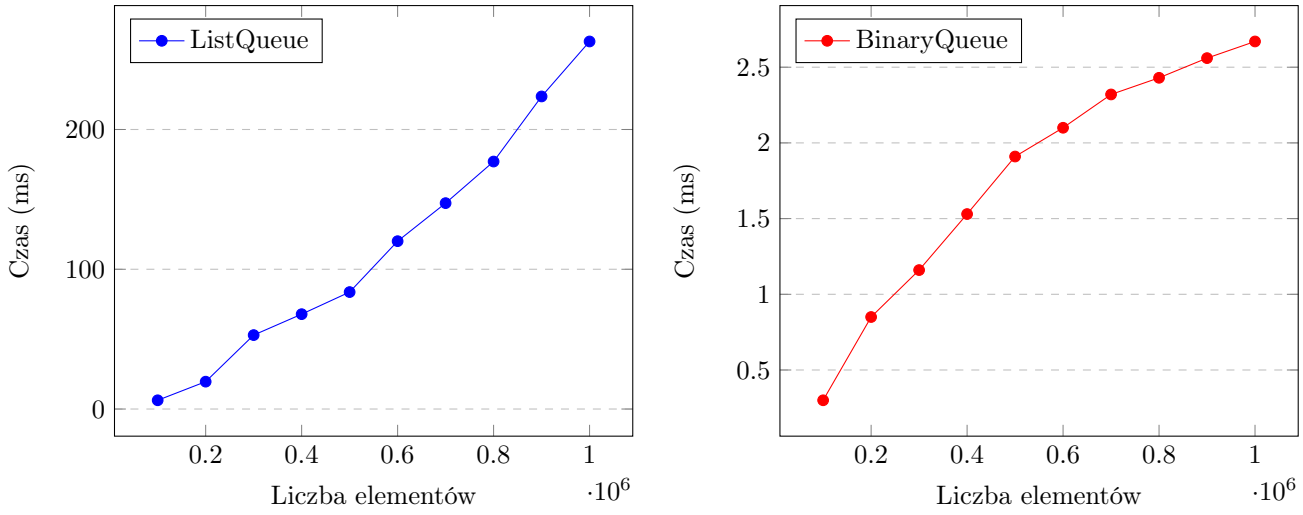
Procedura modyfikacji priorytetu, pierwszego elementu występującego w kolejce priorytetowej, ma złożoność obliczeniową O(log n) dla kopca binarnego oraz O(n) dla listy. W przypadku zmiany priorytetów wszystkich elementów o określonej wartości, złożoność ta zostaje pomnożona przez liczbę elementów o tej wartości.

Tabela 7: Porównanie czasów modyfikowania elementów

Liczba elementów	ModifyFirst [ms]		Modify [ms]	
	ListQueue	BinaryQueue	ListQueue	BinaryQueue
100000	0,03	0,015	6,26	0,30
200000	0,06	0,021	19,58	0,85
300000	0,09	0,024	52,90	1,16
400000	0,12	0,031	67,87	1,53
500000	0,14	0,038	83,68	1,91
600000	0,19	0,040	120,09	2,10
700000	0,22	0,043	147,31	2,32
800000	0,23	0,055	177,07	2,43
900000	0,25	0,062	223,64	2,56
1000000	0,26	0,071	262,98	2,67



Rysunek 8: Porównanie czasów modyfikacji pierwszego elementu



Rysunek 9: Porównanie czasów modyfikacji elementu

4 Wnioski

- **Złożoność czasowa operacji:**
  - Dla większości operacji, w tym dodawania, usuwania i modyfikowania elementów, kopiec binarny wykazuje lepszą złożoność czasową niż lista dwukierunkowa. Wynika to głównie z faktu, że kopiec binarny ma złożoność czasową O(log n) dla operacji dodawania, usuwania i modyfikowania elementów, podczas gdy lista dwukierunkowa ma złożoność czasową O(n) lub O(x·n), gdzie x to liczba elementów o podanej wartości.
  - Jedynie, dla niektórych operacji, takich jak wczytywanie danych, lista dwukierunkowa może wykazywać lepszą wydajność dla mniejszych zbiorów danych. Jednak dla większych ilości jest ona nieoptymalna ponieważ ma złożoność czasową $O(n^2)$, podczas gdy kopiec binarny ma złożoność O(n log n).
  - Podczas analizy porównawczej procesów ładowania miliona danych w strukturze listy oraz kopca wykazano istotne różnice czasowe. W przypadku struktury listy, proces ten wymagał ponad godzinę, co jest efektem skomplikowanej złożoności obliczeniowej i operacji wymaganych do dostosowania kolejności oraz alokacji pamięci dla każdego elementu. W odróżnieniu od tego, proces ładowania tego samego miliona danych do struktury kopca został zrealizowany w czasie nieprzekraczającym sekundy. Jest to konsekwencją efektywnego mechanizmu przetrzymywania danych w kopcu, który minimalizuje koszt obliczeniowy związany z operacjami wstawiania i organizowania elementów w strukturze danych. Te wyniki podkreślają znaczenie wyboru odpowiedniej struktury danych w zależności od specyfiki operacji oraz wymagań dotyczących wydajności.
  - W przypadku operacji związanych z odczytywaniem aktualnej liczby elementów w kolejce (`GetSize`), oba podejścia wykazują podobną wydajność, złożoność czasowa wynosi O(1) dla obu struktur.
- **Efektywność pamięciowa:**
  - Kopiec binarny, będąc strukturą tablicową, może być bardziej efektywny pod względem zużycia pamięci w porównaniu z listą dwukierunkową, która wymaga dodatkowych wskaźników dla każdego elementu.
  - Lista dwukierunkowa może być bardziej elastyczna i umożliwia szybsze dodawanie i usuwanie elementów na początku oraz na końcu kolejki, co może być korzystne w niektórych przypadkach.
- **Wydajność operacji:**
  - Operacje dodawania i usuwania elementów o najwyższym priorytecie (`AddHighest`, `GetHighest`) są znacznie szybsze w kopcu binarnym ze względu na jego właściwości strukturalne, co sprawia, że operacje te mają złożoność czasową O(log n).
  - Operacje modyfikowania priorytetów (`Modify`, `ModifyFirst`) są wydajniejsze w kopcu binarnym, ponieważ naprawa struktury kopca po zmianie priorytetu ma złożoność czasową O(log n), podczas gdy w liście dwukierunkowej zależy od liczby elementów o podanej wartości.
  - Operacje odczytywania wartości pierwszego elementu (`Peek`) są podobnie wydajne w obu strukturach, mając złożoność czasową O(1).