Politechnika Wrocławska

Wydział Informatyki
i Telekomunikacji

STRUKTURY DANYCH

Projekt nr 2

Wykonali:
Filip SANOWSKI - 280062
Kacper KRUSZELNICKI - 280150

Prowadzący:
mgr. inż Piotr NOWAK

7 maja 2025

Spis treści

link do repozytorium GitHub z pełną implementacją

1 Wstęp

Kolejka priorytetowa to abstrakcyjna struktura danych, która przypomina zwykłą kolejke (FIFO), lecz rózni się tym, że każdy element tej kolejki ma przypisany priorytet. Element o najwyższym priorytecie (lub w przypadku min-heap najniższym) jest usuwany jako pierwszy - niezależnie od kolejności dodania. Podstawowe operacje na kolejkach to dodawanie elementów z określonym priorytetem (*insert/push*), zwracanie elementu o najwyższym priorytecie lub też następnego w kolejce bez usuwania (*peek/top*), usuwanie elementu o najwyższym priorytecie (*pop*), zmiania priorytetu (*changePriority*) oraz zwrot informacji o rozmiarze kolejki (*size*). Kolejkę priorytetową można zastosować w algorytmach grafowych, kolejkach zadań w SO, symulacji zdarzeń oraz jako kolejkę zdarzeń w grafach i aplikacjach czasu rzeczywistego. W naszych testach przygotowalismy dwie struktury dnaych: posortowaną listę wiązaną oraz kopiec binarny (min-heap) i zaimplementowaliśmy na nich kolejke priorytetową.

2 Implementacja

2.1 Lista wiązana

Moja implementacja opiera się na posortowanej liście wiązanej, gdzie każdy węzeł zawiera *wartość*, *priorytet elementu*, *wskaźnik na kolejny węzeł*. Węzły są wstawiane w kolejności malejącej względem priorytetu, więc element o najwyższym priorytecie (czyli najmniejszym numerze, np. 1) znajduje się na początku listy. metoda *push* wstawia element w odpowiednie miejsce w liście, aby zachować porządek względem priorytetu. jeśli nowy priorytet jest większy od obecengo head-a, staje się nowym head-me. Jeśli lista jest pusta, nowy element zosatje jednocześnie head i end. *Pop* usuwa pierwszy element, co odpowiada usnięciu lementu o najwyższym priorytecie. Metoda *changeP* przeszukuje listę w celu znalezienia węzła o podanej wartości. Usuwa go, a następnie ponownie wstawia z nowym priorytetem. Ta struktura jest dobra dla niewielkich zbiorów danych, lecz wolniej dodaje, ponieważ wymaga przeglądania listy w celu znalezienia miejsca. Nie wspiera wyszukiwania pzoycji w O(1), przez co *changeP* jest kosztowne oraz przez to żee każdy węzeł wyamaga osobnej alokacji i wskaźnika są problemy z zajmowaniem zbyt dużej ilości pamięci. Złożoność operacji zależy od sposobu organizacji danych w liście. Operacja push ma złożoność O(n), ponieważ nowy element musi zostać wstawiony w odpowiednie miejsce według priorytetu, co wymaga liniowego przeszukiwania listy od początku aż do znalezienia właściwej pozycji. Operacja changeP również ma złożoność O(n), ponieważ polega na odnalezieniu danego elementu (co w najgorszym przypadku wymaga przeszukania całej listy), jego usunięciu, a następnie ponownym wstawieniu w odpowiednie miejsce – razem daje to łącznie złożoność rzędu O(n). Natomiast operacje pop, peek, size i isEmpty są bardzo szybkie – wykonują się w czasie stałym O(1) – ponieważ pop po prostu usuwa pierwszy element (head), peek odczytuje jego wartość, a size i isEmpty operują na prostych zmiennych.

Operacja	**Opis działania**	**Złożoność**
`push(wart, priorytet)`	Przechodzi listę, by wstawić element w odpowiednie miejsce	$O(n)$
`pop()`	Usuwa pierwszy element listy	$O(1)$
`peek()`	Odczyt wartości pierwszego elementu	$O(1)$
`changeP(wart, newPro)`	Szuka węzła, usuwa go, dodaje z nowym priorytetem	$O(n)$
`size()`	Licznik aktualizowany na bieżąco	$O(1)$
`isEmpty()`	Błyskawiczna kontrola pustki listy	$O(1)$

Tabela 1: Złożoność operacji w implementacji kolejki priorytetowej opartej na liście wiązanej

2.2 Kopiec

Implementacja kopca binarnego (min-heap) polega na tym, że użyto dynamicznej tablicy, która reprezentuje kompletne drzewo binarne (klasyczna i nawygodniejsza implementacja min-heap). Elementy przechwowywane sa jako obiekty *Element<T>*, które zwierają: wartość użytkową oraz liczbe określającą

piorytet (niższa = wyższy piorytet). Metody *heapifyUp* oraz *heapifyDown* służa do utrzymania własności kopca po dodaniu/usunięciu/modyfikacji priorytetu, w najgorsyzm przpadku mamy złożonośc obliczeniową O(log n). Metoda *push* dodaje nowy element na koniec wektora, a następnie przywraca porządek kopca przez *heapifyUp*. Metoda *pop* zastepuję korzeń ostatnim elementem i przywraca porządek przez *heapifyDown*. Metoda *changePriority* przeszukuje kopiec liniowo w celu odnalezienia elementu o danej wartości. Po aktualizacji priorytetu naprawia kopiec w odpowiednim kierunku. Złożonośc to O(n) wyszukiwanie O(log n) aktualizacja. Kolejka ma prostą i zrozumiałą strukturę, choć może zostac poddana optymalizacji *changePriority* przez utrzymanie mapy pozycji elementów. W tej strukturze dane są przechowywane w dynamicznej tablicy, która spełnia warunek kopca: dla każdego węzła jego priorytet jest nie większy niż priorytety jego dzieci. Dzięki temu dodanie (push) nowego elementu odbywa się poprzez dodanie go na końcu tablicy, a następnie przesuwanie w górę (heapifyUp) aż do zachowania warunku kopca – co zajmuje maksymalnie O(log n). Podobnie pop usuwa element z początku (korzenia), zamienia go z ostatnim, a następnie naprawia strukturę kopca poprzez przesuwanie w dół (heapifyDown) – również O(log n). Operacje peek, size i empty są natychmiastowe i wykonują się w czasie O(1), ponieważ dotyczą odczytu danych ze znanych indeksów lub prostych porównań. Operacja changePriority jest mniej efektywna, ponieważ najpierw wymaga przeszukania całego kopca, aby znaleźć dany element (co daje O(n)), a następnie naprawienia kopca przez heapifyUp lub heapifyDown (czyli O(log n)), co razem daje złożoność O(n + log n).

Operacja	**Opis działania**	**Złożoność**
`push(value, priority)`	Dodaje element i przywraca własność kopca przez `heapifyUp`	$O(\log n)$
`pop()`	Usuwa element o najwyższym priorytecie i przywraca kopiec (`heapifyDown`)	$O(\log n)$
`peek()`	Zwraca element o najwyższym priorytecie (pierwszy w kopcu)	$O(1)$
`size()`	Zwraca aktualną liczbę elementów w kopcu	$O(1)$
`empty()`	Sprawdza, czy kopiec jest pusty	$O(1)$
`changePriority()`	Szuka elementu liniowo, zmienia jego priorytet i naprawia kopiec	$O(n + \log n)$

Tabela 2: Złożoność operacji w kopcu (implementacja kolejki priorytetowej)

3 Badania

Badania wydajności operacji na kolejkach priorytetowych zostały przeprowadzone na komputerze o poniższej specyfikacji sprzętowej i programowej:

- **Procesor:** Intel(R) Core(TM) i5-14600K
- **Liczba rdzeni fizycznych:** 14
- **Liczba wątków logicznych:** 20
- **Pamięć operacyjna RAM:** 16 GB
- **System operacyjny:** Microsoft Windows 10 Home 64-bit
- **Wersja kompilatora:** Microsoft Visual C++ (MSVC)

3.1 Implementacja

W celu zbadania efektywności czasowej operacji na kolejkach priorytetowych, zaimplementowano dedykowany program testowy w języku C++. Testy wydajności przeprowadzono dla dwóch struktur: kopca binarnego oraz listy wiązanej. Do pomiaru czasu użyto biblioteki `<chrono>`, która pozwala na precyzyjne rejestrowanie czasu trwania operacji z dokładnością do nanosekund.

Każda operacja była testowana osobno, a pomiar był wykonywany w następujący sposób:

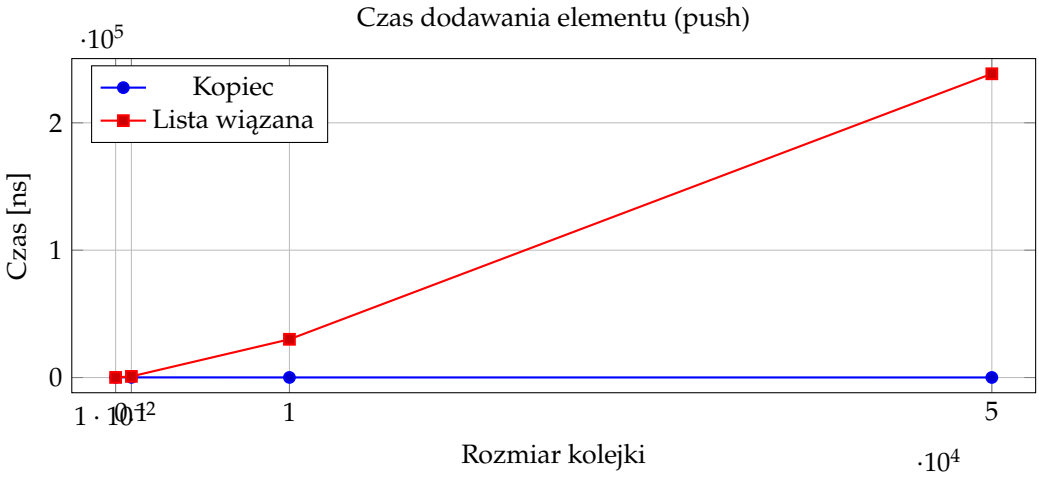
1. Dla ustalonego rozmiaru struktury (np. 10000 elementów) generowano losowe dane wejściowe (wartość i priorytet).

2. Wypełniano obie struktury tymi samymi elementami, aby zapewnić identyczne warunki testowe.
3. Operacja (np. `push`) była wykonywana 1000 razy, a czas każdej iteracji mierzono za pomocą zegara wysokiej rozdzielczości.
4. Po każdej iteracji struktura była przywracana do stanu wyjściowego (np. usuwano nowo dodany element).
5. Sumaryczny czas był dzielony przez liczbę powtórzeń, aby uzyskać średni czas wykonania jednej operacji.

Badania przeprowadzono dla następujących rozmiarów danych wejściowych: 100, 1000, 10 000 oraz 50 000 elementów.

Dzięki ujednoliconemu schematowi testowania oraz zastosowaniu identycznych danych wejściowych dla obu struktur, uzyskano porównywalne i miarodajne wyniki. Dane zostały zapisane do pliku `CSV`, a następnie przetworzone i przedstawione w formie wykresów i tabel przy użyciu pakietu `pgfplots` w systemie LaTeX.

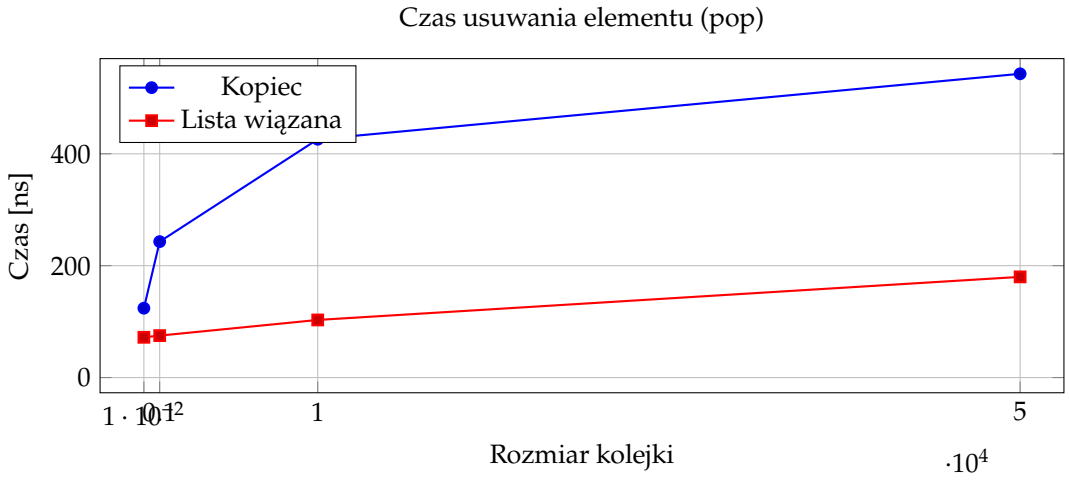
3.2 Dodawanie elementów



Rozmiar	Kopiec [ns]	Lista wiązana [ns]
100	163	90
1000	131	898
10000	84	30,020
50000	65	$2.38 \cdot 10^5$

Rysunek 1: Porównanie czasu dodawania elementów (push)

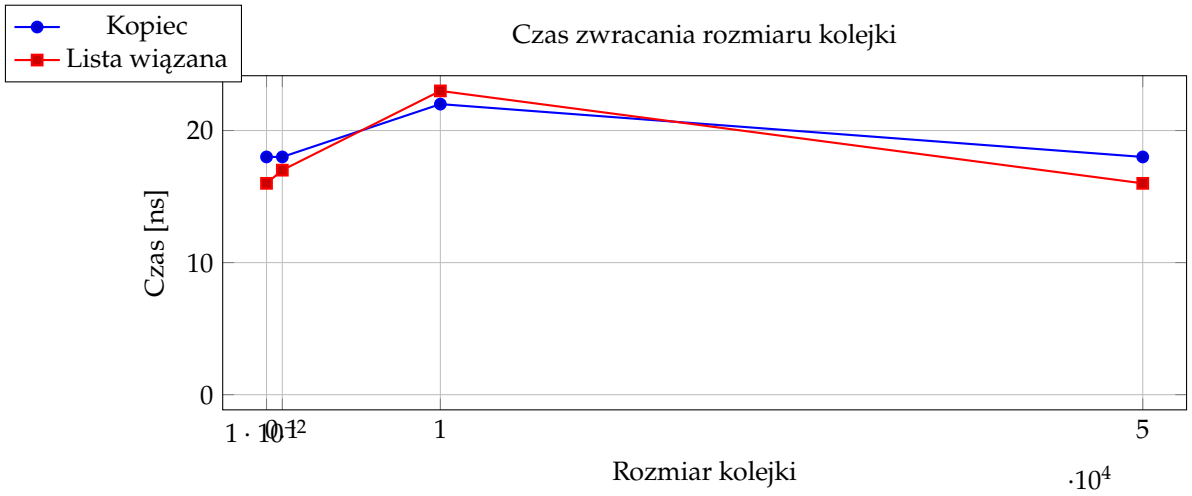
3.3 Usuwanie Elementu



Rozmiar	Kopiec [ns]	Lista wiązana [ns]
100	124	72
1000	243	75
10000	426	103
50000	543	180

Rysunek 2: Porównanie czasu usuwania elementów (pop)

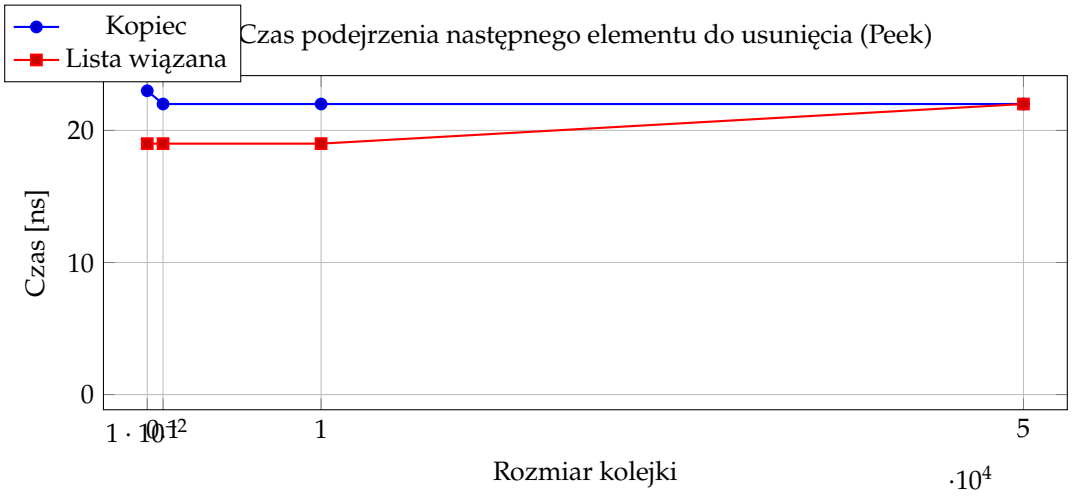
3.4 Zwracanie rozmiaru kolejki.



Rozmiar	Kopiec [ns]	Lista wiązana [ns]
100	18	16
1000	18	17
10000	22	23
50000	18	16

Rysunek 3: porównanie czasu zwracania rozmiaru kolejki

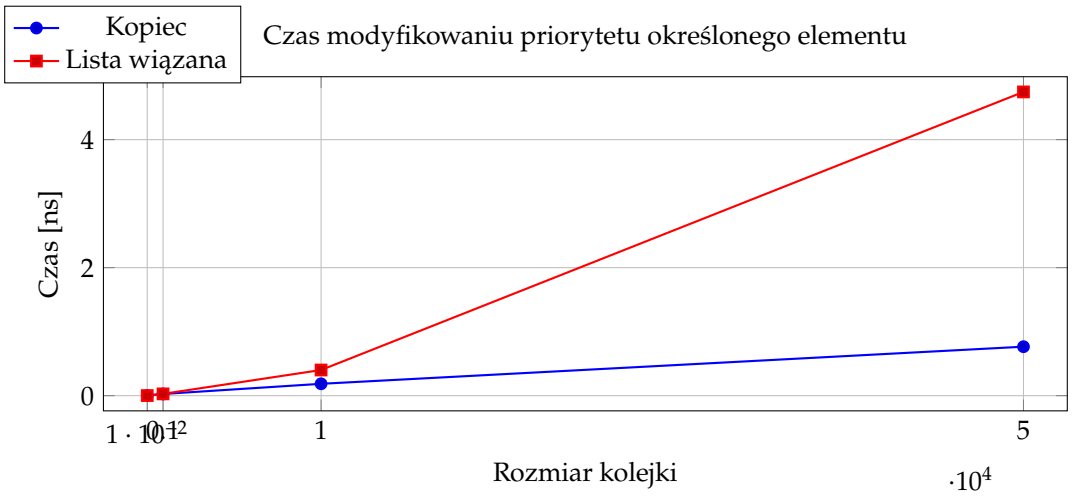
3.5 ”Peek” - podejrzenie następnego elementu do usunięcia



Rozmiar	Kopiec [ns]	Lista wiązana [ns]
100	23	19
1000	22	19
10000	22	19
50000	22	22

Rysunek 4: Porównanie czasu podejrzenie następntego elementu do usunięcia (Peek)

3.6 Modyfikacja priorytetu określonego elementu



Rozmiar	Kopiec [ns]	Lista wiązana [ns]
100	325	171
1000	2,652	2,801
10000	18,533	40,105
50000	76,511	$4.75 \cdot 10^5$

Rysunek 5: Porównanie czasu modyfikacji priorytetu określonego elementu

4 Wnioski

Na podstawie przeprowadzonych testów i analizy wyników zawartych w tabelach i wykresach można sformułować następujące wnioski dotyczące wydajności implementacji kolejki priorytetowej na dwóch

różnych strukturach danych: kopcu binarnym (min-heap) oraz liście wiązanej posortowanej malejąco względem priorytetu.

- **Dodawanie elementów (push):** Dla małych rozmiarów (np. 100 elementów) lista wiązana okazuje się nieco szybsza od kopca. Jednak przy większych rozmiarach (od 1000 wzwyż) kopiec zdecydowanie wygrywa, co potwierdza jego złożoność czasowa $O(\log n)$ wobec $O(n)$ w liście. Dla 50 000 elementów czas dodania w liście wiązanej był aż 3600 razy dłuższy niż w kopcu.
- **Usuwanie elementów (pop):** Dla małych rozmiarów lista wiązana radzi sobie bardzo dobrze, ale w większych przypadkach kopiec osiąga porównywalne wyniki. Kopiec niestety przegrywa z listą wiązaną ale różnice nie są jednak aż tak drastyczne jak przy dodawaniu.
- **Zwracanie rozmiaru (size) i podejrzenie elementu (peek):** Obie operacje wykonują się bardzo szybko i różnice czasowe są pomijalne. W obu strukturach są one zaimplementowane w czasie stałym $O(1)$.
- **Zmiana priorytetu (changePriority):** Ta operacja ujawnia największe różnice między strukturami. W liście wiązanej wymaga usunięcia elementu i ponownego wstawienia, co skutkuje ogromnym czasem dla dużych rozmiarów (prawie 0,5 ms dla 50 000 elementów). W kopcu operacja też nie jest trywialna, ale czas wykonania rośnie dużo wolniej.
- **Skalowalność:** Dla małych struktur (100 elementów) lista wiązana może być konkurencyjna. Jednak już dla 1000–10 000 elementów zaczyna przegrywać niemal we wszystkich aspektach. Kopiec binarny okazuje się znacznie bardziej stabilny i skalowalny.

Podsumowując, kopiec binarny jest znacznie lepszym wyborem dla większości zastosowań praktycznych, zwłaszcza przy dużych danych i wymagających zastosowaniach. Lista wiązana może być stosowana w prostych, małych przypadkach lub gdy istotna jest łatwość implementacji.

5 Źródła

- Wikipedia. (2024). *Kolejka priorytetowa*. Pozyskano z: `https://pl.wikipedia.org/wiki/Kolejka_priorytetowa`
- Wikipedia. (2024). *Kopiec (informatyka)*. Pozyskano z: `https://pl.wikipedia.org/wiki/Kopiec_(informatyka)`
- Wykłady profesora Jarosława Rudego