Politechnika Wrocławska

Struktury danych

Projekt 1

Freddy-Ms
ArtsyKimiko

8 maja 2024

1 Wstęp teoretyczny

Tablica dynamiczna jest to struktura danych, która jest podobna do standardowej tablicy, ale ma zdolność dynamicznego zmieniania swojego rozmiaru w czasie działania programu. Oznacza to, że może ona być zwiększana lub zmniejszana w zależności od potrzeb. W przeciwieństwie do statycznych tablic, które mają z góry ustaloną wielkość, tablice dynamiczne mogą być dostosowywane do zmieniającej się ilości danych.

Lista jednokierunkowa jest to struktura danych, która składa się z węzłów, z których każdy zawiera pewną wartość (np. liczba, ciąg znaków) oraz wskaźnik do następnego węzła w sekwencji. Węzły są połączone w taki sposób, że tworzą liniową sekwencję, w której dostęp do każdego elementu jest możliwy poprzez przejście od początku do końca listy. Każdy węzeł posiada wskaźnik (nazywany również referencją) do następnego węzła. Ostatni węzeł w liście zazwyczaj ma wskaźnik do nulla, oznaczającego koniec listy.

Lista dwukierunkowa to struktura danych, która jest podobna do listy jednokierunkowej, z tą różnicą, że każdy węzeł zawiera dodatkowo wskaźnik do poprzedniego węzła. Dzięki temu można przeglądać listę w obie strony - od początku do końca lub od końca do początku - co umożliwia szybkie przemieszczanie się po elementach listy w obu kierunkach.

Złożoność obliczeniowa poszczególnych funkcji							
	AddFirst	AddAt	AddLast	RemoveFirst	RemoveAt	RemoveLast	Contains
ArrayList	O(n)	O(n)	O(n)/O(1)	O(n)	O(n)	O(1)	O(n)
HeadList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
SingleList	O(1)	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)
DoubleList	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)

Rysunek 1: Złożoność obliczeniowa wykorzystanych funkcji

Przyjęte oznaczenia:

- ArrayList - tablica dynamiczna
- HeadList - lista jednokierunkowa ze wskazaniem na głowę
- SingleList - lista jednokierunkowa ze wskazaniem na głowę i ogon
- DoubleList - lista dwukierunkowa

2 Działanie programu

2.1 Tablica dynamiczna

2.1.1 Automatyczne rozszerzanie tablicy

W momencie, gdy tablica osiąga swoją maksymalną pojemność, funkcja dwukrotnie zwiększa jej pojemność, przenosząc istniejące elementy do nowej, większej tablicy.

```
void resize(){
    T* temp = new T[capacity*2]; // allocating an array with twice the capacity
    for(int i = 0; i < length; i++)
        temp[i] = arr[i]; // copying elements
    delete[] arr; // freeing memory of the old array
    arr = temp; // assigning the new array
    capacity *= 2; // updating the capacity
}
```

2.1.2 Dodawanie elementów

2.1.2.1 Dodawanie na początek tablicy

Aby dodać nowy element na początku dynamicznej tablicy, program najpierw sprawdza, czy tablica osiągnęła swoją maksymalną pojemność. Jeśli tak, wykonuje operację `resize`, która zwiększa pojemność tablicy, a następnie przesuwa wszystkie istniejące elementy o jedną pozycję w prawo. Dzięki temu tworzy się miejsce dla nowego elementu, który zostanie umieszczony na początku tablicy.

```
virtual void addFirst(T element) override{
    if(isFull()) // if we exceed the size of the array, we expand it
        resize();
    for(int i = length; i > 0; i--)
        arr[i] = arr[i-1]; // shifting each element to the right
    arr[0] = element; // assigning the element as the first element of the array
    length++; // increasing the number of elements in the array
}
```

2.1.2.2 Dodawanie na koniec

Podczas dodawania nowego elementu do tablicy jako ostatniego, program najpierw sprawdza jej aktualną pojemność. Jeśli tablica jest już zapełniona, stosuje operację `resize`, aby zwiększyć jej rozmiar, a następnie umieszcza nowy element na końcu tablicy, zwiększając jej rozmiar o jeden.

```
virtual void addLast(T element) override{
    if(isFull()) // if we exceed the size of the array, we expand it
        resize();
    arr[length] = element; // assigning the element as the last element of the array
    length++; // increasing the number of elements in the array
}
```

2.1.2.3 Dodawanie w wybranym miejscu

Program najpierw sprawdza, czy indeks mieści się w granicach istniejących elementów. Jeśli nie, zgłasza błąd. Następnie przesuwa elementy poza wstawianym indeksem w prawo, aby zrobić miejsce dla nowego elementu, który zostaje umieszczony na wskazanym indeksie, zwiększając długość listy o jeden.

```
virtual void addAt(int index, T element) override{
    if(index < 0 || index > length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(isFull()) // if we exceed the size of the array, we expand it
        resize();
    for(int i = length ; i >= index; i--)
        arr[i] = arr[i-1]; // shifting each element to the right
    arr[index] = element; // insert the new element at the specified index
    length++; // increasing the length of the list
}
```

2.1.3 Usuwanie elementów

2.1.3.1 Usuwanie z wybranego miejsca

Aby usunąć element z określonego miejsca w tablicy, funkcja najpierw sprawdza poprawność indeksu, a następnie przesuwa wszystkie elementy znajdujące się za usuwanym indeksem w lewo, aby zapełnić lukę pozostawioną po usuniętym elemencie.

```
virtual void removeAt(int index) override{
    if(index < 0 || index >= length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    for(int i = index; i < length; i++)
        arr[i] = arr[i+1]; // shifting each element to the left
    length--; // decreasing the length of the list
}
```

2.1.3.2 Usuwanie na początku

Funkcja usuwająca pierwszy element wykorzystuje wcześniej opisaną metodę usuwającą, przywołując ją z indeksem 0, co odpowiada pierwszemu elementowi w zbiorze.

```
virtual void removeFirst() override{
    removeAt(0);
}
```

2.1.3.3 Usuwanie na końcu

Funkcja usuwająca ostatni element korzysta z wcześniej opisanej metody usuwania, wywołując ją z indeksem równym wielkości zbioru pomniejszonym o 1, co odpowiada ostatniemu elementowi w zbiorze.

```
virtual void removeLast() override{
    removeAt(length-1);
}
```

2.2 Lista jednokierunkowa ze wskazaniem na głowę

2.2.1 Dodawanie elementów

2.2.1.1 Dodawanie na początek

Aby dodać nowy element na początku listy jednokierunkowej, program najpierw tworzy nowy węzeł zawierający podany element następnie ustawia go jako nową głowę listy oraz zwiększa liczbę elementów w liście.

```
virtual void addFirst(T element) override{
    Node* newNode = new Node(element, head); // new node with element, pointing to head
    head = newNode; // update head to point to new node, making it the first node
    length++; // increasing the number of elements in the array
}
```

2.2.1.2 Dodawanie na koniec

Podczas dodawania elementu na koniec listy funkcja w pierwszej kolejności sprawdza czy lista jest pusta. Jeśli tak, ustawia nowy węzeł jako głowę. W przeciwnym przypadku przeszukuje listę, aż znajdzie ostatni węzeł, i ustawia nowy węzeł jako jego następce oraz zwiększa liczbę elementów w liście.

```
virtual void addLast(T element) override{
    Node* newNode = new Node(element); // create a new node with given element
    if(!head) // checking if the list is empty
        head = newNode; // set the new node as the head of the list
    else{
        Node* temp = head; // temporary pointer to the head of the list
        while(temp->next)
            temp = temp->next; // shifting the pointer until that is the last node
        temp->next = newNode; // set the next pointer of the last node to the new node
    }
    length++; // increasing the number of elements in the array
}
```

2.2.1.3 Dodawanie w wybranym miejscu

Przed dodaniem elementu na wybrane miejsce, funkcja najpierw sprawdza poprawność indeksu. Jeśli jest on równy 0 lub długości listy, odpowiednio wywoływane są metody `addFirst()` lub `addLast()`. W przeciwnym razie funkcja przechodzi po liście do elementu o 1 mniejszego niż wskazany aby wstawić nowy węzeł między nim a kolejnym elementem, zaktualizować wskaźniki i zmniejszyć długość listy.

```
virtual void addAt(int index, T element) override{
    if(index < 0 || index > length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(index == 0)
        addFirst(element);
    else if(index == length)
        addLast(element);
    else{
        Node* newNode = new Node(element); // create a new node with given element
        Node* temp = head; // temporary pointer to the head of the list
        for(int i = 0; i < index - 1; i++)
            temp = temp->next; // shifting the pointer to the element before the index
        newNode->next = temp->next; // new node points to element before index position
        temp->next = newNode; // the previous element points to the newly added node
        length++; // increasing the number of elements in the array
    }
}
```

2.2.2 Usuwanie elementów

2.2.2.1 Usuwanie na początku

W pierwszej kolejności sprawdzane jest czy lista nie jest pusta, następnie funkcja ustawia głowę na drugi element listy i usuwa pierwszy. Na koniec zmniejszana jest liczba elementów w liście.

```
virtual void removeFirst() override{
    if(!head){ // checking if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head; // temporary pointer to the head of the list
    head = head->next; // assigning the head to the next element
    delete temp; // deleting the previous head of the list
    length--; // decreasing the length of the list
}
```

2.2.2.2 Usuwanie na końcu

Najpierw sprawdzane jest, czy lista nie jest pusta. Jeśli lista zawiera tylko jeden element, jest on usuwany, a głowa listy ustawiana jest na `nullptr`. W przypadku większej liczby elementów, przeszukiwana jest lista w celu znalezienia przedostatniego elementu. Ostatni element zostaje usunięty, wskaźnik na niego ustawiany jest na `nullptr`, a liczba elementów zmniejszana jest o 1.

```
virtual void removeLast() override{
    if(!head){ // checking if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    if(length == 1){ // check if the list contains only one element
        delete head;
        head = nullptr;
        length = 0;
        return;
    }
    Node* temp = head; // temporary pointer to the head of the list
    while(temp->next->next)
        temp = temp->next; // shifting the pointer until that is the second last node
    delete temp->next; // delete the last node
    temp->next = nullptr; // set the last element pointer to nullptr
    length--; // decreasing the length of the list
}
```

2.2.2.3 Usuwanie z wybranego miejsca

Przed usunięciem elementu z wybranego miejsca, funkcja sprawdza poprawność indeksu. Jeśli indeks jest równy 0, wywoływana jest funkcja `removeFirst()`, a jeśli jest równy długości listy minus jeden, wywoływana jest funkcja `removeLast()`. W przypadku nie spełnienia tych warunków, funkcja przechodzi przez listę, aby znaleźć element poprzedzający ten, który ma zostać usunięty. Następnie aktualizuje wskaźnik `next`, usuwa element i zmniejsza długość listy o 1.

```
virtual void removeAt(int index) override{
    if(index < 0 || index >= length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(index == 0)
        removeFirst();
    else if(index == length - 1)
        removeLast();
    else{
        Node* temp = head; // temporary pointer to the head of the list
        for(int i = 0; i < index - 1; i++)
            temp = temp->next; // shifting the pointer to the element before the index
        Node* toDelete = temp->next; // storing the node to delete
        temp->next = toDelete->next; // pointer to element by element to be destroyed
        delete toDelete; // deleting the node
        length--; // decreasing the length of the list
    }
}
```

2.3 Lista jednokierunkowa ze wskazaniem na głowę i ogon

2.3.1 Dodawanie elementów

2.3.1.1 Dodawanie na początek

Aby dodać element na początku listy, funkcja tworzy nowy węzeł zawierający ten element i ustawia jego wskaźnik na następny węzeł - aktualny początek listy. Jeśli lista jest pusta, nowy węzeł staje się zarówno głową, jak i ogonem. Następnie aktualizowany jest wskaźnik głowy na nowy węzeł i zwiększana długość listy o jeden.

```
virtual void addFirst(T element) override{
    Node* newNode = new Node(element, head); // new node with element, pointing to head
    if(!head){ // if the list is empty, set head and tail to the new node
        tail = newNode;
        head = newNode;
    }
    head = newNode; // update the head pointer to the new node
    length++; // increasing the length of the list
}
```

2.3.1.2 Dodawanie na koniec

Funkcja dodająca element na końcu listy tworzy nowy węzeł zawierający dany element. Jeśli lista jest pusta, nowy węzeł staje się zarówno jej początkiem, jak i końcem. Następnie ustawia wskaźnik na następny węzeł ostatniego węzła na nowy węzeł, aktualizuje wskaźnik końca listy na nowy węzeł oraz zwiększa długość listy o jeden.

```
virtual void addLast(T element) override{
    Node* newNode = new Node(element); // new node with the given element
    if(!head) { // if the list is empty, set head and tail to the new node
        head = newNode;
        tail = newNode;
    }
    else{
        tail->next = newNode; // set next pointer of current tail node to the new node
        tail = newNode; // update the tail pointer to the new node
    }
    length++; // increasing the length of the list
}
```

2.3.1.3 Dodawanie w wybranym miejscu

Aby dodać nowy element w wybranym miejscu tablicy, funkcja najpierw dokonuje sprawdzenia poprawności indeksu. Następnie analizuje, czy indeks wskazuje na początek lub koniec listy, i stosownie wywołuje funkcje `addFirst` lub `addLast`. W przypadku, gdy indeks znajduje się wewnątrz listy, tworzony jest nowy węzeł z podanym elementem. Funkcja przeszukuje listę w poszukiwaniu węzła znajdującego się na pozycji poprzedzającej wskazany indeks, aby umieścić nowy węzeł. Następnie aktualizuje wskaźniki, aby wskazywały na nowo utworzony węzeł oraz na poprzedni węzeł. Na końcu zwiększa długość listy o 1.

```
virtual void addAt(int index, T element) override{
    if(index < 0 || index > length){// checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(index == 0)
        addFirst(element);
    else if(index == length)
        addLast(element);
    else{
        Node* newNode = new Node(element);// new node with the given element
        Node* temp = head;// temporary pointer to the head of the list
        for(int i = 0; i < index - 1; i++)
            temp = temp->next; // shifting the pointer to the element
        newNode->next = temp->next; // new node points to element before index
        temp->next = newNode; // previous element points to the newly added node
        length++; // increasing the length of the list
    }
}
```

2.3.2 Usuwanie elementów

2.3.2.1 Usuwanie na początku

Po sprawdzeniu czy lista nie jest pusta, funkcja przypisuje głowę listy do następnego elementu, usuwa poprzednią głowę oraz zmniejsza długość listy o 1.

```
virtual void removeFirst() override{
    if(!head){ // checking if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head; // temporary pointer to the head of the list
    head = head->next; // assigning the head to the next element
    delete temp; // deleting the previous head of the list
    length--; // decreasing the length of the list
}
```

2.3.2.2 Usuwanie na końcu

Przed usunięciem elementu, funkcja sprawdza czy lista nie jest pusta. Jeśli nie, to sprawdza, czy lista zawiera tylko jeden element. Jeśli tak, usuwa ten element oraz wskaźniki na głowę i ogon listy ustawia na `nullptr`. Gdy warunki te nie są spełnione, funkcja znajduje przedostatni element, usuwa ostatni element, ustawia wskaźnik ogona na ten przedostatni element, zeruje wskaźnik na poprzedni ogon oraz zmniejsza długość listy o 1.

```
virtual void removeLast() override{
    if(!head){ // check if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    if(length == 1){ // check if the list contains only one element
        delete head;
        head = nullptr;
        tail = nullptr;
    }
    else{
        Node* temp = head; // temporary pointer to the head of the list
        while(temp->next != tail)
            temp = temp->next; //shifting the pointer until that is not a tail
        delete tail; // delete the last node - tail
        tail = temp; // update tail pointer to the second last node
        tail->next = nullptr; // set next pointer of the new tail to nullptr
    }
    length--; // decreasing the length of the list
}
```

2.3.2.3 Usuwanie z wybranego miejsca

Przed usunięciem elementu z wybranego miejsca, funkcja sprawdza poprawność indeksu. Jeśli indeks jest równy 0, wywoływana jest funkcja `removeFirst()`, a jeśli jest równy długości listy minus jeden, wywoływana jest funkcja `removeLast()`. W przypadku nie spełnienia tych warunków, funkcja przechodzi przez listę, aby znaleźć element poprzedzający ten, który ma zostać usunięty. Następnie aktualizuje wskaźnik `next`, usuwa element i zmniejsza długość listy o 1.

```
virtual void removeAt(int index) override{
    if(index < 0 || index >= length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(index == 0)
        removeFirst();
    else if(index == length - 1)
        removeLast();
    else{
        Node* temp = head; // temporary pointer to the head of the list
        for(int i = 0; i < index - 1; i++)
            temp = temp->next; // shifting the pointer to the element before the index
        Node* toDelete = temp->next; // storing the node to delete
        temp->next = toDelete->next; // pointer to element by element to be destroyed
        delete toDelete; // deleting the node
        length--; // decrementing the length of the list
    }
}
```

2.4 Lista dwukierunkowa

2.4.1 Dodawanie elementów

2.4.1.1 Dodawanie na początek

W celu dodania elementu na początek listy funkcja tworzy nowy węzeł z danym elementem. Jeśli lista jest pusta (czyli wskaźnik na głowę head jest pusty), to głowa listy wskazuje na nowy węzeł. Jeśli ogon listy (czyli wskaźnik na ogon tail) nie jest pusty, to jego wskaźnik na następny węzeł ustawiany jest na nowy węzeł. Następnie ogon listy wskazuje na nowy węzeł, a długość listy zwiększa się o 1.

```
virtual void addFirst(T element) override{
    Node* newNode = new Node(element, head, nullptr);
    if(!tail) // if the list is empty, set the new node as the head
        tail = newNode;
    if(head) // if tail exists, update its next pointer to point to the new node
        head->prev = newNode;
    head = newNode; // set the tail to the new node
    length++; // increasing the number of elements in the array
}
```

2.4.1.2 Dodawanie na koniec

Aby dodać element na koniec listy funkcja tworzy nowy węzeł z danym elementem. Jeśli ogon listy jest pusty, to ogon ustawiany na nowy węzeł. Jeśli głowa listy nie jest pusta, to jej wskaźnik na poprzedni węzeł ustawiany jest na nowy węzeł. Następnie głowa listy wskazuje na nowy węzeł, a długość listy zwiększa się o 1.

```
virtual void addLast(T element) override{
    Node* newNode = new Node(element,nullptr,tail); // new node with provided element,
        pointing to nullptr as next and tail as prev
    if(!head) // if the list is empty, set the new node as the head
        head = newNode;
    if(tail) // if tail exists, update its next pointer to point to the new node
        tail->next = newNode;
    tail = newNode; // set the tail to the new node
    length++; // increasing the number of elements in the array
}
```

2.4.1.3 Dodawanie w wybranym miejscu

Przed dodaniem elementu w wybrane miejsce, funkcja sprawdza poprawność indeksu. Jeśli indeks mieści się w zakresie znajdywany jest istniejący węzeł na podstawie podanego indeksu. Tworzony jest nowy węzeł z danym elementem, który jest wstawiany pomiędzy istniejące węzły. Jeśli istnieje poprzedni węzeł dla węzła na podanej pozycji, to jego wskaźnik na następny węzeł ustawiany jest na nowy węzeł. Podobnie, jeśli istnieje następny węzeł dla węzła na podanej pozycji, to jego wskaźnik na poprzedni węzeł ustawiany jest na nowy węzeł. Następnie, jeśli węzeł na podanej pozycji był głową listy, nowy węzeł staje się nową głową, a jeśli był ogonem, staje się nowym ogonem. Długość listy zwiększa się o 1.

```
virtual void addAt(int index, T element) override{
    if(index < 0 || index > length){ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    Node* existingNode = getNode(index); // get the node at the specified index
    Node* newNode = new Node(element, existingNode, existingNode->prev); // new node
        with the element, pointing to the existingNode as next and its prev as prev
    if(existingNode->prev) // if existingNode's previous node exists, update its next
        pointer to point to the new node
        existingNode->prev->next = newNode;
    if(existingNode) // update the prev of existingNode to point to the new node
        existingNode->prev = newNode;
    if(existingNode == head) // if existingNode is head, update head to the new node
        head = newNode;
    if(existingNode == tail) // if existingNode is tail, update tail to the new node
        tail = newNode;
    length++; // increasing the number of elements in the array
}
```

2.4.2 Usuwanie elementów

2.4.2.1 Usuwanie na początku

Przed usunięciem elementu, występuje sprawdzenie czy lista nie jest pusta. Jeśli nie, to funkcja usuwa pierwszy element listy, przesuwając głowę (head) na następny węzeł. W przypadku, gdy lista po usunięciu staje się pusta, funkcja ta dodatkowo ustawia wskaźnik `tail` na `nullptr`.

```
virtual void removeFirst() override{
    if(!head){ // checking if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    Node* toDelete = head; // pointer to the head of the list
    head = head->next; // assigning the head to the next element
    if(!head) // if the list is empty after deletion, update the tail pointer
        tail = nullptr;
    else // set the previous pointer of the new head to null
        head->prev = nullptr;
    delete toDelete; // deleting the previous head of the list
    length--; // decreasing the length of the list
}
```

2.4.2.2 Usuwanie na końcu

Najpierw sprawdzane jest, czy lista nie jest pusta. Jeśli lista zawiera tylko jeden element, jest on usuwany, a wskaźniki głowy i ogona zostają ustawione na `nullptr`.Dla list dłuższych, wskaźnik `tail` jest przesuwany na przedostatni element, którego wskaźnik `next` jest zerowany, odłączając ostatni element. Następnie, ostatni element jest usuwany a liczba elementów listy zmniejsza się o jeden.

```
virtual void removeLast() override{
    if(!head){ // checking if the list is empty
        cout << "List is empty" << endl;
        return;
    }
    if(length == 1){ // check if the list contains only one element
        delete head;
        head = nullptr;
        tail = nullptr;
        length--;
        return;
    }
    Node* temp = tail; // temporary pointer to the tail of the list
    tail = tail->prev; // update the tail to the previous node.
    tail->next = nullptr; // set the next pointer of the new tail to null
    delete temp; // deleting the previous head of the list
    length--; // decreasing the length of the list
}
```

2.4.2.3 Usuwanie z wybranego miejsca

Przed usunięciem elementu, funkcja sprawdza indeks. Jeśli to pierwszy element, używana jest `removeFirst()`, jeśli ostatni - `removeLast()`. W innym przypadku przeszukuje listę, by znaleźć węzeł o danym indeksie. Następnie aktualizuje wskaźniki sąsiednich węzłów tak aby pomijać usuwany węzeł. Po tym usuwany jest element i zmniejszana długość listy.

```
virtual void removeAt(int index) override{ // checking if the index is within bounds
        cout << "Index out of bounds" << endl;
        return;
    }
    if(index == 0)
        removeFirst();
    else if(index == length - 1)
        removeLast();
    else {
        Node* toDelete = getNode(index); // get the node at the specified index
        // adjust pointers of previous and next nodes to skip the node to be deleted:
        toDelete->prev->next = toDelete->next;
        toDelete->next->prev = toDelete->prev;
        delete toDelete; // delete the node at the specified index
        length--; // decreasing the length of the list
    }
}
```

2.5 Wyszukiwanie elementu

2.5.1 Dla tablicy

Funkcja przechodzi przez każdy element tablicy sprawdzając czy jest on taki sam jak poszukiwany element.

```
virtual bool contains(const T& element) override{
    for(int i = 0; i < length; i++)
        if(arr[i] == element) // check if current element matches the target element
            return true;
    return false;
}
```

2.5.2 Dla list

Funkcja przechodzi przez każdy węzeł listy sprawdzając czy jest on równy poszukiwanemu elementowi.

```
virtual bool contains(const T& element) override{
    Node* temp = head; // start from the head of the linked list
    while(temp){
        if(temp->data == element) // if current node's data matches the target element
            return true;
        temp = temp->next; // move to the next node
    }
    return false;
}
```

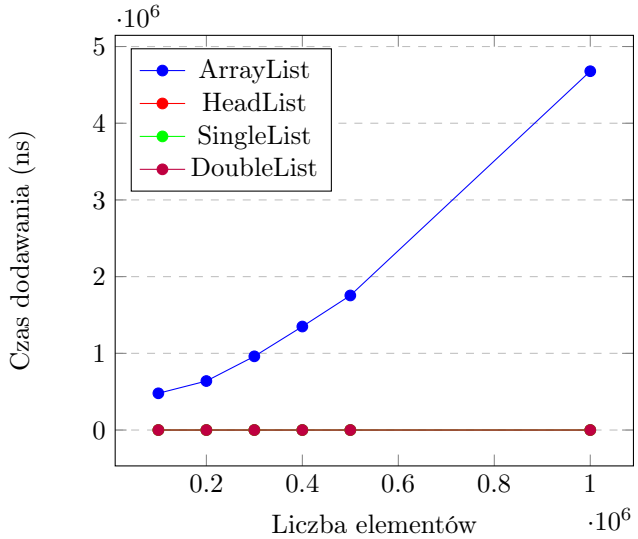
3 Badania

3.1 Dodawanie elementów dla wszystkich funkcji

3.1.1 Dodawanie na początek

Tabela 1: Porównanie czasów dodawania elementu na początku (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	478834	38.7744	18.0668	39.8153
200000	638762	17.1821	18.0568	19.3792
300000	960953	39.8404	17.6921	18.0807
400000	1349370	36.455	35.1866	18.0937
500000	1754400	17.0879	17.697	18.1088
1000000	4678120	30.9548	17.7211	18.1499



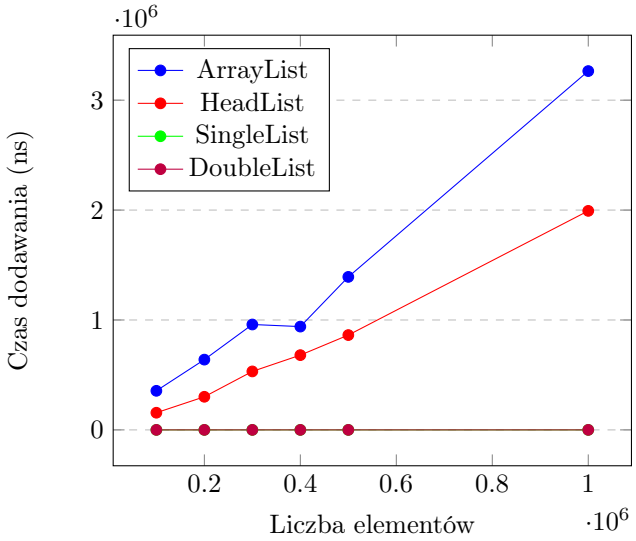
Rysunek 2: Porównanie czasów dodawania elementu na początku (w nanosekundach)

Na powyższym wykresie przebieg funkcji `HeadList`, `SingleList` i `DoubleList` pokrywają się.

3.1.2 Dodawanie na koniec

Tabela 2: Porównanie czasów dodawania elementu na końcu (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	355554	156072	18.7119	19.6076
200000	639294	301292	19.0616	19.6026
300000	958969	531983	18.6779	19.5745
400000	939844	680229	18.6669	19.5846
500000	1392020	862730	18.7059	19.6306
1000000	3264740	1992720	20.6806	39.3675



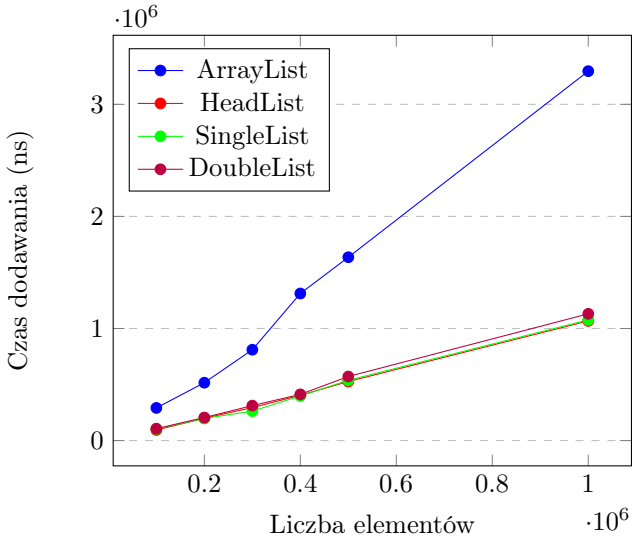
Rysunek 3: Porównanie czasów dodawania elementu na końcu (w nanosekundach)

Na powyższym wykresie przebieg funkcji `SingleList` i `DoubleList` pokrywają się.

3.1.3 Dodawanie w wybranym miejscu

Tabela 3: Porównanie czasów dodawania elementu w wybranym miejscu (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	291164	94466	103322	106228
200000	516464	198871	197689	205674
300000	810493	294460	262660	311852
400000	1312140	404034	397953	412380
500000	1635810	527074	537954	571557
1000000	3296020	1067640	1076510	1130840



Rysunek 4: Porównanie czasów dodawania elementu w wybranym miejscu (w nanosekundach)

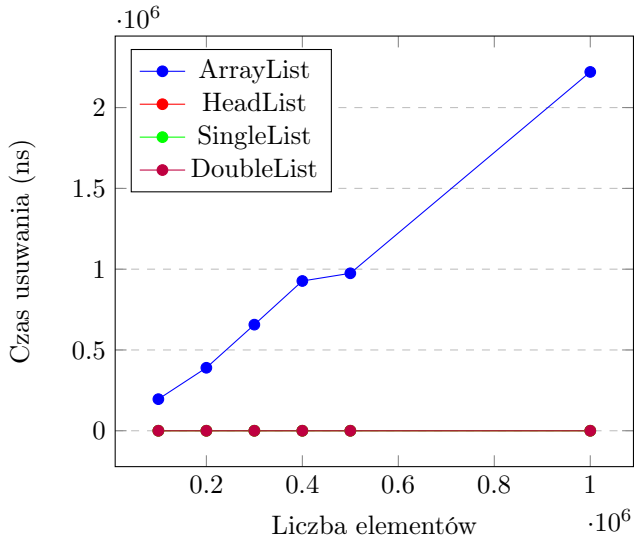
Na powyższym wykresie przebieg funkcji `HeadList`, `SingleList` i `DoubleList` pokrywają się.

3.2 Usuwanie elementów dla wszystkich funkcji

3.2.1 Usuwanie na początku

Tabela 4: Porównanie czasów usuwania elementu na początku (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	195605	10.3674	10.6539	20.2748
200000	389837	10.4565	10.6959	20.7477
300000	656646	10.4365	10.703	23.7674
400000	926861	10.4365	10.702	20.2187
500000	974519	13.0353	10.5687	21.9679
1000000	2220620	10.5146	10.8804	19.8721



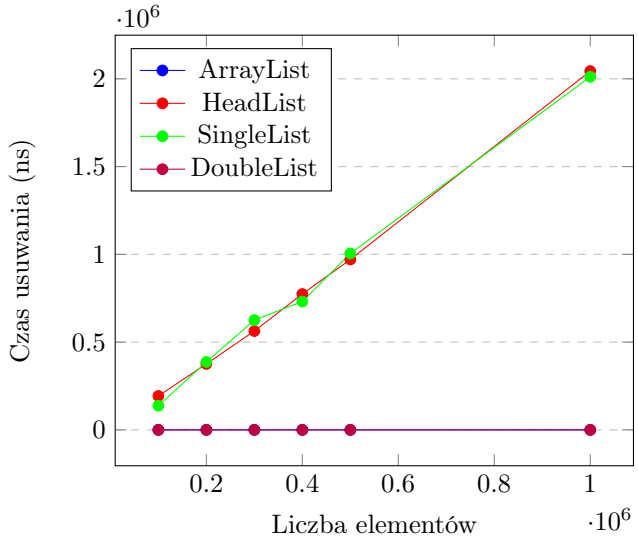
Rysunek 5: Porównanie czasów usuwania elementu na początku (w nanosekundach)

Na powyższym wykresie przebieg funkcji `HeadList`, `SingleList` i `DoubleList` pokrywają się.

3.2.2 Usuwanie na końcu

Tabela 5: Porównanie czasów usuwania elementu na końcu (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	161	193131	137747	13.9189
200000	260	375821	386862	13.895
300000	151	563202	625217	12.0094
400000	251	773994	731295	14.0052
500000	130	970732	1004830	14.0582
1000000	200	2044200	2011640	13.9801



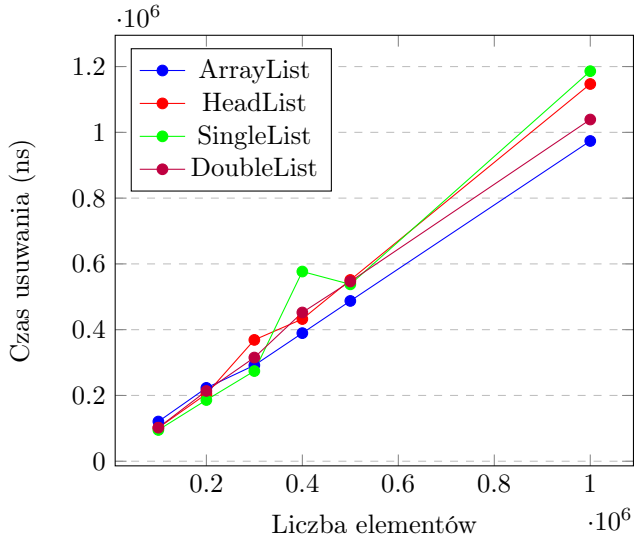
Rysunek 6: Porównanie czasów usuwania elementu na końcu (w nanosekundach)

Na powyższym wykresie przebieg funkcji `ArrayList` i `DoubleList` pokrywają się.

3.2.3 Usuwanie z wybranego miejsca

Tabela 6: Porównanie czasów usuwania elementu z wybranego miejsca (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	120695	101709	95028	102020
200000	222977	204522	186107	213709
300000	291705	368878	274171	315249
400000	389467	432317	576757	452295
500000	487369	551169	537824	546520
1000000	973727	1147000	1186050	1039000



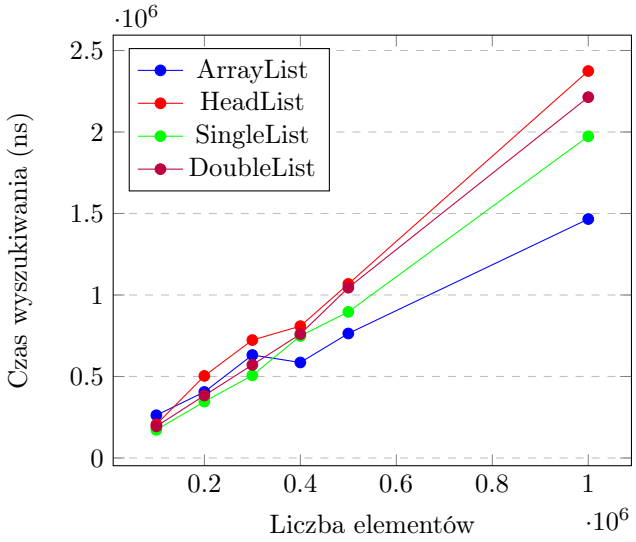
Rysunek 7: Porównanie czasów usuwania elementu z wybranego miejsca (w nanosekundach)

3.3 Wyszukiwanie elementu

W każdym przypadku wzięto element, który nie istnieje dlatego w każdym przypadku funkcja przeszukuje wszystkie elementy.

Tabela 7: Porównanie czasów wyszukiwania elementu (w nanosekundach)

Liczba elementów	ArrayList	HeadList	SingleList	DoubleList
100000	262157	205753	172682	194152
200000	405075	503479	345424	383486
300000	631627	723388	506754	571125
400000	585993	808888	748636	760087
500000	763704	1067500	896141	1044740
1000000	1465940	2374070	1973530	2213720



Rysunek 8: Porównanie czasów wyszukiwania elementu (w nanosekundach)

4 Wnioski

Zgodnie z oczekiwaniami, część teoretyczna pokryła się z częścią praktyczną, czyli naszymi badaniami. Wszystkie funkcje zachowują się zgodnie z założeniami. Na przykład, funkcje o złożoności O(n) wykazują wzrost czasu wykonania wraz ze wzrostem liczby elementów, podczas gdy te o złożoności O(1) utrzymują stały czas wykonania niezależnie od liczby elementów. Analizując wyniki badań, możemy zauważyć, że najbardziej efektywnym rozwiązaniem okazała się lista dwukierunkowa, posiadająca wskaźniki zarówno na głowę, jak i ogon. To rozwiązanie nie tylko eliminuje konieczność określania z góry wielkości struktury, ale także charakteryzuje się najkrótszym czasem wykonania we wszystkich funkcjach.