



Politechnika Wrocławska

Projekt 2 - Struktury Danych „Kolejka priorytetowa”

Wydział Informatyki i Telekomunikacji
Informatyczne Systemy Automatyki

Aleksander Żołnowski, 272536
Franciszek Jeziorowski, 272526

Spis treści

1 Wstęp teoretyczny.....	3
1.1 Kolejka jako struktura danych.....	3
1.2 Kolejka priorytetowa	3
2 Implementacja	3
2.1 Lista wiązana.....	3
2.2 Kopiec	4
3 Badania.....	5
3.1 Lista wiązana.....	5
3.1.1 Push - Dodanie elementu.....	5
3.1.2 Pop - Usunięcie i zwrócenie elementu	7
3.1.3. Peek - Podejrzenie elementu	8
3.1.4 GetSize – Zwrócenie rozmiaru	9
3.1.5 Change – Modyfikacja priorytetu	10
3.1.6 Złożoność obliczeniowa	11
3.2 Kopiec	13
3.2.1 Insert.....	13
3.2.2 RemoveMax	16
3.2.3 Peak	17
3.2.4 ReturnSize	18
3.2.5 ModifyPriority.....	19
3.2.6 Analiza złożoności obliczeniowej	21
4 Porównanie.....	22
5 Wnioski	23
6 Bibliografia.....	24

1 Wstęp teoretyczny

1.1 Kolejka jako struktura danych

Kolejki to podstawowa struktura danych, która operuje według zasady „FIFO – pierwszy wszedł, pierwszy wyszedł (First-In-First-Out)”. Elementy są dodawane na końcu kolejki, a usuwane z jej początku. Kolejki jako struktura danych mają szerokie zastosowanie w dziedzinie informatyki.

1.2 Kolejka priorytetowa

Kolejki priorytetowe to specjalna odmiana kolejek, używana do przechowywania i zarządzania elementami w sposób uporządkowany według określonego priorytetu. Każdy element ma przypisany swój priorytet.

W przeciwieństwie do tradycyjnych kolejek FIFO (First-In-First-Out), w których elementy są usuwane w kolejności ich dodania, w kolejce priorytetowej elementy są usuwane na podstawie ich priorytetu. Zdejmowany jest zawsze element o najwyższym lub najniższym priorytecie w zależności od implementacji, czy jest to kolejka priorytetowa typu min czy max.

2 Implementacja

Istnieje wiele sposobów implementacji kolejek priorytetowych, w tym projekcie porównamy ze sobą implementacje poprzez listę wiązana oraz kopiec.

2.1 Lista wiązana

Implementacja kolejki priorytetowej typu max za pomocą listy wiązanej.

Lista wiązana to sekwencja węzłów, w której każdy element zawiera wskaźnik do następnego elementu listy. W przypadku implementacji kolejki priorytetowej za pomocą listy wiązanej, nasza struktura węzła zawierała będzie jeszcze wartość elementu i jego priorytet. Sama struktura kolejki w swojej implementacji zawiera dwa wskaźniki: „front” – wskaźnik na początek kolejki i „rear” – koniec kolejki.

W celu implementacji kolejki priorytetowej utworzono jej klasę, która umożliwia główne operacje na strukturze:

- Push(wartość, priorytet) - dodanie elementu do kolejki (metoda dodaje element w odpowiednim miejscu w zależności od jego priorytetu)
- Pop() – zwraca i usuwa element o najwyższym priorytecie

- Peek() – podejrzenie elementu o najwyższym priorytecie
- Change(wartość, nowy priorytet) – zmiana priorytetu istniejącego elementu kolejki
- GetSize() – zwraca rozmiar kolejki

2.2 Kopiec

Implementacja kolejki priorytetowej typu max za pomocą kopca.

Kopiec maksymalny to struktura opierająca się na drzewie binarnym. Każdy element takiego drzewa jest wskazywany przez osobny index tablicy dynamicznej, zachowujący zależności:

i – rodzic

$i * 2 + 1$ – lewe dziecko

$i * 2 + 2$ – prawe dziecko

W przypadku kopca maksymalnego, mamy zachowaną także zależność - każdy rodzic jest większy od swoich dzieci. Kolejka priorytetowa oparta na kopcu maksymalnym, zapisanym na tablicy dynamicznej zawiera wskaźnik na początek tablicy dynamicznej, zmienną size – ilość elementów oraz capacity – pojemność tablicy dynamicznej.

W implementacji kolejki priorytetowej opartej na kopcu maksymalnym, zastosowałem następujące metody w klasie:

Funkcje główne:

- insert(wartość, priorytet) – wprowadza nowy element do kolejki o określonej wartości i priorytecie
- removeMax() – usuwa element ze szczytu kopca i zwraca jego wartość
- modifyPriority(wartość elementu do zmiany, nowy priorytet) – zmienia priorytet elementu o podanej wartości
- returnSize() – zwraca aktualną ilość elementów kopca
- peak() – zwraca wartość z szczytu kolejki (roota)

Funkcje wymagane do poprawnego działania:

- heapifydown(index) – funkcja potrzebna do zachowania właściwości kopca maksymalnego. Sprawdza czy każdy element poniżej podanego elementu o indexie danym w argumentach, ma mniejszy priorytet niż rodzic. Jeżeli nie, sortuje kopiec do momentu aż zachowany będzie taki warunek.

- heapifyup(index) - funkcja potrzebna do zachowania właściwości kopca maksymalnego. Sprawdza czy każdy element powyżej podanego elementu o indexie danym w argumentach, ma większy priorytet. Jeżeli nie, sortuje kopiec do momentu aż zachowany będzie taki warunek.

Funkcje pomocnicze:

- display() – Wyświetla wszystkie zmienne podanego obiektu klasy kolejki priorytetowej oraz wyświetla po kolei każdy element, jego wartość, priorytet oraz index

3 Badania

3.1 Lista wiązana

W celu oceny wydajności implementacji kolejki priorytetowej za pomocą listy wiązanej przeprowadzono serię testów głównych operacji wykonywanych na kolejkach priorytetowych. Testy zostały przeprowadzone na różnych rozmiarach kolejki, które zwiększały się logarytmicznie o 10-krotność dla każdego kroku. Zakres wielkości kolejki priorytetowej, na którym przeprowadzono badania, wynosił od 10 do 1 000 000 elementów. Wydajność badanych operacji została zmierzona w nanosekundach (ns).

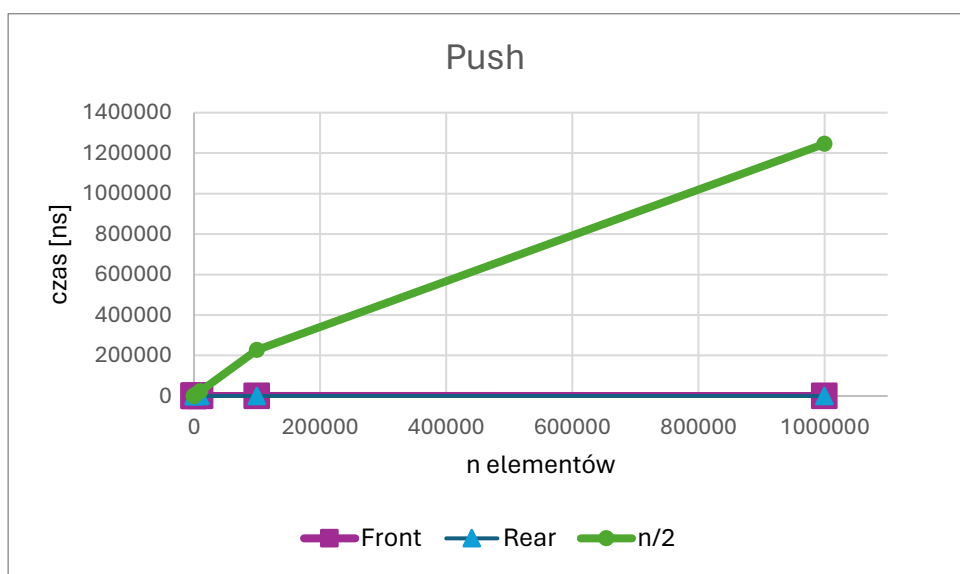
**Dokładna implementacja znajduje się w repozytorium:
<https://github.com/AbaturDev/Struktury-Danych>**

3.1.1 Push - Dodanie elementu

Badanie wydajności operacji dodawania elementu o określonym priorytecie do kolejki. Przeprowadzone zostały badania dla różnych przypadków wartości priorytetu dodawanego elementu. Kiedy priorytet jest wyższy niż pierwszego elementu – dodanie na początek kolejki, kiedy priorytet jest niższy niż ostatniego elementu – dodanie na koniec kolejki. Zbadano również przypadek dodania do „środka” kolejki, priorytet dodawanego elementu umiejscowi go w $\frac{n}{2}$ miejscu kolejki.

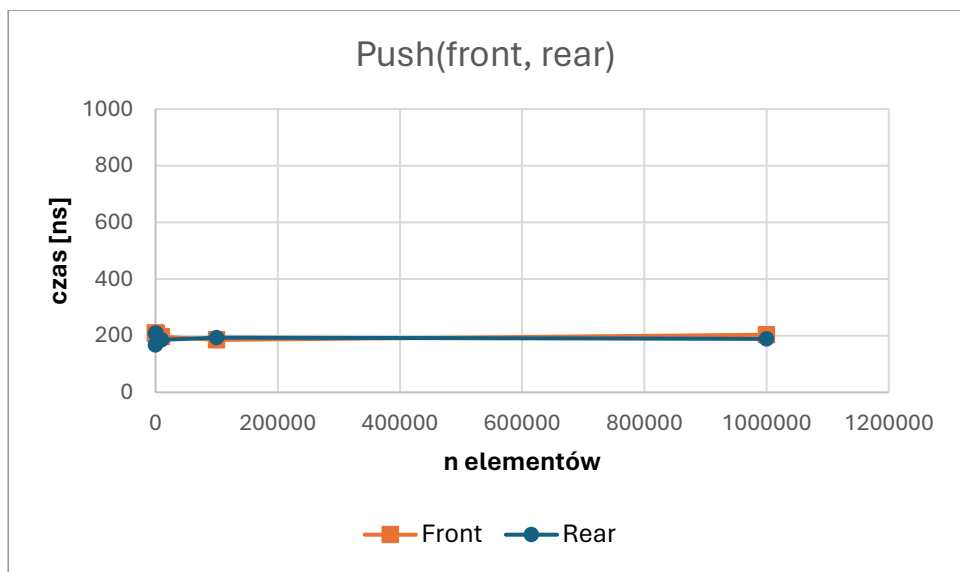
Dodanie elementu					
Początek(front)		Środek(n/2)		Koniec(rear)	
n	t [ns]	n	t [ns]	n	t [ns]
10	208	10	333	10	208
100	209	100	625	100	167
1000	209	1000	2985	1000	209
10000	197	10000	23041	10000	185
100000	185	100000	227542	100000	193
1000000	203	1000000	1245959	1000000	189

Tabela 1 Wyniki operacji Push



Rysunek 1 Charakterystyka złożoności czasowej dla operacji Push

Z racji tego, że wyniki czasowe operacji dodawania na początku i końcu są niskie poniżej przedstawiono osobny, dokładny wykres dla tych operacji.



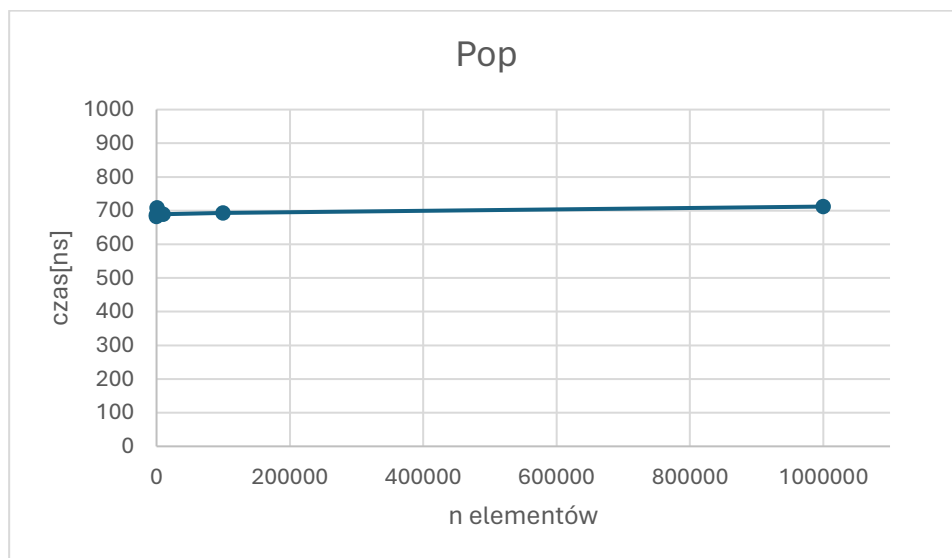
Rysunek 2 Dokładana charakterystyka czasowa dla dodawania na początku i koniec

3.1.2 Pop - Usunięcie i zwrócenie elementu

Badanie wydajności operacji usunięcia i zwrócenia wartości elementu o najwyższym priorytecie.

Usunięcie elementu	
n	t [ns]
10	683
100	687
1000	708
10000	689
100000	693
1000000	712

Tabela 2 Wyniki dla operacji Pop



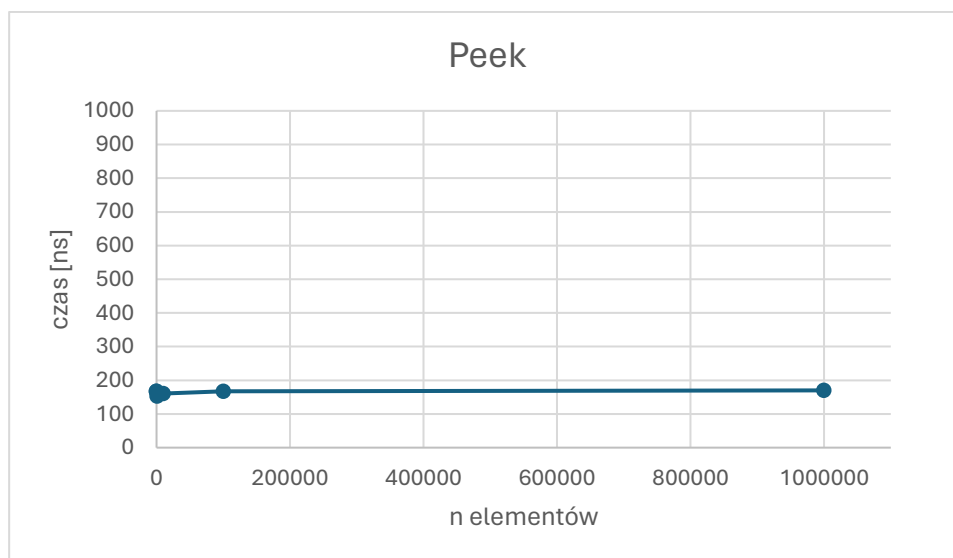
Rysunek 3 Charakterystyka złożoności czasowej dla operacji Pop

3.1.3. Peek - Podejrzenie elementu

Badanie wydajności operacji „podejrzenia”, czyli zwrócenia elementu o najwyższym priorytecie bez usuwania go z kolejki.

Podejrzenie elementu	
n	t [ns]
10	167
100	167
1000	153
10000	161
100000	167
1000000	170

Tabela 3 Wyniki dla operacji Peek



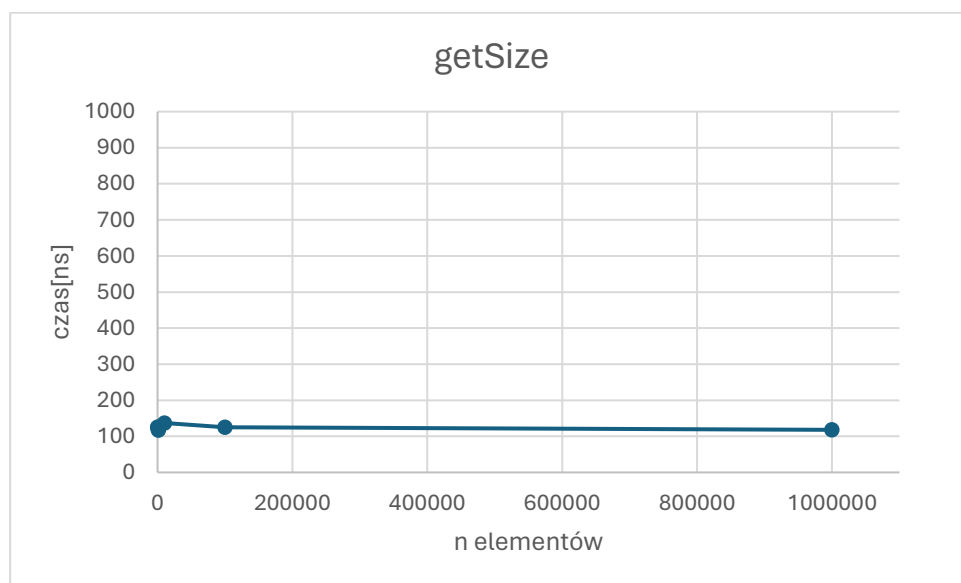
Rysunek 4 Charakterystyka złożoności czasowej dla operacji Peek

3.1.4 GetSize – Zwrócenie rozmiaru

Badanie wydajności operacji zwrócenia rozmiaru kolejki.

Zwrócenie rozmiaru	
n	t [ns]
10	125
100	125
1000	117
10000	137
100000	125
1000000	118

Tabela 4 Wyniki dla operacji getSize



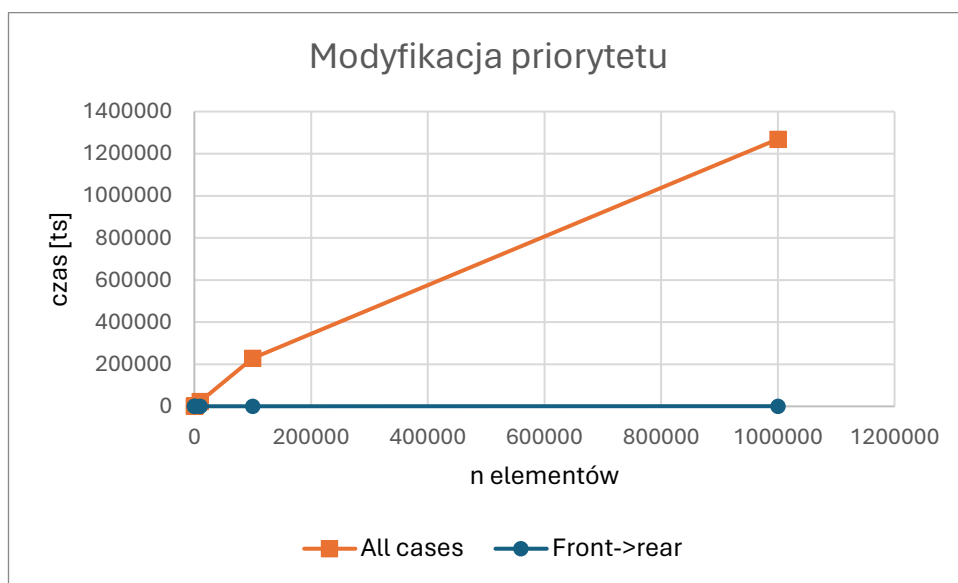
Rysunek 5 Charakterystyka złożoności czasowej dla operacji getSize

3.1.5 Change – Modyfikacja priorytetu

Badanie wydajności operacji modyfikacji priorytetu wybranego elementu kolejki priorytetowej. Operacja polegała na zmianie priorytetu wybranego elementu kolejki, co wiązało się z koniecznością znalezienia elementu, modyfikacji jego priorytetu i przeniesienia go w odpowiednie miejsce w kolejce. Zbadano przypadki zmiany priorytetu pierwszego elementu (najwyższy priorytet) na najniższy – przeniesienie na koniec kolejki. Zbadano również przypadek zmiany priorytetu dowolnego elementu, na rzecz badań był to element umiejscowiony w środku kolejki priorytetowej.

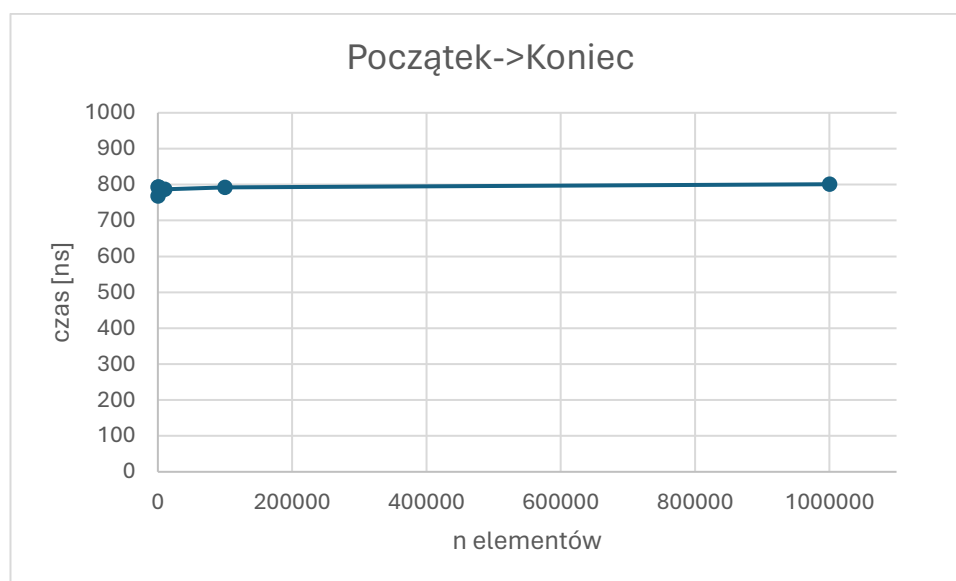
Modyfikacja priorytetu			
Początek->Koniec		Každy inny przypadek	
n	t [ns]	n	t [ns]
10	793	10	791
100	768	100	1236
1000	793	1000	3000
10000	787	10000	23167
100000	792	100000	228333
1000000	801	1000000	1268958

Tabela 5 Wyniki dla operacji modyfikacji priorytetu



Rysunek 6 Charakterystyka złożoności czasowej dla operacji modyfikacji priorytetu

Z racji tego, że wyniki czasowe operacji modyfikacji priorytetu z początku na koniec są niskie poniżej przedstawiono osobny, dokładny wykres dla tej operacji.



Rysunek 7 Dokładna charakterystyka złożoności czasowej dla operacji modyfikacji priorytetu początek->koniec

3.1.6 Złożoność obliczeniowa

Następnie przeanalizowano złożoność wyników zgodnie z notacją dużego O.

Operacja	Złożoność
Pop	$O(1)$
Peek	$O(1)$
getSize	$O(1)$
Push - optymistyczne	$O(1)$
Push - średnio	$O(n)$
Change - optymistyczne	$O(1)$
Change - średnio	$O(n)$

Tabela 6 Analiza wyników operacji zgodnie z notacją dużego O

Wyniki otrzymane po przebadaniu zostały porównane z wynikami z literatury naukowej.

<i>Operation</i>	<i>Unsorted List</i>	<i>Sorted List</i>
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Tabela 7 Wyniki z literatury naukowej (naszej implementacji dotyczy przypadek posortowany)

Kolejka priorytetowa (5)

- ▶ Możemy również odwrócić koncepcję – dodajemy elementy od razu w potrzebne miejsce (jak insert sort), wtedy lista jest posortowana i największy element będzie zawsze na początku.
- ▶ Operacje:
 - ▶ $\text{insert}(e,p)$ – za pomocą wyszukania odpowiedniego miejsca i wstawienia, czas $O(n)$.
 - ▶ $\text{extract-max}()$ – zwracamy i usuwamy pierwszy element, czas $O(1)$ (amortyzowany lub nie).
 - ▶ $\text{peek}()$ – analogicznie do $\text{extract-max}()$, ale bez usuwania, czas $O(1)$ (amortyzowany lub nie).
 - ▶ $\text{modify-key}(e,p)$ – znalezienie elementu, modyfikacja jego priorytetu i przeniesienie elementu w odpowiednie miejsce, czas $O(n)$.

SD 4: Kolejki

dr inż. Jacek Rudy

Rysunek 8 Wyniki przedstawione na wykładzie

Nasze wyniki potwierdzają oczekiwania teoretyczne i są zgodne z wynikami opisanymi w literaturze. Niektóre przebadane przypadki są korzystniejsze na rzecz naszej implementacji ze względu na usprawnianie niektórych przypadków w operacjach takich jak modyfikacja priorytetu czy dodanie elementu, lecz są to przypadki optymistyczne.

3.2 Kopiec

Aby ocenić wydajności implementacji kolejki priorytetowej za pomocą kopca maksymalnego, przeprowadzono serię testów funkcji głównych kolejki priorytetowej. Testy zostały przeprowadzone na różnych rozmiarach kolejki, które zwiększały się logarytmicznie o 10-krotność dla każdego kroku. Zakres wielkości kolejki priorytetowej, na którym przeprowadzono badania, wynosił od 10 do 1 000 000 elementów. Wydajność badanych operacji została zmierzona w nanosekundach (ns).

Dokładna implementacja znajduje się w repozytorium:
<https://github.com/farfek/Data-dstructures-2>

3.2.1 Insert

Insert – dodawanie elementu o określonym priorytecie. Ze względu na implementację opartą na dynamicznym alokowaniu pamięci uwzględniłem badania 3 głównych przypadków:

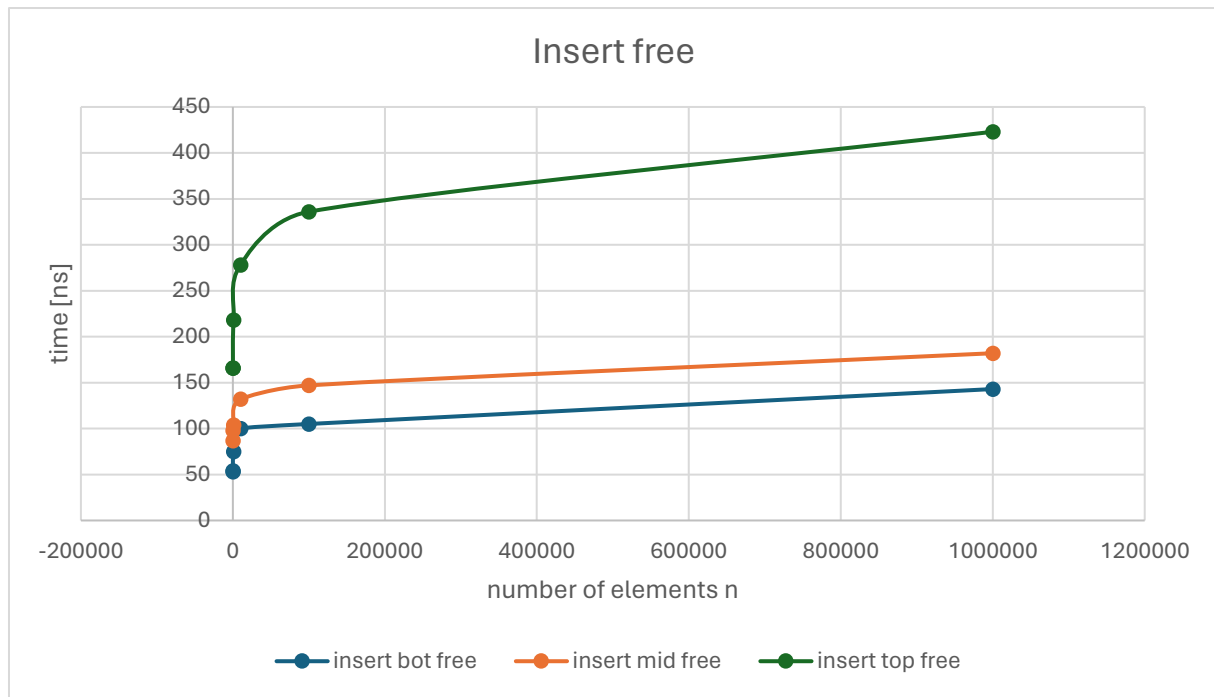
wprowadzania elementu o priorytecie takim, że element kończy na:

- samym dole kopca
- środku kopca
- szczycie kopca

Dla przypadku, gdy trzeba zaalokować więcej pamięci oraz gdy nie ma takiej potrzeby.

Dodanie elemntu w przypadku kopca z wolnym miejscem					
insert bot free		insert mid free		insert top free	
n	t [ns]	n	t [ns]	n	t [ns]
10	54	10	87	10	166
100	53	100	98	100	166
1000	75	1000	104	1000	218
10000	100	10000	132	10000	278
100000	105	100000	147	100000	336
1000000	143	1000000	182	1000000	423

Tabela 8 Wyniki dla operacji Insert free space



Rysunek 9 Charakterystyka złożoności czasowej dla operacji Insert free space

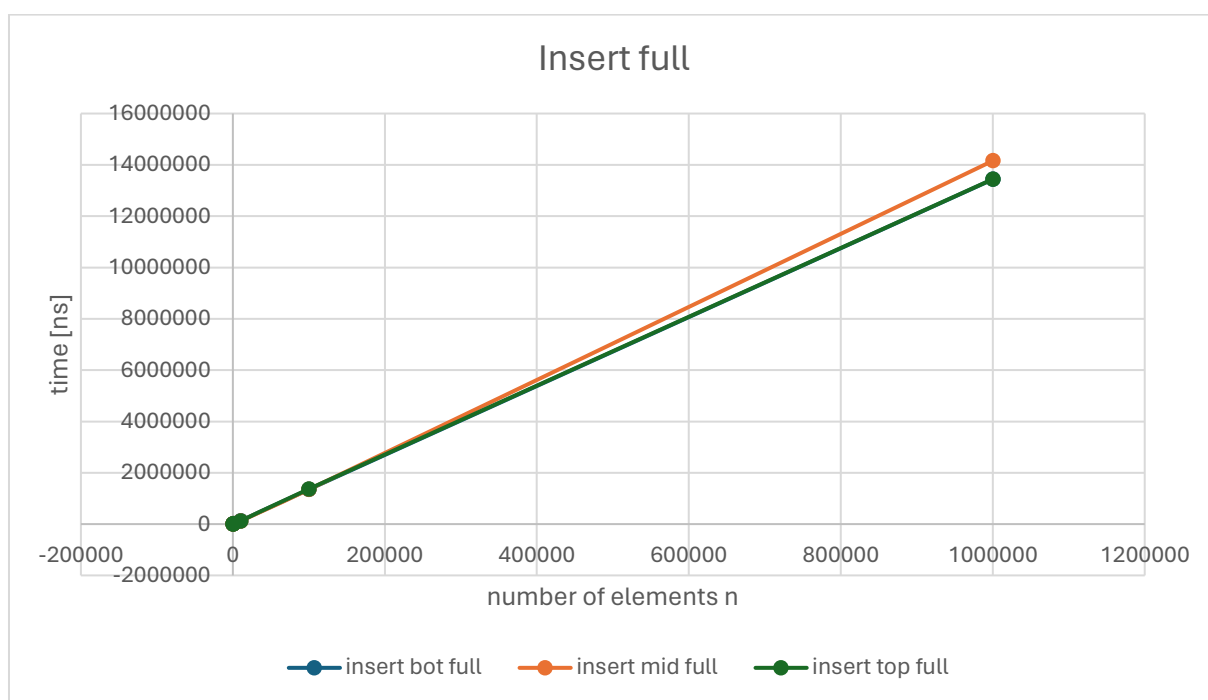
Według kodu, złożoność obliczeniowa powinna wynosić:

- a) Insert bot free – $O(1)$
- b) Insert mid free – $O(\frac{\log(n)}{2})$
- c) Insert top free – $O(\log(n))$

Na wykresie można zaobserwować, że dla insert bot free, złożoność z badań nie wynosi $O(1)$. Jest to spowodowane małą precyznością pomiarową. Działania te są tak krótkie, że większość czasu mierzonego to dostęp do pamięci systemu w przypadku wielu elementów (widać skok wartości czasu im więcej jest używanych elementów) oraz innymi losowymi wydarzeniami występującymi w komputerze. Jednak pomimo tego, przyjmuję że złożoność wynosi $O(1)$, gdyż są to bardzo małe wartości rzędów 50-150 nanosekund. Pozostałe funkcje wyglądają na zgodne z oczekiwaną złożonością z kodu.

Dodanie elementu w przypadku kopca z zajęтым miejscem					
insert bot full		insert mid full		insert top full	
n	t [ns]	n	t [ns]	n	t [ns]
10	696	10	1378	10	939
100	1678	100	3374	100	2710
1000	12633	1000	17189	1000	14851
10000	125005	10000	119164	10000	116503
100000	1351090	100000	1349850	100000	1367730
1000000	13434600	1000000	14157000	1000000	13440300

Tabela 9 Wyniki dla operacji Insert no space



Rysunek 10 Charakterystyka złożoności czasowej dla operacji Insert no space

Według kodu, złożoność obliczeniowa powinna wynosić:

- Insert bot free – $O(n)$
- Insert mid free – $O(n + \frac{\log(n)}{2})$
- Insert top free – $O(n + \log(n))$

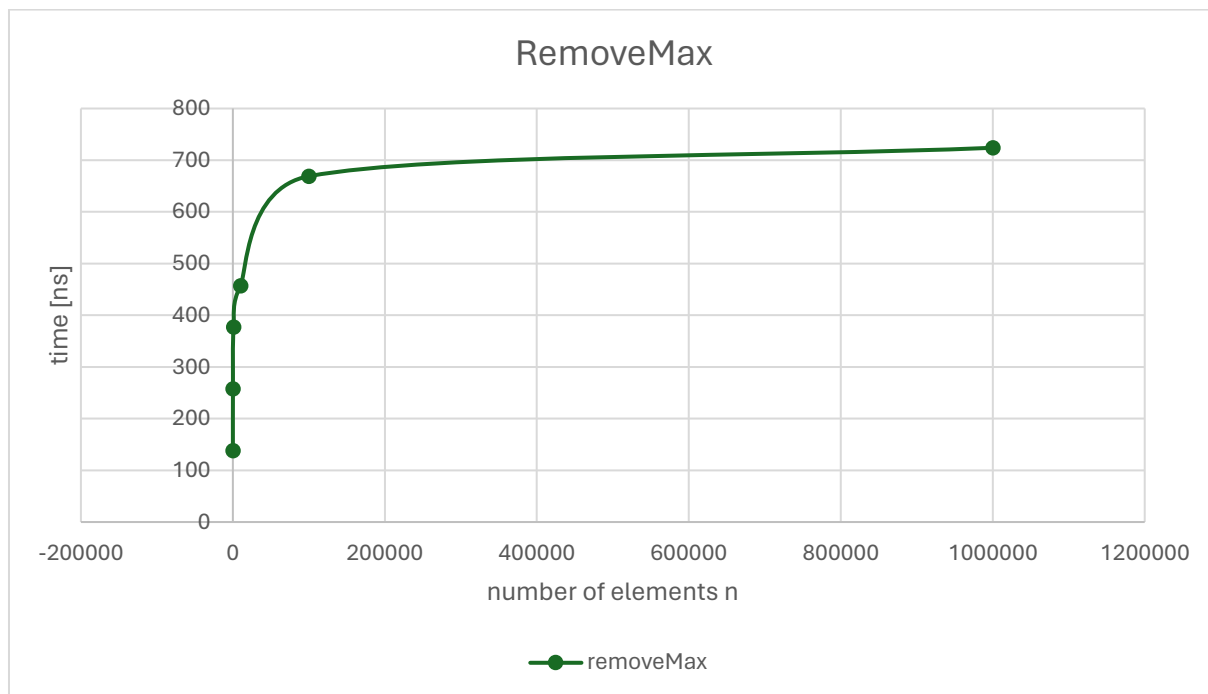
Wszystkie złożoności z wykresu są zgodne z oczekiwaniami.

3.2.2 RemoveMax

RemoveMax – usunięcie elementu znajdującego się na szczycie kolejki oraz zwrócenie jego wartości.

removeMax	
n	t [ns]
10	138
100	258
1000	377
10000	457
100000	669
1000000	724

Tabela 10 Wyniki dla operacji RemoveMax



Rysunek 11 Charakterystyka złożoności czasowej dla operacji RemoveMax

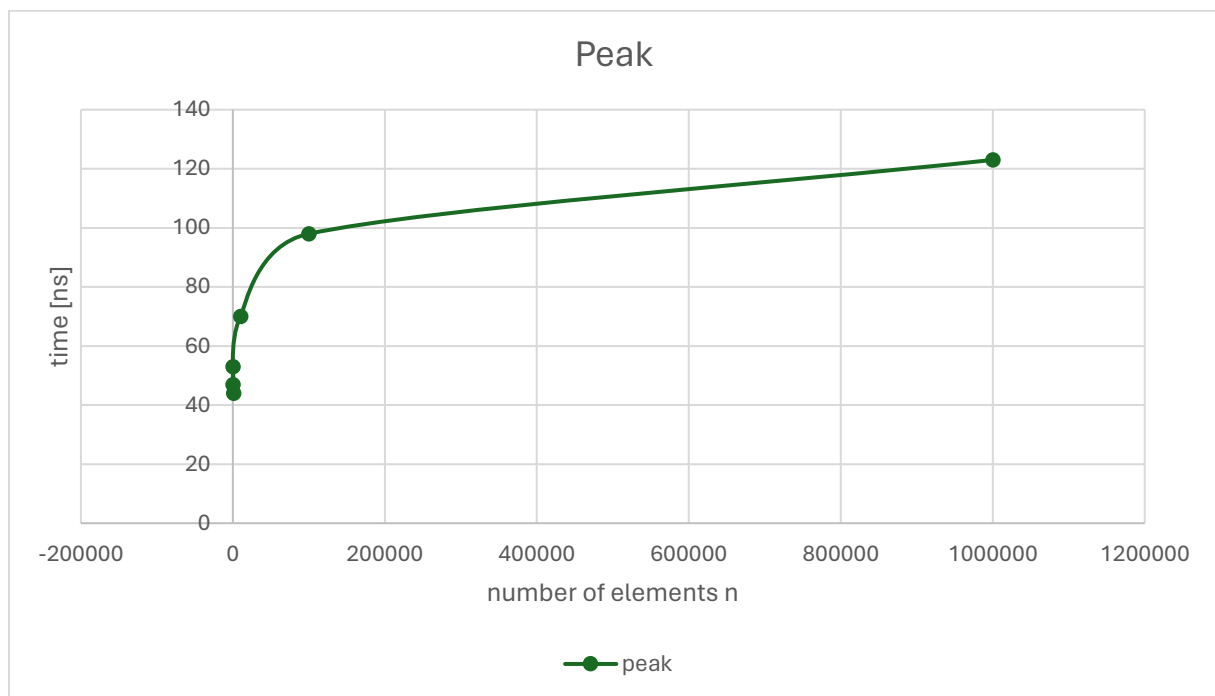
Według kodu, złożoność obliczeniowa powinna wynosić $O(\log(n))$. Jest to zgodne z wykresem.

3.2.3 Peak

Peak – zwrócenie wartości elementu na szczycie kopca, bez jego usuwania.

peak	
n	t [ns]
10	53
100	47
1000	44
10000	70
100000	98
1000000	123

Tabela 11 Wyniki dla operacji Peak



Rysunek 12 Charakterystyka złożoności czasowej dla operacji Peak

Według kodu, złożoność obliczeniowa powinna wynosić $O(1)$.

Podobnie jak w badaniach inserta, przy tak małych wartościach czasu, przez małą precyzyjność pomiarów przyjmuję, że jest to zgodne z wykresem.

3.2.4 ReturnSize

ReturnSize – zwrócenie aktualnej ilości elementów kopca.

returnSize	
n	t [ns]
10	48
100	62
1000	62
10000	57
100000	66
1000000	93

Tabela 12 Wyniki dla operacji ReturnSize



Rysunek 13 Charakterystyka złożoności czasowej dla operacji ReturnSize

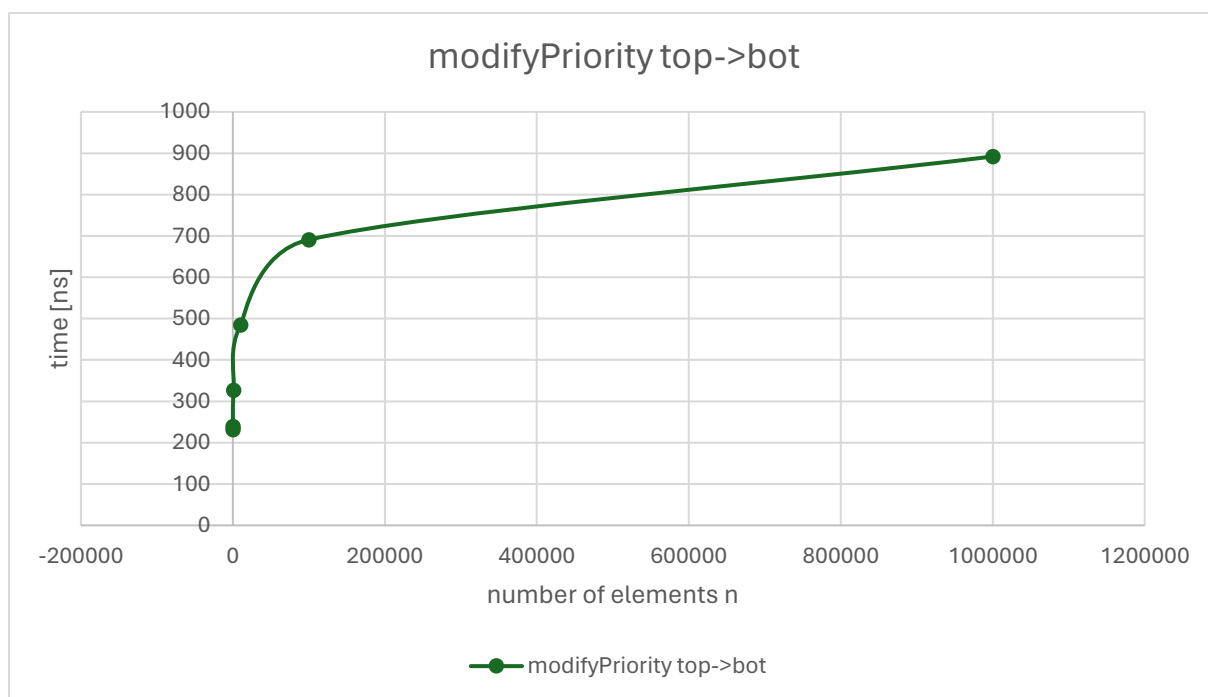
Według kodu, złożoność obliczeniowa powinna wynosić $O(1)$. Podobnie jak w badaniach inserta, przy tak małych wartościach czasu, przez małą precyzyjność pomiarów przyjmuję, że jest to zgodne z wykresem.

3.2.5 ModifyPriority

ModifyPriority – zmiana priorytetu elementu o podanej wartości w argumentach dla przypadku zmiany z góry na dół oraz innych przypadków.

modifyPriority top->bot	
n	t [ns]
10	232
100	239
1000	327
10000	485
100000	691
1000000	892

Tabela 13 Wyniki dla operacji ModifyPriority top->bot

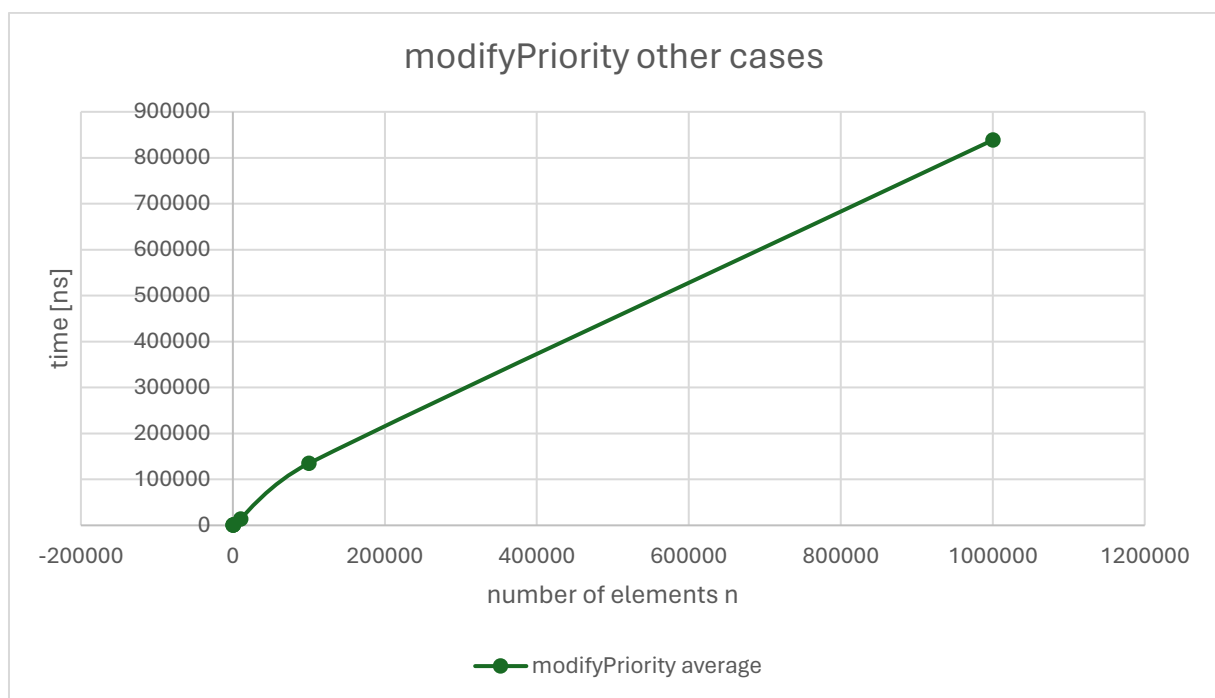


Rysunek 14 Charakterystyka złożoności czasowej dla operacji ModifyPriority top -> bot

Według kodu, złożoność obliczeniowa powinna wynosić $O(\log(n))$. Jest to zgodne z wykresem.

modifyPriority average	
n	t [ns]
10	253
100	383
1000	1419
10000	13925
100000	134832
1000000	838461

Tabela 14 Wyniki dla operacji ModifyPriority average case



Rysunek 15 Charakterystyka złożoności czasowej dla operacji ModifyPriority average case

Według kodu, złożoność obliczeniowa powinna wynosić $O(n + \log(n))$. Jest to zgodne z wykresem.

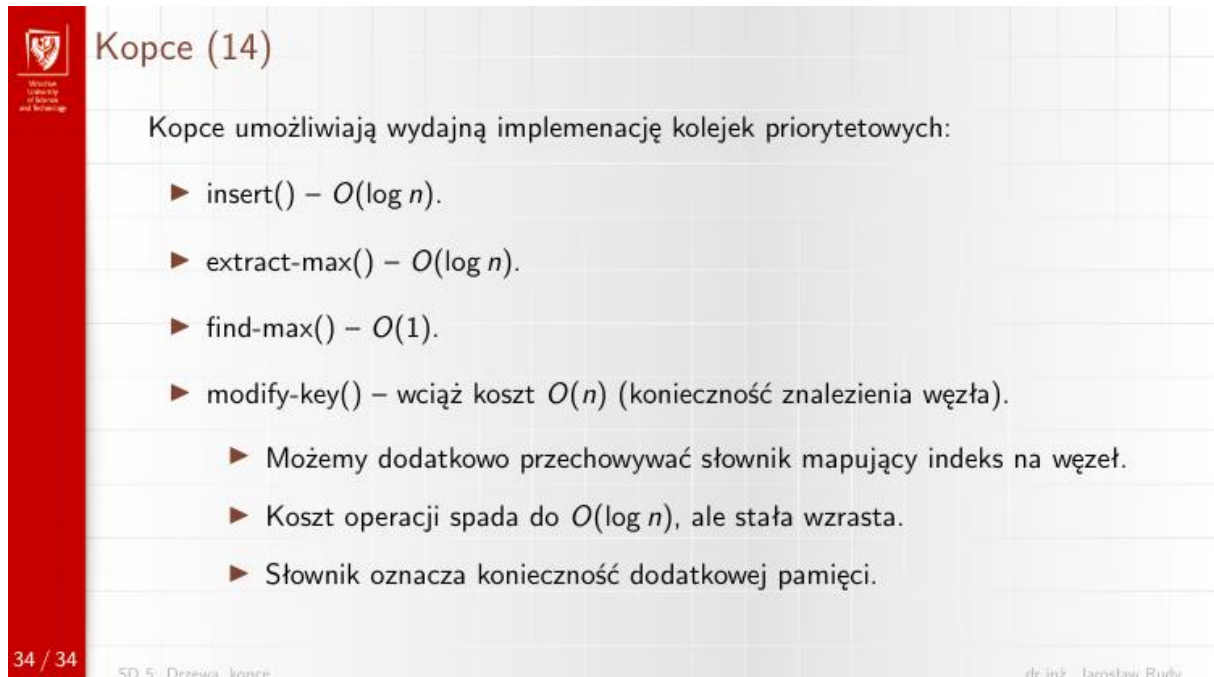
3.2.6 Analiza złożoności obliczeniowej

Z powyższych wykresów oraz analizy kodu, otrzymałem następujące wyniki:

Operation	O(n) notation
Insert free optimistic	$O(1)$
Insert free pessimistic	$O(\log(n))$
Insert full optimistic	$O(n)$
Insert full pessimistic	$O(n + \log(n))$
Remove Max	$O(\log(n))$
Peak	$O(1)$
Return Size	$O(1)$
Modify Priority top->bot	$O(\log(n))$
Modify Priority average	$O(n + \log(n))$

Tabela 15 Zestawienie złożoności obliczeniowej dla kolejki opartej na kopcu

Wyniki te zostały porównane z literaturą:



Kopce (14)

Kopce umożliwiają wydajną implementację kolejek priorytetowych:

- ▶ $\text{insert}()$ – $O(\log n)$.
- ▶ $\text{extract-max}()$ – $O(\log n)$.
- ▶ $\text{find-max}()$ – $O(1)$.
- ▶ $\text{modify-key}()$ – wciąż koszt $O(n)$ (konieczność znalezienia węzła).
 - ▶ Możemy dodatkowo przechowywać słownik mapujący indeks na węzeł.
 - ▶ Koszt operacji spada do $O(\log n)$, ale stała wzrasta.
 - ▶ Słownik oznacza konieczność dodatkowej pamięci.

Wyniki pokrywają się z literaturą. Modify Priority average case $O(n + \log(n))$ można uprościć do $O(n)$, więc ten wynik również się zgadza.

<i>Operation</i>	<i>Time</i>
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Rysunek 17 Złożoności obliczeniowe kolejki priorytetowej opartej na kopcu z literatury (w tym przypadku kopca min)

Wyniki pokrywają się z literaturą. Kopiec min działa na tej samej zasadzie co kopiec max. Modify Priority average case $O(n + \log(n))$ można uprościć do $O(n)$, więc ten wynik również się zgadza.

4 Porównanie

Porównanie złożoności obliczeniowych dla przebadanych operacji kolejki priorytetowej zaimplementowanych za pomocą kopca i listy wiązanej.

Operacja	Lista	Kopiec
Pop/RemoveMax	$O(1)$	$O(\log n)$
Peek	$O(1)$	$O(1)$
getSize	$O(1)$	$O(1)$
Push/Insert - optymistyczne	$O(1)$	$O(1)$
Push/Insert - średnio	$O(n)$	$O(\log n)$
Change - optymistyczne	$O(1)$	$O(\log n)$
Change - średnio	$O(n)$	$O(\log n)$

Tabela 16 Porównanie złożoności obliczeniowych operacji dla różnych implementacji

- Wyniki dla operacji takich jak Peek, getSize wypadają dla obu implementacji tak samo, złożoność obliczeniowa jest stała $O(1)$.
- Dla operacji Pop/RemoveMax odpowiadającej za zwrócenie i usunięcie elementu z najwyższym priorytetem lepiej wypada lista wiązana, która dla tej operacji ma złożoność obliczeniową $O(1)$ do $O(\log n)$ dla kopca.
- Złożoność obliczeniowa operacji Push/Insert, odpowiadająca za dodanie elementu do kolejki, w optymistycznym przypadku dla obu implementacji jest taka sama, czyli stała $O(1)$. Natomiast średnio można zaobserwować zdecydowaną przewagę kopca, który dla tej operacji ma złożoność $O(\log n)$ do $O(n)$ dla listy.
- Operacja Change, która na celu ma zmianę priorytetu wybranego elementu, wypada średnio korzystniej dla kopca, który ma złożoność obliczeniową dla tej operacji $O(\log n)$ do $O(n)$ dla listy. Lista w optymistycznym przypadku, w którym zmieniamy priorytet elementu z najwyższym priorytetem na najniższy, wypada lepiej ze stałą złożonością $O(1)$.

5 Wnioski

- a) Wybór implementacji: Przeprowadzone badania wykazały, że wybór implementacji kolejki priorytetowej zależy od konkretnych potrzeb i oczekiwań co do wydajności poszczególnych operacji. Lista wiązana sprawdza się dobrze w przypadku operacji Pop/RemoveMax, gdzie ma stałą złożoność obliczeniową $O(1)$, jednak w przypadku operacji Push/Insert i Change, kopiec wykazuje się lepszą wydajnością, szczególnie przy większych kolejkach.
- b) Złożoność obliczeniowa: Warto zauważyć, że złożoność operacji różni się w zależności od implementacji. Na przykład operacja Push/Insert średnio ma złożoność $O(n)$ dla listy, a $O(\log n)$ dla kopca. Podobnie, operacja Change średnio ma złożoność $O(n)$ dla listy, ale $O(\log n)$ dla kopca.
- c) Stabilność czasowa: W przypadku kopca, czas działania operacji nie zmienia się znacząco wraz ze wzrostem liczby elementów w kolejce, dzięki czemu zachowuje stabilność wydajnościową nawet dla bardzo dużych danych. Natomiast w przypadku listy, niektóre operacje mogą stać się wyraźnie wolniejsze dla większych rozmiarów kolejki.
- d) Optymalizacje: Istnieją optymalizacje, które mogą poprawić wydajność operacji dla obu implementacji. Na przykład dla listy wiązanej można stosować techniki takie jak równoważenie drzewa, co może zmniejszyć czas wykonywania operacji. Dla kopca, optymalizacje związane z alokacją pamięci mogą przyspieszyć działanie.
- e) Pamięć: Kopiec maksymalny jest zaimplementowany za pomocą tablicy, gdzie każdy element wymaga tylko jednej jednostki pamięci, co oznacza, że zajmuje on mniej dodatkowej pamięci niż lista wiązana, która wymaga dodatkowych wskaźników dla każdego węzła.
- f) Lepszą implementacją kolejki priorytetowej jest kopiec. Sprawdza się on w zdecydowanie większej ilości przypadków lepiej niż lista.

6 Bibliografia

- Wprowadzenie do algorytmów Cormen Thomas H., Leiserson Charles E., Rivest Ronald L, Clifford Stein ISBN 978-83-01-16911-4
- Data Structures and Algorithms in C++, 2nd Edition Michael T. Goodrich, Roberto Tamassia, David M. Mount ISBN: 978-0-470-38327-8
- Wykład 4 – Kolejki, Jarosław Rudy
<http://jaroslaw.rudy.staff.iiar.pwr.wroc.pl/files/sd/w4.pdf>