



# Politechnika Wrocławska

---

Wydział Elektroniki, Fotoniki i Mikrosystemów

Projektowanie algorytmów i metody sztucznej  
inteligencji

---

## Projekt 1

Zadanie na ocenę bdb

---

*Prowadzący:*

Dr inż. Łukasz Jeleń

*Wykonała:*

Zuzanna Mejer, 259382

Wrocław, 1 kwietnia 2022r.

# 1. Wstępne założenia

Zadanie polegało na zaprojektowaniu i zaimplementowaniu algorytmu, który radziłby sobie z sortowaniem otrzymanych pakietów składających się z numeru i tekstu wiadomości. Do wykonania zadania przyjęto pewne założenia. Pierwszym z nich jest przyjęcie perspektywy Anny i jej komputera. Oznacza to tyle, że napisany program nie zajmuje się dzieleniem wiadomości na  $n$  pakietów oraz nie nadaje im numerów, gdyż wymienione czynności wykonywał, zgodnie z poleceniem, Jan na swoim komputerze. Zatem program został przygotowany tak, żeby pracować już z losowo ułożonymi i ponumerowanymi pakietami. Drugim przyjętym założeniem jest przesyłanie tych pakietów w jednym pliku tekstowym jako drugi argument wywołania podczas uruchamiania pliku wykonywalnego. Kolejnym założeniem jest forma zakończenia programu i wyświetlenie posortowanej wiadomości. Zdecydowano, że posortowana wiadomość nie tylko będzie wyświetlana na standardowym wyjściu, ale również będzie zapisywana do nowo tworzonego pliku nazwanego „uporządkowany\_list.txt”.

# 2. Struktura danych

W celu zrealizowania zadania wybrano strukturę danych do przechowywania informacji (numeru i tekstu wiadomości), jaką jest kolejka priorytetowa na bazie listy dwukierunkowej. Poniżej przedstawiono uzasadnienie wyboru. Kolejka priorytetowa jest abstrakcyjnym typem danych, służącym do przechowywania zbioru elementów, przy czym każdy element posiada dodatkowe pole do przechowywania priorytetu (klucza). Zatem, z założenia, do kolejki priorytetowej można wprowadzać pakiety, składające się z numeru (priorytetu) i tekstu. Kolejnym argumentem przemawiającym za użyciem kolejki priorytetowej jest układanie elementów w kolejce kolejności priorytetu rosnąco. Lista dwukierunkowa umożliwia wstawianie elementu w dowolnym miejscu, nie tylko na początku lub na końcu listy, co pozwala na wydajne sortowanie elementów przy ich wstawianiu do kolejki. Analogicznie, lista dwukierunkowa pozwala także na usuwanie elementów z dowolnego miejsca.

# 3. Omówienie programu

## 3.1. Obsługa plików

Do obsługi plików została stworzona klasa o nazwie *file* z dwoma prywatnymi polami określającymi plik wejściowy *in\_file* oraz plik wyjściowy *out\_file*. Stworzone zostały metody służące do otwierania i zamykania plików wejściowych i wyjściowych (*open\_in\_file*, *open\_out\_file*, *close\_in\_file*, *close\_out\_file*), czytania (*read\_file*) i sprawdzania końca (*end\_of\_file*) pliku wejściowego oraz zapisywania do pliku wyjściowego (*write\_out\_file*).

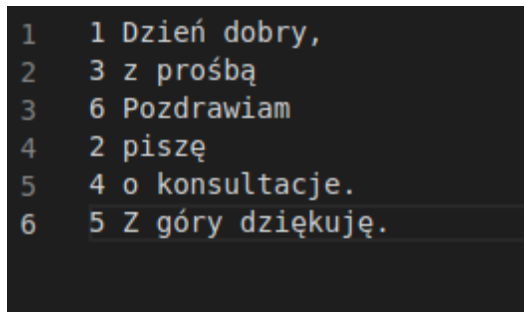
```
1 #ifndef OBSLUGA_PLIKOW.HH
2 #define OBSLUGA_PLIKOW.HH
3
4 #include <iostream>
5 #include <string>
6 #include <fstream>
7
8 class file
9 {
10     private:
```

```

11     std::fstream in_file;
12     std::fstream out_file;
13
14     public:
15         void open_in_file(std::string name_file);
16         void open_out_file(std::string name_file);
17         void read_file(int &key, std::string &text);
18         void write_out_file(const std::string &text);
19         void close_in_file();
20         void close_out_file();
21         bool end_of_file();
22     };
23 #endif

```

Plik wejściowy podawany przy uruchomieniu programu ma domyślnie formę przedstawioną na poniższym zdjęciu. Na początku linii jest numer pakietu, a po spacji wiadomość.



```

1  1 Dzień dobry,
2  3 z prośbą
3  6 Pozdrawiam
4  2 piszę
5  4 o konsultacje.
6  5 Z góry dziękuję.

```

Rysunek 1: Format pliku wejściowego

### 3.2. Kolejka priorytetowa jako lista dwukierunkowa

Do zaimplementowania kolejki na liście dwukierunkowej, została stworzona struktura *node*, która ma dwa pola do przechowywania pakietu: *key* oraz *text*, oraz dwa wskaźniki na następny i poprzedni element: *\*next* i *\*prev*. Została stworzona klasa *priority\_queue* z pierwszym wskaźnikiem na początek kolejki: *\*header* typu struktury *node*. Zostały zaimplementowane metody:

- *comparison* - służy do porównywania wartości dwóch kluczy i jest wykorzystywana przy sortowaniu elementów podczas wprowadzania do kolejki.
- *insert* - odpowiednik metody *push*; służy do wstawiania elementów do kolejki w dowolnym miejscu w kolejce, tak, aby tworzyć kolejkę uporządkowaną rosnąco.
- *display\_text* - odpowiednik metody *top*; odpowiada za wyświetlanie samego tekstu pierwszego elementu w kolejce.
- *remove\_minimum* - odpowiednik *pop*; służy do usuwania elementu z początku kolejki.
- *empty* - zwraca informację o tym, czy kolejka jest pusta.
- *return\_minimum* - jest funkcją pomocniczą do zwracania tekstu pierwszego elementu w kolejce. Wykorzystuje ją metoda zapisywania do pliku.

- *size* - zwraca ilość elementów umieszczonych w kolejce priorytetowej.

```
1 #ifndef PRIORITY_QUEUE.HH
2 #define PRIORITY_QUEUE.HH
3 #include <iostream>
4
5 struct node
6 {
7     int key;
8     std::string text;
9     node *prev = nullptr;
10    node *next = nullptr;
11 };
12
13 class priority_queue
14 {
15
16 private:
17     node *header = nullptr;
18     bool comparison(int x, int y);
19
20 public:
21     void insert(int key, std::string text);
22     void display_text();
23     void remove_minimum();
24     bool empty();
25     std::string return_minimum();
26     int size();
27 };
28 #endif
```

### 3.3. Działanie programu

Program wykonuje następujące działania:

1. Otwiera zarówno plik wejściowy, podany jako argument wywołania pliku wykonywalnego, jak i wyjściowy, do którego będą zapisywane dane.
2. Aż do natrafienia na koniec danych w pliku, program czytuje kolejne sekwencje, tworzy nowy element typu *node*, przypisuje do niego wartości klucza i wiadomości, a następnie wstawia w odpowiednie miejsce w kolejce, sortując rosnąco względem wartości klucza.
3. Kiedy nie ma więcej danych w pliku i kolejka jest już utworzona, następuje wyświetlenie tekstu elementu o najmniejszej wartości klucza na standardowe wyjście, zapisanie go do pliku wyjściowego oraz usunięcie go z kolejki priorytetowej. Ten etap powtarza się aż do wyczyszczenia całej kolejki.

```

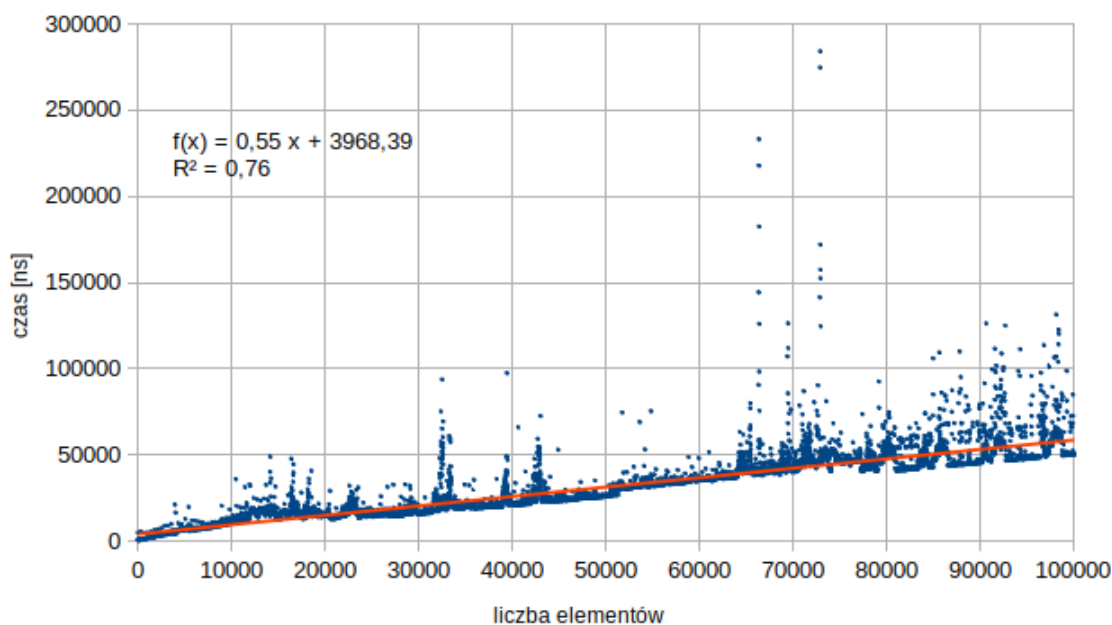
1  #include "obsługa_plikow.hh"
2  #include "priority_queue.hh"
3  #include <iostream>
4
5  int main(int argc, char *argv[])
6  {
7      priority_queue kolejka;
8      file plik;
9      int klucz;
10     std::string tekst;
11     plik.open_in_file(argv[1]);
12     plik.open_out_file("uporzadkowany_list");
13
14     while (!plik.end_of_file())
15     {
16         plik.read_file(klucz, tekst);
17         kolejka.insert(klucz, tekst);
18     }
19
20     while (!kolejka.empty())
21     {
22         kolejka.display_text();
23         plik.write_out_file(kolejka.return_minimum());
24         kolejka.remove_minimum();
25     }
26     std::cout << "\n";
27
28     plik.close_in_file();
29     plik.close_out_file();
30 }

```

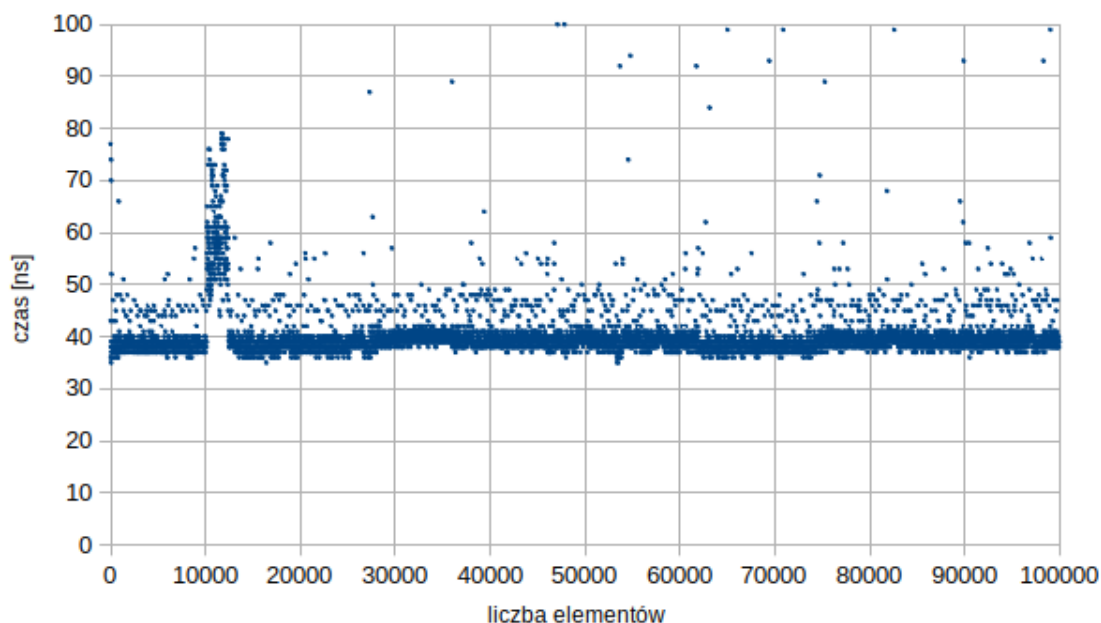
## 4. Analiza złożoności obliczeniowej

Najbardziej znaczące metody do badania złożoności obliczeniowej programu to metody *insert* oraz *remove\_minimum*. Ze względu na zastosowanie implementacji na liście posortowanej, można się spodziewać, że złożoność obliczeniowa metody *insert* będzie równa  $O(n)$ , co oznaczałoby, że czas potrzebny do wykonania się tej metody będzie liniowo zależny od liczby elementów w pliku. Z kolei złożoność obliczeniowa metody *remove\_minimum* powinna wynosić  $O(1)$ , co oznaczałoby, że metoda ta jest stała w czasie - niezależna od liczby elementów w kolejce.

Przeprowadzono testy, które miały potwierdzić lub odrzucić przedstawioną hipotezę. W tym celu wykorzystano bibliotekę *chrono* i funkcję *clock*. Zmierzono zależność czasu w nanosekundach od liczby elementów (ograniczając się do 10 000 elementów). W przypadku badania metody *insert*, wykorzystano najgorszy przypadek, to znaczy, że w pliku wejściowym wartości kluczy były zapisane rosnąco, tak, aby metoda za każdym razem dodawała na koniec kolejki - wykonywała najwięcej przesunięć. Poniżej przedstawiono wykresy badanych zależności:



Rysunek 2: Zależność czasu działania metody *insert* od liczby elementów w kolejce



Rysunek 3: Zależność czasu działania metody *remove\_minimum* od liczby elementów w kolejce

Pomimo dużych odchyień, można zauważyć, że wykresy potwierdziły hipotezę - metoda *insert* ma złożoność obliczeniową liniową  $O(n)$ , a metoda *remove\_minimum* ma złożoność obliczeniową stałą w czasie  $O(1)$ .

## 5. Podsumowanie i wnioski

- Dobór struktury danych okazał się być dobry. Lista dwukierunkowa bardzo usprawniła działanie programu i efektywność metod.

- Lista uporządkowana sprawiła, że złożoność obliczeniowa metody *insert* oraz *remove\_minimum* wyniosła kolejno  $O(n)$  oraz  $O(1)$ . Zastosowanie listy nieuporządkowanej nie zmieniłoby ogólnej złożoności obliczeniowej programu.
- Można powiedzieć, że złożoność obliczeniowa programu jest funkcją liniową. Jest to dobry wynik, lecz dla dużej ilości danych lepsza byłaby złożoność w funkcji logarytmicznej.

## 6. Literatura

1. Wykład „Projektowanie Algorytmów i Metody Sztucznej Inteligencji”
2. M.T.Goodrich, R. Tamassia, D.Mount „Data Structures & Algorithms in C++”
3. [https://cpp0x.pl/kursy/Kurs-STL-C++/Adapter-kolejki-priorytetowej-std-priority\\_queue/118](https://cpp0x.pl/kursy/Kurs-STL-C++/Adapter-kolejki-priorytetowej-std-priority_queue/118) (dostęp: 31.03.2022)