Politechnika Wrocławska

Projektowanie i Analiza Algorytmów
Projekt 1
Sprawozdanie

Kod kursu: K00-37d
Dominik Molenda
264273

1 Tablica dwuwymiarowa

1.1 Deklaracja Tablicy

```
int **arr = new int *[a];
for (int i = 0; i < a; i++) {
    arr[i] = new int[b];
}
```

Rysunek 1. Deklaracja tablicy dynamicznej o rozmiarach a x b.

Deklaracja tablicy dwuwymiarowej (znajdująca się w funkcji int main) najpierw tworzy tablice jednowymiarową o rozmiarze **a**, (w implementacji programu użytkownik wpisuje rozmiar), a następnie dla każdego elementu tej tablicy, jest przypisywana kolejna tablica.

1.2 Wypełnianie losowymi wartościami

```
void insertRandomValues(int **arr, int x, int a, int b) {
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++) {
            arr[i][j] = rand() % x;
        }
    }
}
```

Rysunek 2. Wypełnienie tablicy o rozmiarze a x b losowymi wartościami w przedziale od 0 do x.

Wypełnianie tablicy dwuwymiarowej losowymi wartościami w przedziale od 0 do x, odbywa się poprzez iterowanie kolejno przez dwa wymiary tablicy, a następnie losowanie pojedynczej wartości za pomocą funkcji rand.

1.3 Wyświetlanie tablicy

```
void displayArray(int **arr, int a, int b) {
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

Rysunek 3. Wyświetlanie tablicy o rozmiarze a x b.

Wyświetlanie tablicy odbywa się analogicznie do przypisania losowej wartości dla danego miejsca w tablicy. Zamiast przypisywać wartość do zmiennej - wyświetlamy ją. Po zakończeniu wewnętrznej pętli for, wyświetlamy endline'a aby wszystko było dobrze widoczne w konsoli.

1.4 Znajdywanie maksymalnej wartości

```
void displayMax(int **arr, int a, int b) {
    int temp = 0;
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++) {
            if (arr[i][j] > temp) {
                temp = arr[i][j];
            }
        }
    }
    cout << "Max value is: " << temp;
}
```

Rysunek 4. Znajdywanie największej wartości w tablicy o rozmiarze a x b.

Przy szukaniu maksymalnej wartości w tablicy, trzeba zadeklarować zmienną tymczasową w funkcji, która podczas każdej iteracji przez tablicę jest sprawdzana, czy nie jest mniejsza od obecnego elementu tablicy - jeżeli jest, to wtedy dany element tablicy jest przypisywany do tej zmiennej.

1.5 Złożoności obliczeniowe i pamięciowe

Złożoność	Obliczeniowa	Pamięciowa
Przypisanie zmiennych	$O(1)$	$O(1)$
Tworzenie tablicy dwuwymiarowej	$O(n)$	$O(n)$
Wstawianie losowych wartości do tablicy	$O(n^2)$	$O(n)$
Wyświetlanie tablicy	$O(n^2)$	$O(1)$
Szukanie maksymalnej wartości	$O(n^2)$	$O(n)$

2 Zapis i odczyt z plików

W programie posługujemy się tablicą jednowymiarową o rozmiarze 5, do której można wykorzystać funkcje zbliżoną do funkcji "insertRandomValues" z poprzedniego zadania.

2.1 Zapisywanie do pliku tekstowego

```
void saveToTXT(int arr[]) {
    file.open("array.txt", ios::out | ios::app);
    if (file.is_open()) {
        for (int i = 0; i < 5; i++) {
            file << arr[i] << endl;
        }
        file.close();
    }
}
```

Rysunek 5. Funkcja odpowiedzialna za zapisanie tablicy do pliku .txt.

Funkcja zapisywania do pliku tekstowego najpierw otwiera plik, następnie sprawdza czy został poprawnie otwarty. Następnie odbywa się iteracja przez każdy element tablicy i za pomocą funkcji z biblioteki *"fstream"*, zapisuje każdy element tablicy do pliku, po czym zamyka plik.

2.2 Odczytanie z pliku tekstowego

```
void loadFromTXT(int arr[]) {
    file.open("array.txt", ios::in);
    if (file.is_open()) {
        for (int i = 0; i < 5; i++) {
            file >> arr[i];
        }
        file.close();
    }
}
```

Rysunek 6. Funkcja odpowiedzialna za odczytanie tablicy z pliku .txt.

Odczytywanie z pliku tekstowego odbywa się odwrotnie (poza włączeniem, sprawdzeniem i wyłączeniem pliku) - do każdego elementu tablicy jest przypisywana kolejna wartość z pliku.

2.3 Zapisywanie do pliku binarnego

```
void saveToBIN(int arr[]) {
    ofstream file("array2.bin", ios::binary);
    if (file.is_open()) {
        for (int i = 0; i < 5; i++) {
            file << arr[i] << endl;
        }
        file.close();
    }
}
```

Rysunek 7. Funkcja odpowiedzialna za zapisanie tablicy do pliku .bin.

Zapisywanie do pliku binarnego odbywa się analogicznie do pliku tekstowego.

2.4 Odczytanie z pliku binarnego

```
void loadFromBIN(int arr[]) {
    ifstream file("array2.bin", ios::binary);
    if (file.is_open()) {
        for (int i = 0; i < 5; i++) {
            file >> arr[i];
        }
        file.close();
    }
}
```

Rysunek 8. Funkcja odpowiedzialna za odczytanie tablicy z pliku .bin.

Odczytywanie z pliku binarnego odbywa się analogicznie do pliku tekstowego.

2.5 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
Wstawianie losowych wartości do tablicy	$O(n)$	$O(n)$
Wyświetlanie tablicy	$O(n)$	$O(1)$
Zapisanie tablicy do pliku	$O(n)$	$O(n)$
Odczytanie tablicy do pliku	$O(n)$	$O(n)$

3 Rekurencja

3.1 Potęga

```
int Potega(int x, int p) {
    if (p == 0)
        return 1;
    return x * Potega(x, p - 1);
}
```

Rysunek 9. Funkcja rekurencyjna licząca x do potęgi p.

Funkcja do obliczania potęgi działa rekurencyjnie, czyli wywołuje się za każdym razem gdy p jest większe od 1, ale każde wywołanie dekrementuje parametr p o 1. Przykładowo gdy chcemy policzyć 3 do potęgi trzeciej (warunki

początkowe: x = 3, p = 3), to funkcja wywoła się 4 razy, najpierw dla 3^3, następnie wywoła się dla p = 2, a całe równanie, które będzie zwracane tymczasowo będzie miało postać $3 * 3^2$, aż dojdzie do momentu $3 * 3 * 3 * 1$.

3.2 Silnia

```
int Silnia(int x) {
    if (x == 1)
        return 1;
    return x * Silnia(x - 1);
}
```

Rysunek 10. Funkcja rekurencyjna licząca silnie.

W przypadku funkcji obliczającej silnie zasada działania jest podobna, a nawet i prostsza. Chcąc policzyć silnie dla 6 pierwsze wywołania rekurencyjne w returnie wyglądają następująco: $6 * Silnia(6 - 1)$, $6 * 5 * Silnia(5 - 1)$, ... , $6 * 5 * 4 * 3 * 2 * 1$.

3.3 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
Potęgowanie	$O(n)$	$O(n)$
Silnia	$O(n)$	$O(n)$

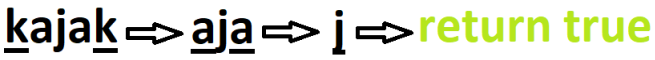
4 Palindrom

4.1 Działanie funkcji

```
bool jestPal(string testStr) {
    if (testStr.length() <= 1) {
        return true;
    } else if (testStr.front() == testStr.back()) {
        return jestPal(testStr.substr(1, testStr.length() - 2));
    } else {
        return false;
    }
}
```

Rysunek 11. Funkcja sprawdzająca czy ciąg znaków jest palindromem.

Funkcja porównuje pierwszy i ostatni element, gdy są takie same, wywołuje się ponownie bez tych dwóch elementów, aż do momentu gdy zostanie 1 albo 0 - wtedy zwróci true. Jeżeli dwa skrajne elementy są rożne - funkcja zwróci false.



Rysunek 12. Funkcja sprawdzająca czy ciąg znaków jest palindromem dla ciągu znaku który jest palindromem.

Rysunek 13. Funkcja sprawdzająca czy ciąg znaków jest palindromem dla ciągu znaku który nie jest palindromem.

4.2 Złożoności obliczeniowe i pamięciowe

Złożoność	Obliczeniowa	Pamięciowa
Działanie funkcji	$O(n)$	$O(n)$

5 Permutacje

5.1 Funkcja tworząca permutacje

```
void permute(string a, int l, int r) {
    if (l == r) {
        // checking if permutation is palindrome - if it is push it to array
        if (jestPal(a)) {
            palList[counter] = a;
            counter++;
        }
    } else {
        for (int i = l; i <= r; i++) {
            // swaping left index with i index
            swap(a[l], a[i]);
            // recursive call to itself with incrementing left index (ex. change from 1st to 2nd letter)
            permute(a, l + 1, r);
        }
    }
}
```

Rysunek 14. Fragment kodu przedstawiający funkcje, która tworzy permutacje oraz sprawdza czy dana permutacja jest palindromem.

Ta funkcja generuje permutacje łańcucha znakowego a od pozycji l (początkowo indeks 0 w łańcuchu znaków) do r (indeks końcowy w łańcuchu znaków) włącznie, sprawdzając każdą permutację czy jest palindromem za pomocą funkcji jestPal() z poprzedniego zadania, a jeżeli jest dodaje ją do tablicy palList. Funkcja korzysta z rekurencji i iteracyjnie zamienia każdy znak na pozycji l z każdym pozostałym na pozycjach od l do r. Po każdej zamianie wywołuje samą siebie dla pozycji l+1 i r, aby uzyskać wszystkie możliwe permutacje. Ta funkcja wykorzystuje także zmienną globalną counter, która zlicza ilość palindromów w tablicy palList.

5.2 Usuwanie duplikatów z tablicy

```
void usunDup() {
    for (int i = 0; i < counter; i++) {
        for (int j = i + 1; j < counter; j++) {
            if (palList[i] == palList[j]) {
                for (int k = j; k < counter - 1; k++) {
                    palList[k] = palList[k + 1];
                }
                counter--;
                j--;
            }
        }
    }
}
```

Rysunek 15. Fragment kodu przedstawiający funkcje, która usuwa duplikaty w tablicy palList zdefiniowanej .

Funkcja ta usuwa duplikaty z tablicy iterując po jej wartościach, wtedy sprawdza czy w dalszym fragmencie tablicy pojawia się aktualna wartość, a jeżeli tak - to cała tablica jest „zsuwana" a zmienna counter dekrementowana.

5.3 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
Znajdowanie permutacji	$O(n!)$	$O(n)$
Usuwanie duplikatów	$O(n^3)$	$O(1)$

6 Lista

6.1 Klasa Node - Węzeł listy

```
class Node {
public:
    int value;
    Node *next;

    Node(int tempValue) {
        this->value = tempValue;
        this->next = NULL;
    }
};
```

Rysunek 16. Fragment kodu przedstawiający implementacje klasy Node.

Klasa Node (węzeł) definiuje węzeł listy, który zawiera dwie zmienne składowe: "value" - wartość węzła oraz "next" - wskaźnik na kolejny element.

6.2 Klasa List - Lista

```
class List {
private:
    Node *head;

public:
    List() {…

    void displayList() {…

    void addLast(int tempValue) {…

    void addFirst(int tempValue) {…

    void deleteByValue(int tempValue) {…

    void deleteAll() {…
};
```

Rysunek 17. Fragment kodu przedstawiający implementacje klasy List.

Klasa List definiuje listę jednokierunkową, która zawiera prywatną zmienną head, czyli wskaźnik na pierwszy element. Wszystkie metody w tej klasie są odpowiednimi funkcjonalnościami, które pozwalają na: wyświetlenie listy, dodanie elementu na końcu, dodanie elementu na początku, usuwanie element o podanej wartości oraz usuwanie całej listy.

6.3 Metoda Wyświetlania - displayList()

```
void displayList() {
    Node *current = head;
    while (current != NULL) {
        cout << current->value << " ";
        current = current->next;
    }
    cout << endl;
}
```

Rysunek 18. Fragment kodu przedstawiający implementacje metody wyświetlającej listę.

Metoda ”displayList” iteruje przez elementy listy, zaczynając od początku listy (czyli od węzła wskazywanego przez ’head’). W każdej iteracji wyświetla wartość bieżącego węzła i przesuwa się na kolejny element listy, aż do momentu, gdy natrafi na wskaźnik ’NULL’, który oznacza koniec listy.

6.4 Metoda dodawania elementu na końcu

```
void addLast(int tempValue) {
    Node *newNode = new Node(tempValue);
    // checking if head is empty
    if (head == NULL) {
        head = newNode;
    // if not - make pointer in the "last one" to new one
    } else {
        Node *current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        current = current->next;
        current->next = NULL;
    }
}
```

Rysunek 19. Fragment kodu przedstawiający implementacje metody, która dodaje element na końcu listy.

Metoda "addLast" dodaje nowy element do końca listy. Tworzy nowy obiekt klasy Node, przypisuje do jego pola 'value' wartość przekazaną jako argument do funkcji, a następnie szuka ostatniego elementu listy, czyli elementu, którego pole 'next' wskazuje na NULL. Po znalezieniu ostatniego elementu ustawia jego pole 'next' na nowy węzeł.

6.5 Metoda dodawania na elementu na początku

```
void addFirst(int tempValue) {
    Node *newNode = new Node(tempValue);
    // checking if head is empty
    if (head == NULL) {
        head = newNode;
    //if not - repin head to new node, and use "old head" as new node's next
    } else {
        newNode->next = head;
        head = newNode;
    }
}
```

Rysunek 20. Fragment kodu przedstawiający implementacje metody, która dodaje element na początku listy.

Metoda 'addFirst' działa analogicznie do 'addLast', jednak nowy element jest dodawany na początku listy. W tym przypadku należy zmienić wartość pola 'head' tak, aby wskazywała na nowy węzeł, a stary wskaźnik head przepisać jako next w nowym head node.

6.6 Metoda usuwania po wartości

```
void deleteByValue(int tempValue) {
    // check if link is empty
    if (head == NULL) {
        cout << "List is empty!";
    } else {
        // if searched value is head
        if (head->value == tempValue) {
            Node *tempNode = head;
            head = head->next;
            delete tempNode;
            // if not - search first element with
        } else {
            Node *current = head;
            while (current->next != NULL) {
                if (current->next->value == tempValue) {
                    Node *tempNode = current->next;
                    current->next = current->next->next;
                    delete tempNode;
                }
                current = current->next;
            }
        }
    }
}
```

Rysunek 21. Fragment kodu przedstawiający implementacje metody, usuwający element listy.

Metoda 'deleteByValue' usuwa element listy, którego wartość pola 'value' jest równa przekazanej do funkcji wartości. W przypadku, gdy lista jest pusta, metoda wyświetla komunikat "List is empty!". W przeciwnym przypadku, metoda iteruje przez elementy listy, szukając elementu o podanej wartości. Jeśli element ten zostanie znaleziony, usunięty zostaje poprzez zmianę wskaźnika na następny element.

6.7 Metoda usunięcia całej listy

```
void deleteAll() {
    Node *current = head;
    while (current != NULL) {
        Node *tempNode = current;
        current = current->next;
        delete tempNode;
    }
    head = NULL;
}
```

Rysunek 22. Fragment kodu przedstawiający implementacje metody, usuwający wszystkie elementy listy.

Metoda 'deleteAll' usuwa wszystkie elementy z listy. Iteruje przez elementy listy, usuwając je jeden po drugim, aż do momentu, gdy wskaźnik 'head' wskazuje na NULL, co oznacza, że lista jest pusta.

6.8 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
Wyświetlanie listy	$O(n)$	$O(1)$
Dodawanie na koniec	$O(1)$	$O(1)$
Dodawanie na początek	$O(1)$	$O(1)$
Usuwanie elementu po wartości	$O(n)$	$O(1)$
Usuwanie wszystkich elementów	$O(n)$	$O(n)$

7 Kolejka

7.1 Klasa Node - Węzeł kolejki

```
class Node {
public:
    int value;
    Node *next;

    Node(int tempValue) {
        this->value = tempValue;
        this->next = NULL;
    }
};
```

Rysunek 23. Fragment kodu przedstawiający implementacje klasy Node.

Klasa Node (węzeł) definiuje węzeł kolejki, który zawiera dwie zmienne składowe: "value" - wartość węzła oraz "next" - wskaźnik na kolejny element, dodatkowo posiada tail, czyli wskaźnik na ostatni element w kolejce.

7.2 Klasa Queue - Kolejka

```
class Queue {
private:
    Node *head;
    Node *tail;

public:
    Queue() {…

    void displayQueue() {…

    void enqueue(int tempValue) {…

    void dequeue() {…

    void deleteAll() {…
};
```

Rysunek 24. Fragment kodu przedstawiający implementacje klasy List.

Klasa Queue reprezentuje kolejkę (FIFO - First In First Out), w której elementy dodawane są na końcu kolejki, a usuwane z początku kolejki - praktycznie tak jak kolejka w sklepie. Posiada head, analogicznie jak lista.

7.3 Metoda wyświetlenia kolejki

```
void displayQueue() {
    Node *current = head;
    while (current != NULL) {
        cout << current->value << " ";
        current = current->next;
    }
}
```

Rysunek 25. Fragment kodu przedstawiający implementacje klasy List.

Metoda ta wyświetla zawartość kolejki iterując przez wszystkie elementy poprzez wskaźnik na kolejny element, do momentu aż wskaźnik będzie wskazywał na NULL.

7.4 Metoda dodawania do kolejki - enqueue

```
void enqueue(int tempValue) {
    Node *newNode = new Node(tempValue);

    if (head == NULL && tail == NULL) {
        head = newNode;
        tail = newNode;
    } else {
        Node *current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        current = current->next;
        current->next = NULL;
        tail = current;
    }
}
```

Rysunek 26. Fragment kodu przedstawiający implementacje klasy List.

Dodaje nowy element do kolejki. Metoda tworzy nowy węzeł typu Node z podaną wartością, a następnie dodaje go na końcu kolejki (tail wskazuje na nowy element) poprzez iterowanie do momentu aż wskaźnik nie wskaże na NULL. Jeśli kolejka jest pusta, to zarówno wskaźnik na początek kolejki (head) jak i na koniec kolejki (tail) wskazują na ten sam węzeł.

7.5 Metoda usuwania z kolejki - dequeue

```
void dequeue() {
    if (head == NULL) {
        cout << "Queue is empty!";
    } else if (head == tail) {
        Node *current = head;
        delete current;
        head = NULL;
        tail = NULL;
    } else {
        Node *current = head;
        head = current->next;
        delete current;
    }
}
```

Rysunek 27. Fragment kodu przedstawiający implementacje klasy List.

Usuwa pierwszy element z kolejki. Metoda usuwa pierwszy węzeł z kolejki, czyli ten na który wskazuje wskaźnik head, w kodzie przed jego usunięciem wskaźnik na head będzie przepięty na drugi element kolejki. W przypadku, gdy kolejka jest już pusta, metoda wyświetla komunikat o tym na ekranie.

7.6 Metoda usuwania całej kolejki

```
void deleteAll() {
    while (head != NULL) {
        dequeue();
    }
}
```

Rysunek 28. Fragment kodu przedstawiający implementacje klasy List.

Wywołuje usuwanie pierwszego elementu (dequeue), do momentu aż head nie będzie wskazywał na NULL - wtedy ma pewność, że kolejka jest pusta.

7.7 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
Wyświetlanie kolejki	$O(n)$	$O(1)$
Enqueue	$O(1)$	$O(1)$
Dequeue	$O(1)$	$O(1)$
Usuwanie całej kolejki	$O(n)$	$O(1)$

8 Deque

8.1 Wstęp teoretyczny

Deque (**Double-Ended Queue**), jest strukturą danych która umożliwia dodawanie i usuwanie elementów z obu jej końców. Normalnie deque jest implementowana na liście dwukierunkowej lub na tablicy dynamicznej - natomiast w tej implementacji użyłem tablicy statycznej. Podczas implementacji na tablicy statycznej - dane nie są tyle co usuwane, a przestają być wyświetlane do momentu ich nadpisania.

```
class Deque {
public:
    int dequeArr[SIZE] = {…
    int front = -1;
    int rear = -1;
    int currentSize = 0;

    Deque() {}

    void getSize() {…

    void isEmpty() {…

    void getFront() {…

    void getRear() {…

    void insertFront(int tempValue) {…

    void deleteFront() {…

    void insertRear(int tempValue) {…

    void deleteRear() {…

    void displayDeque() {…
};
```

Rysunek 29. Fragment kodu przedstawiający implementacje klasy Deque.

8.2 Metoda getSize

```
void getSize() {
    cout << "Size of deque" << currentSize;
}
```

Rysunek 30. Fragment kodu przedstawiający implementacje metody wyświetlającą rozmiar Deque.

Zmienna currentSize typu int, pozwala na znajomość "liczby obecnie używanych" obszarów w pamięci, a metoda getSize() ją wyświetla.

8.3 Metoda isEmpty

```
void isEmpty() {
    if (currentSize == 0) {
        cout << "Deque is empty";
    } else {
        cout << "Deque is not empty";
    }
}
```

Rysunek 31. Fragment kodu przedstawiający implementacje metody wyświetlającą informacje czy Deque jest puste.

Korzystając ze zmiennej currentSize metoda isEmpty sprawdza czy zmienna jest równa 0, jeżeli tak - to Deque jest puste, jeżeli nie - to nie jest.

8.4 Metoda getFront i getRear

```
void getFront() {
    if ((front == -1) && (rear == -1)) {
        throw "Deque is empty";
    } else {
        cout << dequeArr[front];
    }
}
```

```
void getRear() {
    if ((front == -1) && (rear == -1)) {
        throw "Deque is empty";
    } else {
        cout << dequeArr[rear];
    }
}
```

Rysunek 32. Fragment kodu przedstawiający implementacje metody wyświetlającą wartość z początku Deque.

Jeżeli zmienne odpowiadająca za skrajnie przedni i tylni element są równe -1, oznacza to, że Deque jest empty. W innym wypadku metoda wyświetla element tablicy na miejscu front lub rear.

8.5 Metoda insertFront

```
void insertFront(int tempValue) {
    if (((front == 0) && (rear == SIZE - 1)) || (front == rear + 1)) {
        throw "Deque is full";
    } else if (front == -1 && rear == -1) {
        front = rear = 0;
        dequeArr[front] = tempValue;
        currentSize++;
    } else if (front == 0) {
        front = SIZE - 1;
        dequeArr[front] = tempValue;
        currentSize++;
    } else {
        front--;
        dequeArr[front] = tempValue;
        currentSize++;
    }
}
```

Rysunek 32. Fragment kodu przedstawiający implementacje metody dodający element na początku Deque.

Pierwszy warunek sprawdza czy tablica nie jest zajęta w dwa sposoby:

- Jeżeli front jest równe 0, a rear równe rozmiarowi pomniejszonemu o jeden - tablica jest pełna od miejsca z indeksem 0 do indeksu rozmiar pomniejszony o jeden.
- Jeżeli front jest równy rear powiększonemu o jeden - w takim momencie dodanie elementu z przodu kolejki, spowodowałoby nadpisanie elementu znajdującego się z tyłu kolejki.

Następnie kolejny warunek sprawdza czy Deque jest puste, jeżeli tak, przyrównuje front i rear do 0, a następnie na tym indeksie dodaje pierwszy element, po czym inkrementuje zmienną o aktualnym rozmiarze.
Kolejny warunek sprawdza, czy front nie znajduje się na zerowym indeksie tablicy - jeżeli tak, front jest przerzucany na koniec tablicy, a następnie tam dodawana jest wartość przekazana do funkcji. Analogicznie jak w poprzednim else if'ie, inkrementujemy zmienną currentSize.
Ostatni warunek sprawdza się jeżeli np. front jest na 5 pozycji w tablicy, wtedy dekrementujemy go o 1, po czym w tablicy o danym indeksie zapisujemy zmienną. Następnie inkrementujemy zmienną currentSize.

8.6 Metoda deleteFront

```
void deleteFront() {
    if ((front == -1) && (rear == -1)) {
        throw "Deque is empty";
    } else if (front == rear) {
        front = -1;
        rear = -1;
        currentSize--;
    } else if (front == (SIZE - 1)) {
        front = 0;
        currentSize--;
    } else {
        front++;
        currentSize--;
    }
}
```

Rysunek 33. Fragment kodu przedstawiający implementacje metody usuwający element z początku Deque.

Pierwszy warunek sprawdza czy Deque jest puste - jeżeli tak, wyrzuca błąd. Kolejny warunek sprawdza czy Deque zawiera tylko jeden element - jeżeli tak, to można go bezpiecznie usunąć i ustawić zmienne front i rear na -1, aby zasygnalizować, że Deque jest puste, po czym dekrementowana jest zmienna currentSize.
Następnie sprawdzane jest czy front nie wskazuje na ostatni indeks tablicy, gdy tak sie dzieje, przesuwamy front na 0, tak aby umożliwić nadpisane wartości na ostatnim indeksie.
Jeżeli żaden z tych warunków nie został spełniony, nie ma żadnych problemów aby przesunąć front "w prawo" i zmiejszyć zmienną sygnalizującą o liczbe danych w tablicy.

8.7 Metoda insertRear

```
void insertRear(int tempValue) {
    if (((front == 0 && rear == SIZE - 1) || front == rear + 1)) {
        throw "Deque is full";
    } else if (front == -1 && rear == -1) {
        rear = 0;
        dequeArr[rear] = tempValue;
        currentSize++;
    } else if (rear == SIZE - 1) {
        rear = 0;
        dequeArr[rear] = tempValue;
        currentSize++;
    } else {
        rear++;
        dequeArr[rear] = tempValue;
        currentSize++;
    }
}
```

Rysunek 34. Fragment kodu przedstawiający implementacje metody dodający element na koniec Deque.

Pierwszy if działa analogicznie do metody insertFront.
Druga instrukcja warunkowa wykonuje się gdy front i rear sygnalizują, że Deque jest puste - wtedy rear ustawiamy na pierwszy indeks tablicy, a następnie zapisujemy tam zmienną tempValue. Tak jak w poprzednich przypadkach - zmienna currentSize jest inkrementowana.
W kolejnym else if'ie rozpatrywany jest przypadek gdy rear znajduje sie na końcu tablic - wtedy reat trzeba ustawić na jej początek, dodać wartość i zinkrementować currentSize.
W pozostałych przypadkach nie ma nic skomplikowanego, np. jeżeli rear jest równe 4, inrkrementujemy rear do 5, wstawiamy na ten indeks tempValue, a na koniec inkrementujemy zmienną currentSize.

8.8 Metoda deleteRear

```
void deleteRear() {
    if ((front == -1) && (rear == -1)) {
        throw "Deque is empty";
    } else if (front == rear) {
        front = -1;
        rear = -1;
        currentSize--;
    } else if (rear == 0) {
        rear = SIZE - 1;
        currentSize--;
    } else {
        rear--;
        currentSize--;
    }
}
```

Rysunek 35. Fragment kodu przedstawiający implementacje metody usuwający element z końca Deque.

Pierwsze dwa if'y działają analogicznie jak w metodzie deleteFront.
Jeżeli rear jest równe 0 (znajduje się na pierwszym indeksie tablicy), przenosimy go na ostatni i zmienna informująca o zapełnieniu tablicy jest dekrementowana. Ostatni przypadek analogicznie do wszystkich powyższych.

8.9 Metoda displayDeque

```
void displayDeque() {
    if (currentSize > 0) {
        int i = front;
        while (i != rear) {
            cout << dequeArr[i];
            i = (i++) % SIZE;
            cout << " ";
        }
        cout << dequeArr[rear];
    } else {
        throw "Deque is empty";
    }
}
```

Rysunek 37. Fragment kodu przedstawiający implementacje metody wyświetlającą Deque.

Wyświetlanie Deque odbywa się poprzez sprawdzenie, czy Deque znajdują się elementy dla nas. Jeżeli tak, to iterujemy od początku (zmienna front) wyświetlając kolejno kolejne elementy Deque, aż do momentu gdy i będzie równe końcowi (rear). Sama interacja jest całkiem nietypowa, ponieważ może zdarzyć się takie ułożenie tablicy że początek będzie na indeksie 8, a koniec na indeksie 3 - dlatego w pętli while, znajduje się modulo z SIZE, aby po przeskoczeniu na koniec tablicy, znaleźć się na jej początku. Na koniec wyświetlany jest element z końca tablicy (rear).

8.10 Implementacja dodatkowa - Metoda jestPal

```
void jestPal() {
    if (currentSize <= 1) {
        cout << "Palindome!";
    } else if (this->getFront() == this->getRear()) {
        this->deleteFront();
        this->deleteRear();
        jestPal();
    } else {
        cout << "It is not a palindrome.";
    }
}
```

Rysunek 38. Fragment kodu przedstawiający implementacje metode sprawdzającą czy zawartość Deque jest palindromem.

Po zmianie kilku metod które były voidami na inty (bądź chary, zależy od implementacji programu) udało mi się stworzyć program, który wykonuje to samo co program z podpunktu 4, tylko tym razem wykorzystuje metody jakie zaimplementowałem w klasie Deque. Analogicznie jak program z zadania 4 - funkcja wywołuje sie rekurencyjnie usuwając zawartość deque, do momentu aż albo zawartość nie okaże się palindromem, albo gdy będzie pusta, albo gdy zostanie 1 symbol (co oznacza że jest palindromem).

8.11 Złożoności obliczeniowe i pamięciowe

Złożoność	**Obliczeniowa**	**Pamięciowa**
getSize()	$O(1)$	$O(1)$
isEmpty()	$O(1)$	$O(1)$
getFront()	$O(1)$	$O(1)$
getRear()	$O(1)$	$O(1)$
insertFront()	$O(1)$	$O(1)$
deleteFront()	$O(1)$	$O(1)$
insertRear()	$O(1)$	$O(1)$
deleteRear()	$O(1)$	$O(1)$
displayDeque()	$O(n)$	$O(1)$
jestPal()	$O(n)$	$O(1)$