P O L I T E C H N I K A R Z E S Z O W S K A
im. Ignacego Łukasiewicza
WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ

Damian Stechnij
Nr albumu: 173219

PROJEKT
ALGORYTMY I STRUKTURY DANYCH

kierunek studiów: inżynieria i analiza danych

Rzeszów, 2023

Spis treści

1 TEMAT PRACY

Dokonaj implementacji struktury danych typu lista dwukierunkowa wraz z wszelkimi potrzebnymi operacjami charakterystycznymi dla tej struktury (inicjowanie struktury, dodawanie/usuwanie elementów, wyświetlanie elementów, zliczanie elementów/wyszukiwanie zadanego elementu itp.).

2 PROJEKTOWANIE

2.1 Problem zagadnienia

Lista dwukierunkowa to struktura danych, w której każdy element (węzeł) zawiera informację o poprzednim i następnym elemencie. Dzięki temu, możliwe jest przeglądanie listy zarówno w kierunku od pierwszego do ostatniego elementu, jak i od ostatniego do pierwszego elementu.

Struktura listy dwukierunkowej składa się z dwóch elementów:

- Wartości (dane przechowywane przez element),
- Dwóch wskaźników (next i prev), które odpowiednio wskazują na następny i poprzedni element w liście.

W przeciwieństwie do listy jednokierunkowej, gdzie tylko jeden wskaźnik jest potrzebny, w liście dwukierunkowej potrzebne są dwa wskaźniki, co zwiększa zużycie pamięci.

2.2 Pseudokod

2.2.1 Definiowanie struktury

```
Zdefiniuj typ danych "ListaElementow_typ" jako strukturę zawierającą trzy pola:
"data", "previous" i "next".
Pole "data" przechowuje int.
Pole "previous" jest wskaźnikiem na poprzedni element listy.
Pole "next" jest wskaźnikiem na następny element listy.
```

2.2.2 Funkcja dodawania na początek listy

```
Wejście (ListaElementow_typ **glowa, liczba)
    jeżeli lista jest pusta wykonaj
        alokuj pamięć dla nowego elementu
        przypisz wartość "liczba" do pola "data" nowego elementu
        ustaw pole "poprzedni" na NULL
        ustaw pole "następny" na NULL
        przypisz adres nowego elementu do wskaźnika "glowa"
    w przeciwnym razie
        zdefiniuj wskaźnik current do nowego elementu
        zaalokuj pamięć dla nowego elementu
        przypisz wartość "number" do pola "data" nowego elementu.
        ustaw pole "poprzedni" na NULL.
        ustaw pole "następny" nowego elementu na obecny "head"
        ustaw pole "poprzedni" obecnego "head" na "current"
        przypisz adres nowego elementu do wskaźnika "head"
```

2.2.3 Funkcja dodawania na koniec listy

```
Wejście (ListaElementow_typ **glowa, liczba, indeks)
    Jeżeli indeks==0 wykonaj
        użyj funkcji add_front z paramentrami "glowa" i "liczba"
    w przeciwnym razie
        jeżeli indeks jest równy rozmiarowi listy wykonaj
            użyj add_back z parametrami "glowa" "number"
        zdefiniuj wskaźnik "current" jako "glowa"
        zdefiniuj zmienną "i" jako 0
        przejdź przez listę, aż nie dotrzesz do elementu poprzedzającego indeks
        zdefiniuj wskaźnik "tmp" jako następny element po "current"
        zaalokuj pamięć dla nowego elementu
        przypisz wartość "liczba" do pola "data" nowego elementu
        ustaw pole "poprzedni" nowego elementu na "current"
        ustaw pole "poprzedni" tmp na nowy element
        ustaw pole "następny" nowego elementu na "tmp"
```

2.2.4 Funkcja dodawania w wybranym indeksie

```
Wejście (ListaElementow_typ **glowa, liczba)
    jeżeli lista jest pusta wykonaj
        alokuj pamięć dla nowego elementu
        przypisz wartość "liczba" do pola "data" nowego elementu
        ustaw pole "poprzedni" na NULL
        ustaw pole "następny" na NULL
        przypisz adres nowego elementu do wskaźnika "glowa"
    w przeciwnym razie
        zdefiniuj wskaźnik "current" jako "glowa"
        przejdź przez listę, aż nie dotrzesz do ostatniego elementu
        zaalokuj pamięć dla nowego elementu
        przypisz wartość "liczba" do pola "data" nowego elementu
        ustaw pole "następny" obecnego ostatniego elementu na nowy element
        ustaw pole "poprzedni" nowego elementu na obecny ostatni element
        ustaw pole "następny" nowego elementu na NULL
```

2.2.5 Funkcja usuwania z początku listy

```
Wejście (ListaElementow_typ **glowa)
    Jeżeli lista nie jest pusta wykonaj
        Jeżeli pierwszy element jest jedynym na liście
            ustaw wskaźnik "glowa" na NULL
        w przeciwnym razie
            Zdefiniuj wskaźnik "tmp" jako następny element po "glowa"
            zwolnij zaalokowaną pamięć dla pierwszego elementu
            ustaw   "glowa" na "tmp"
            ustaw pole "poprzedni" nowej głowy na NULL
```

2.2.6 Funkcja usuwania z końca listy

```
Wejście (ListaElementow_typ **glowa)
    Jeżeli next po head jest równy NULL wykonaj
        Ustaw head na NULL
    w przeciwnym razie
        Zdefiniuj wskaźnik "current" jako head
        Przejdź przez listę, aż nie dotrzesz do przedostatniego elementu.
        Zwolnij pamięć zaalokowaną dla ostatniego elementu
        Ustaw pole next przedostatniego elementu na NULL
```

2.2.7 Funkcja usuwania wybierając indeks

```
Wejście (ListaElementow_typ **glowa, indeks)
    Jeżeli indeks==0 wykonaj
        Użyj funkcji delete_front() z parametrem "head"
    w przeciwnym razie
        Zdefiniuj wskaźnik "current" jako head
        Zdefiniuj zmienną "i" jako 0
        Przejdź przez listę, aż nie dotrzesz do elementu poprzedzającego pozycję
        Zdefiniuj wskaźnik "tmp" jako następny element po "current"
        Ustaw pole "next" elementu poprzedzającego pozycję, na pole "next" tymczasowego elementu (tmp)
        Ustaw pole "previous" elementu po tymczasowym elementu na current
        Zwolnij pamięć zaalokowaną dla tymczasowego elementu (tmp)
```

2.2.8 Funkcja pokazująca

```
Wejście (ListaElementow_typ *glowa)
    Wypisz pustą linię
    Jeżeli lsita jest pusta wykonaj
        Wypisz "Lista jest pusta"
    W przeciwnym razie
        Zdefiniuj wskaźnik "current" jako head
        Dopóki current != NULL wykonuj
            Wypisz pole data elementu "current"
            Wypisz pustą linię
            Przypisz pole "next" elementu "current" do "current"
```

2.2.9 Funkcja pokazująca w odwrotnej kolejności

```
Wejście (ListaElementow_typ *glowa)
    Wypisz pustą linię
    Jeżeli lista jest pusta wykonaj
        Wypisz  "Lista jest pusta"
    w przeciwnym razie
        Zdefiniuj wskaźnik "current" jako head
        Przejdź przez listę, aż nie dotrzesz do ostatniego elementu
        Dopóki current != NULL wykonuj
            Wypisz pole data elementu "current"
            Wypisz pustą linię
            Przypisz pole "previous" elementu "current" do "current"
```

2.2.10 Funkcja licząca rozmiar listy

```
Wejście (ListaElementow_typ *glowa)
    Zdefiniuj licznik jako 0
    Jeżeli lista jest pusta wykonaj
        Zwróć licznik
    w przeciwnym razie
        Zdefiniuj wskaźnik "current" jako head
        Dopóki current != NULL wykonuj
            Zwiększ licznik o 1
            Przypisz pole "next" elementu "current" do "current"
    Zwróć licznik
```

2.3 Kod źródłowy programu oraz opis działania funkcji

2.3.1 Funkcja type_dev struct

```
6   // Zdefiniowanie struktury
7   typedef struct ListElement {
8       int data;   //pole przechowujace dana
9       struct ListElement * previous;  //pole wskazujace na poprzedni element listy
10      struct ListElement * next;  //pole wskazujace na nastepny element listy
11  } ListElement_type;
```

2.3.2 Funkcja void add_front

```
115 // Dodawanie elementu na poczatek listy
116 void add_front(ListElement_type **head, int number) {
117     if(*head==NULL) {   // jesli lista jest pusta
118         // alokujemy pamiec dla nowego elementu
119         *head = (ListElement_type *)malloc(sizeof(ListElement_type));
120         (*head)->data = number; // przypisanie wartosci
121         (*head)->previous=NULL; // poprzedni element ustawiamy na NULL
122         (*head)->next = NULL;   // nastepny element ustawiamy na NULL
123     } else {    // jesli cos jest na liscie
124         ListElement_type *current;  // tworzymy wskaznik dla nowego elementu
125         // alokujemy pamiec dla nowego elementu
126         current=(ListElement_type *)malloc(sizeof(ListElement_type));
127         current->data=number;   // przypisanie wartosci
128         current->previous=NULL; // poprzedni element ustawiamy na NULL
129         // zamiana nowego elementu na head
130         current->next=(*head);
131         (*head)->previous=current;
132         *head=current;
133     }
134 }
```

Funkcja add_front przyjmuje jako parametry wskaźnik do wskaźnika head i liczbę number, która zostanie przypisana do pola data nowego elementu. Jeśli lista jest pusta, to funkcja tworzy nowy element i ustawia head na wskaźnik do niego. Jeśli lista nie jest pusta, to funkcja tworzy nowy element i ustawia jego next na obecny head, a previous obecnego head na nowy element. Następnie ustawia head na wskaźnik do nowego elementu.

2.3.3 Funkcja void add_back

```
136  // Dodawanie elementy na koncu listy
137  void add_back(ListElement_type **head, int number) {
138      if(*head==NULL) // gdy lista jest pusta
139      {
140          // alokujemy pamiec dla nowego elementu
141          *head = (ListElement_type *)malloc(sizeof(ListElement_type));
142          (*head)->data = number; // przypisanie wartosci
143          (*head)->previous = NULL;   // poprzedni element ustawiamy na NULL
144          (*head)->next = NULL;   // nastepny element ustawiamy na NULL
145      } else {    // jesli cos jest na liscie
146          ListElement_type *current=*head;
147          ListElement_type *new_element;
148          while (current->next != NULL) {
149              current = current->next;
150          }   // przechodzimy przez liste az nastepny element nie bedzie pusty
151          // lokujemy pamiec dla nowego elementu
152          current->next = (ListElement_type *)malloc(sizeof(ListElement_type));
153          current->next->data = number;   // przypisanie wartosci
154          current->next->previous=current;
155          current->next->next = NULL; // nastepny element ustawiamy na NULL
156      }
157  }
```

Funkcja add_back przyjmuje jako parametry wskaźnik do wskaźnika head i liczbę number, która zostanie przypisana do pola data nowego elementu. Jeśli lista jest pusta, to funkcja tworzy nowy element i ustawia head na wskaźnik do niego. Jeśli lista nie jest pusta, to funkcja tworzy nowy element i ustawia jego previous na ostatni element listy. Następnie ustawia next ostatniego elementu na nowy element.

2.3.4 Funkcja void add_by_index

```
159  // Dodawanie elementu o wybranym indeksie
160  void add_by_index(ListElement_type **head, int number, int position) {
161      // jesli indeks to 0, wykorzystujemy funkcje dodajaca elementu na poczatek listy
162      if(position==0) add_front(head, number);
163      else {
164          if(position==list_size(*head)) add_back(head, number);
165          else {
166          ListElement_type *current=*head;
167              ListElement_type *tmp;
168
169              int i=0;
170              while (current->next != NULL && i<position-1) {
171                  current = current->next;
172                  i++;
173              }   // przechodzimy przez liste do elementu poprzedniego od indeksu
174              tmp=current->next;  // tmp jako nastepny
175              // alokujemy pamiec
176              current->next=(ListElement_type *)malloc(sizeof(ListElement_type));
177              current->next->data=number; // przypisujemy wartosc
178              current->next->previous=current;
179              tmp->previous=current->next;
180              current->next->next=tmp;
181          }
182      }
183  }
```

Funkcja add_by_index przyjmuje jako parametry wskaźnik do wskaźnika head, liczbę number, która zostanie przypisana do pola data nowego elementu oraz pozycję na której nowy element ma zostać wstawiony. Jeśli pozycja jest równa 0, to funkcja używa funkcji add_front(). Jeśli pozycja jest równa rozmiarowi listy, to funkcja używa funkcji add_back(). Jeśli pozycja jest inną liczbą, to funkcja przechodzi przez listę, aż nie dotrze do elementu poprzedzającego pozycję, następnie tworzy nowy element i wstawia go na wybraną pozycję.

2.3.5 Funkcja void delete_front

```
185  // Usuwanie pierwszego elementu listy
186  void delete_front(ListElement_type **head) {
187      if (*head!=NULL) {  // jesli lista nie jest pusta
188          if((*head)->next==NULL) {   //jesli na liscie znajduje sie tylko jeden element
189          *head=NULL; // usuwamy element
190          } else {
191              ListElement_type *tmp;
192              tmp=(*head)->next;  // nastepny element po pierwszym przypisujemy do tmp
193              free(*head);    // zwalniamy pamiec
194              *head=tmp;      // tmp staje sie nowa glowa
195              (*head)->previous=NULL; //usuwamy pierwszy element
196          }
197      }
198  }
```

Funkcja delete_front usuwa pierwszy element z listy, przypisując następny element po pierwszym jako nowy head. Jeśli lista jest pusta, to funkcja nic nie robi. Jeśli pierwszy element jest jedynym elementem na liście, to funkcja ustawia head na NULL. Jeśli nie jest to jedyny element, to funkcja zwolnienia pamięci zaalokowanej dla pierwszego elementu, ustawia head na następny element i ustawia previous dla tego elementu na NULL.

2.3.6 Funkcja void delete_back

```
200   // Usuwanie elementu z konca listy
201   void delete_back(ListElement_type **head) {
202       if((*head)->next==NULL) { //jezeli na liscie jest tylko jeden element
203       *head=NULL; // lista staje sie pusta
204       } else {    //jezeli jest wiecej elementow na liscie
205           ListElement_type *current=*head;
206           while (current->next->next!= NULL) {
207                   current = current->next;
208           }   // przechodzimy do przedostatniego elementu
209           free(current->next);// zwolnienie pamieci
210           current->next=NULL; // usuniecie ostatniego elementu
211       }
212   }
```

Funkcja delete_back usuwa ostatni element z listy. Jeśli jest to jedyny element na liście, to funkcja ustawia head na NULL. Jeśli nie jest to jedyny element, to funkcja przechodzi przez listę, aż nie dotrze do przedostatniego elementu, zwalnia pamięć zaalokowaną dla ostatniego elementu i ustawia pole next przedostatniego elementu na NULL.

2.3.7 Funkcja void delete_by_index

```
214   // Usuwanie elementu o wybranym indeksie
215   void delete_by_index(ListElement_type **head, int position) {
216       // jesli indeks=0, wykonaj funkcje usuwajaca element z poczatku listy
217       if(position==0) delete_front(head);
218       else {  // jesli indeks jest inny niz 0
219           ListElement_type *current=*head;
220           ListElement_type *tmp;
221           int i=0;
222           while (current->next != NULL && i<position-1) {
223               current=current->next;
224               i++;
225           }   // przechodzimy po liscie do elementu poprzedzajacego wezel
226           tmp = current->next;    // tymczasowy element przechowuje element, ktory chcemy usunac
227           current->next = tmp->next;
228           current->next->previous=current;
229           free(tmp);  //zwolnij pamiec z tymczasowego elementu
230       }
231   }
```

Funkcja delete_by_index usuwa element z listy na określonej pozycji. Jeśli pozycja jest równa 0, to funkcja używa funkcji delete_front(). W przeciwnym razie, funkcja przechodzi przez listę, aż nie dotrze do elementu poprzedzającego pozycję, zdefiniowanie tymczasowego elementu jako następny element po "current", ustawienie pola next elementu poprzedzającego pozycję na pole next tymczasowego elementu, ustawienie pola previous elementu po tymczasowym elementu na current, zwalnianie pamięci zaalokowanej dla tymczasowego elementu.

2.3.8 Funkcja void show

```
233  // Pokazywanie listy od poczatku
234  void show(ListElement_type *head) {
235      printf("\n");
236      if(head==NULL) printf("Lista jest pusta");
237      else {
238          ListElement_type *current=head;
239          do {    //wypisywanie elementow po kolei
240              printf("%i", current->data);
241              printf("\n");
242              current = current->next;
243          } while (current != NULL);
244      }
245  }
```

Funkcja show wyświetla zawartość listy. Jeśli lista jest pusta, to funkcja wyświetla komunikat "Lista jest pusta". W przeciwnym razie, funkcja używa pętli do-while, aby przejść przez listę i wyświetlić pole data każdego elementu.

2.3.9 Funkcja void show_reverse

```
247  // Pokazanie listy od konca
248  void show_reverse(ListElement_type *head) {
249      printf("\n");
250      if(head==NULL) printf("Lista jest pusta");
251      else {
252          ListElement_type *current=head;
253          while (current->next != NULL) {
254              current = current->next;
255          }   // przechodzimy na koniec listy
256          do {    // wypisujemy elementy od konca listy
257              printf("%i", current->data);
258              printf("\n");
259              current = current->previous;
260          } while(current!=NULL);
261      }
262  }
```

Funkcja show_reverse wyświetla zawartość listy od końca. Jeśli lista jest pusta, to funkcja wyświetla komunikat "Lista jest pusta". W przeciwnym razie, funkcja przechodzi przez listę, aż nie dotrze do ostatniego elementu, a następnie używa pętli do-while, aby przejść przez listę od końca i wyświetlić pole data każdego elementu.

2.3.10 Funkcja int list_size

```
264  // Sprawdzanie ile elementow jest na liscie
265  int list_size(ListElement_type *head) {
266      int counter=0;
267      if(head==NULL) return counter;  // jesli lista jest pusta zwraca 0
268      else {  // w przeciwnym razie
269          ListElement_type *current=head;
270          do {// przechodzac przez liste liczy elementy
271              counter++;
272              current = current->next;
273          } while (current != NULL);
274      }
275      return counter; // zwraca liczbe elementow
276  }
```

Funkcja list_size zwraca liczbę elementów w liście. Jeśli lista jest pusta, zwraca 0. W przeciwnym razie, funkcja przechodzi przez listę i zwiększa licznik o 1 za każdym przejściem przez pętlę. Gdy przejdzie przez całą listę, zwraca liczbę elementów.

3 DZIAŁANIE PROGRAMU

Ten program jest implementacją listy dwukierunkowej. Struktura ListElement zawiera pola: data (przechowujące daną int), previous (wskazujące na poprzedni element listy) oraz next (wskazujące na następny element listy). W kodzie zdefiniowane są również funkcje, które pozwalają na operacje na liście takie jak: dodawanie elementów na początku, końcu, oraz w wybranej pozycji, usuwanie elementów z początku, końca oraz w wybranej pozycji, wyświetlanie listy oraz jej odwrotności.

Po odpaleniu programu pojawia się menu, dzięki któremu można wybrać operację, którą program ma wykonać. Powyżej menu znajduje się podgląd wprowadzanej listy.

3.1 Zaprezentowanie działania

```
------------------
Lista dwukierunkowa
------------------

Poglad listy:
Lista jest pusta


1. Dodaj element na poczatek listy.
2. Dodaj element na koniec listy.
3. Dodaj element o wybranym indeksie.
4. Usun element z poczatku listy.
5. Usun element z konca listy.
6. Usun element o wybranym indeksie.
7. Wyswietl liste w odwrotnej kolejnosci.
0. Zakoncz program.

Wybierz operacje jaka chcesz wykonac:
```

Widok programu w konsoli

```
--------------------
Lista dwukierunkowa
--------------------

Poglad listy:
5646
123
43
3876
2
3123
764578
987
```

Podgląd listy po wprowadzeniu kilku danych

4 WNIOSKI

Zainicjowano strukturę danych jaką jest lista dwukierunkowa, której kod pozwala na dodawanie i usuwanie wybranych elementów, oraz zliczanie elementów listy. Podstawowym typem danych jaki element struktury przechowuje są dane typu int.

Funkcja main() zawiera przedstawione możliwości biblioteki.

Kod zawiera stosowne komentarze opisujące kod.

Kod wraz z sprawozdaniem został umieszczony na serwer GitHub, do którego link został wysłany poprzez mail.