Politechnika Świętokrzyska
w Kielcach
Wydział Elektrotechniki, Automatyki i Informatyki

Podstawy programowania 2

Instrukcja laboratoryjna 4

„Jednokierunkowa i dwukierunkowa lista liniowa”

Przygotowali:
mgr inż. Paweł Pięta
dr inż. Arkadiusz Chrobot

Kielce, 2023

1 Wstęp

Celem zajęć jest praktyczne zapoznanie się z dynamicznymi strukturami danych jednokierunkowej i dwukierunkowej listy liniowej.

2 Jednokierunkowa lista liniowa

Pierwsza abstrakcyjna struktura danych, która zostanie omówiona w ramach tej instrukcji, to jednokierunkowa lista liniowa (ang. *singly linked list*). Dane mogą być w niej przechowywane zarówno w formie uporządkowanej, jak również nieuporządkowanej, a dostęp do nich jest sekwencyjny i można je przeglądać tylko w jedną stronę. Jednym ze sposobów jej implementacji jest dynamiczna struktura elementów powiązanych ze sobą za pomocą wskaźników, posiadająca początek i koniec. Przykład definicji typu bazowego dla dynamicznej struktury jednokierunkowej listy liniowej, przechowującej liczby typu `int`, został przedstawiony w kodzie źródłowym 1 w liniach 4–8. Podstawowe operacje, które będą wykonywane na strukturze jednokierunkowej listy, to:

1. utworzenie nowej listy – operacja *create*,
2. dodanie nowego elementu do listy – operacja *insert* lub *add*,
3. usunięcie elementu z listy – operacja *delete*,
4. wypisanie zawartości elementów listy – operacja *print*,
5. usunięcie listy – operacja *remove*.

Do obsługi jednokierunkowej listy liniowej wystarczy tylko jeden wskaźnik wskazujący na początek listy (ang. *front*). Nie jest natomiast konieczne pamiętanie w programie wskaźnika do końca listy (ang. *back*). Do utworzenia listy i dodania do niej pierwszego elementu służy funkcja `create_list()`, natomiast `remove_list()` usuwa całą listę. Liczby będą w niej przechowywane w sposób nierosnący. Operacje *insert* i *delete* mogą być przeprowadzane w dowolnym miejscu listy. Implementacja funkcji `insert_node()` i `delete_node()` została zaprezentowana w kodzie źródłowym 1 w liniach 51–74 i 104–115. Aby funkcje te zadziałały poprawnie, przekazany im przez parametr wskaźnik na początek listy musi wskazywać na istniejącą listę lub być pusty, jednakże w przypadku wartości `NULL`, w przeciwieństwie do kolejki, nowy element nie zostanie dodany do listy. Ponieważ dodanie nowego elementu będzie przebiegać inaczej na początku, w środku i na końcu listy, poszczególne przypadki będą obsługiwane przez trzy funkcje pomocnicze: `insert_front()`, `insert_after()` i `insert_back()`. To samo dotyczy usunięcia elementu, z tą różnicą, że usuwanie z środka i z końca listy jest realizowane w ten sam sposób, dlatego operacja ta będzie wykonywana za pomocą dwóch funkcji pomocniczych: `delete_front()` i `delete_after()`.

Tak jak w przypadku kolejki, w celu uproszczenia kodu zarządzającego listą możliwe jest wykorzystanie wartowników. Jeden z nich powinien znajdować się na początku listy, a drugi na końcu. Typ bazowy listy z wartownikami jest taki sam jak zwykłej listy, jednakże oprócz wskaźnika na początek listy należy również zapamiętać w programie wskaźnik na koniec listy, co zostało pokazane w kodzie źródłowym 2. Pole `next` wartownika umieszczonego na końcu listy zazwyczaj wskazuje na niego samego. W pustej liście będą się znajdować jedynie elementy wartowników.

Ponieważ wskaźniki `front` i `back` listy z wartownikami zawsze wskazują na odpowiednie elementy wartowników, podczas wykonywania operacji na liście nie ma potrzeby ich modyfikowania (za wyjątkiem usuwania całej listy). Ponadto operacje dodania nowego elementu oraz usunięcia dowolnego elementu będą przebiegać zawsze w ten sam sposób, co znacząco skraca ich zapis w kodzie programu.

Kod źródłowy 1: Przykładowy program z jednokierunkową listą liniową

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct sll_node
5  {
6      int data;
7      struct sll_node *next;
8  };
9
10 struct sll_node *create_list(int data)
11 {
12     struct sll_node *front = (struct sll_node *)
13         malloc(sizeof(struct sll_node));
14     if (NULL != front)
15     {
16         front->data = data;
17         front->next = NULL;
18     }
19     return front;
20 }
21
22 struct sll_node *insert_front(struct sll_node *front,
23                               struct sll_node *new_node)
24 {
25     new_node->next = front;
26     return new_node;
27 }
28
29 struct sll_node *find_spot(struct sll_node *front, int data)
30 {
31     struct sll_node *prev = NULL;
32     while ((NULL != front) && (front->data > data))
33     {
34         prev = front;
35         front = front->next;
36     }
37     return prev;
```

```
38 }
39
40 void insert_after(struct sll_node *node, struct sll_node *new_node)
41 {
42     new_node->next = node->next;
43     node->next = new_node;
44 }
45
46 void insert_back(struct sll_node *back, struct sll_node *new_node)
47 {
48     back->next = new_node;
49 }
50
51 struct sll_node *insert_node(struct sll_node *front, int data)
52 {
53     if (NULL == front)
54         return NULL;
55
56     struct sll_node *new_node = (struct sll_node *)
57         malloc(sizeof(struct sll_node));
58     if (NULL != new_node)
59     {
60         new_node->data = data;
61         new_node->next = NULL;
62         if (front->data <= data)
63             return insert_front(front, new_node);
64         else
65         {
66             struct sll_node *node = find_spot(front, data);
67             if (NULL != node->next)
68                 insert_after(node, new_node);
69             else
70                 insert_back(node, new_node);
71         }
72     }
73     return front;
74 }
75
76 struct sll_node *delete_front(struct sll_node *front)
77 {
78     struct sll_node *next = front->next;
79     free(front);
80     return next;
81 }
82
83 struct sll_node *find_prev_node(struct sll_node *front, int data)
84 {
85     struct sll_node *prev = NULL;
86     while ((NULL != front) && (front->data != data))
87     {
88         prev = front;
89         front = front->next;
90     }
91     return prev;
92 }
93
94 void delete_after(struct sll_node *node)
95 {
```

```
96      struct sll_node *next = node->next;
97      if (NULL != next)
98      {
99          node->next = next->next;
100         free(next);
101     }
102 }
103
104 struct sll_node *delete_node(struct sll_node *front, int data)
105 {
106     if (NULL == front)
107         return NULL;
108
109     if (front->data == data)
110         return delete_front(front);
111
112     struct sll_node *prev = find_prev_node(front, data);
113     delete_after(prev);
114     return front;
115 }
116
117 void print_list(struct sll_node *front)
118 {
119     for (; NULL != front; front = front->next)
120         printf("%d ", front->data);
121     printf("\n");
122 }
123
124 void remove_list(struct sll_node **front)
125 {
126     while (NULL != *front)
127     {
128         struct sll_node *next = (*front)->next;
129         free(*front);
130         *front = next;
131     }
132 }
133
134 int main()
135 {
136     struct sll_node *front = create_list(1);
137     int i;
138
139     for (i=2; i<10; i++)
140         front = insert_node(front, i);
141     printf("List elements:\n");
142     print_list(front);
143
144     front = insert_node(front, 0);
145     printf("List elements after insertion of 0:\n");
146     print_list(front);
147     front = insert_node(front, 5);
148     printf("List elements after insertion of 5:\n");
149     print_list(front);
150     front = insert_node(front, 10);
151     printf("List elements after insertion of 10:\n");
152     print_list(front);
153
```

```
154     front = delete_node(front, 0);
155     printf("List elements after deletion of 0:\n");
156     print_list(front);
157     front = delete_node(front, 1);
158     printf("List elements after deletion of 1:\n");
159     print_list(front);
160     front = delete_node(front, 1);
161     printf("List elements after deletion of 1:\n");
162     print_list(front);
163     front = delete_node(front, 5);
164     printf("List elements after deletion of 5:\n");
165     print_list(front);
166     front = delete_node(front, 10);
167     printf("List elements after deletion of 10:\n");
168     print_list(front);
169 
170     remove_list(&front);
171     return 0;
172 }
```

Kod źródłowy 2: Szablon programu jednokierunkowej listy liniowej z wartownikami

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  
5  struct sll_node
6  {
7      int data;
8      struct sll_node *next;
9  };
10 
11 struct sll_with_guards
12 {
13     struct sll_node *front, *back;
14 };
15 
16 bool create_list(struct sll_with_guards *sll)
17 {
18     struct sll_node *front = (struct sll_node *)
19         malloc(sizeof(struct sll_node));
20     struct sll_node *back = (struct sll_node *)
21         malloc(sizeof(struct sll_node));
22     if ((NULL != front) && (NULL != back))
23     {
24         sll->front = front;
25         sll->back = front->next = back->next = back;
26         return true;
27     }
28     free(front);
29     free(back);
30     return false;
31 }
32 
33 void remove_list(struct sll_with_guards *sll)
34 {
35     while (sll->front != sll->back)
```

```
36      {
37          struct sll_node *next = sll->front->next;
38          free(sll->front);
39          sll->front = next;
40      }
41      free(sll->back);
42      sll->front = sll->back = NULL;
43  }
44
45  int main()
46  {
47      struct sll_with_guards sll = {NULL, NULL};
48
49      if (create_list(&sll))
50      {
51          // Wykonaj jakieś operacje na liście.
52          remove_list(&sll);
53      }
54      return 0;
55  }
```

3 Dwukierunkowa lista liniowa

Kolejna abstrakcyjna struktura danych, która zostanie omówiona w ramach tej instrukcji, to dwukierunkowa lista liniowa (ang. *doubly linked list*). Dane mogą być w niej przechowywane zarówno w formie uporządkowanej, jak również nieuporządkowanej, a dostęp do nich jest sekwencyjny i można je przeglądać w dwie strony. Jednym ze sposobów jej implementacji jest dynamiczna struktura elementów powiązanych ze sobą za pomocą wskaźników, posiadająca początek i koniec. Przykład definicji typu bazowego dla dynamicznej struktury dwukierunkowej listy liniowej, przechowującej liczby typu `int`, został przedstawiony w kodzie źródłowym 3 w liniach 4–8. Lista dwukierunkowa wspiera te same podstawowe operacje co lista jednokierunkowa.

Do obsługi dwukierunkowej listy liniowej wystarczy tylko jeden wskaźnik wskazujący na początek listy, aczkolwiek równie dobrze może on wskazywać na dowolny element listy (przechowywanie adresu pierwszego elementu jest jednak bardziej praktyczne). Nie jest natomiast konieczne pamiętanie w programie wskaźnika do końca listy. Tak jak w przypadku jednokierunkowej listy, do utworzenia jej dwukierunkowej odmiany i dodania do niej pierwszego elementu służy funkcja `create_list()`, natomiast `remove_list()` usuwa całą listę. Liczby będą w niej przechowywane w sposób nierosnący. Operacje *insert* i *delete* mogą być przeprowadzane w dowolnym miejscu listy. Implementacja funkcji `insert_node()` i `delete_node()` została zaprezentowana w kodzie źródłowym 3 w liniach 55–78 i 109–126. Aby funkcje te zadziałały poprawnie, przekazany im przez parametr wskaźnik na początek listy musi wskazywać na istniejącą listę lub być pusty, jednakże dla wartości `NULL` nowy element nie zostanie dodany do listy, tak jak miało to miejsce w przypadku jednokierunkowej listy liniowej. Ponieważ dodanie nowego elementu będzie przebiegać inaczej na początku,

w środku i na końcu listy, poszczególne przypadki będą obsługiwane przez trzy funkcje pomocnicze: `insert_front()`, `insert_after()` i `insert_back()`. To samo dotyczy usunięcia elementu, dlatego operacja ta również będzie wykonywana za pomocą trzech funkcji pomocniczych: `delete_front()`, `delete_within()` i `delete_back()`. W przeciwieństwie do jednokierunkowej listy liniowej, możliwe jest także łatwe zaimplementowanie wypisania zawartości listy w odwrotnej kolejności, co realizuje funkcja `print_list_backwards()`.

Tak jak w przypadku kolejki i jednokierunkowej listy liniowej, w celu uproszczenia kodu zarządzającego listą dwukierunkową możliwe jest wykorzystanie wartowników, co zostało pokazane w kodzie źródłowym 4. Pole `prev` wartownika umieszczonego na początku listy oraz pole `next` wartownika znajdującego się na końcu listy zazwyczaj wskazują odpowiednio na nich samych. Zastosowanie wartowników dla dwukierunkowej listy liniowej niesie ze sobą te same znamiona i korzyści, co w przypadku listy jednokierunkowej.

Kod źródłowy 3: Przykładowy program z dwukierunkową listą liniową

```
#include <stdio.h>
#include <stdlib.h>

struct dll_node
{
    int data;
    struct dll_node *prev, *next;
};

struct dll_node *create_list(int data)
{
    struct dll_node *front = (struct dll_node *)
        malloc(sizeof(struct dll_node));
    if (NULL != front)
    {
        front->data = data;
        front->prev = front->next = NULL;
    }
    return front;
}

struct dll_node *insert_front(struct dll_node *front,
                              struct dll_node *new_node)
{
    new_node->next = front;
    front->prev = new_node;
    return new_node;
}

struct dll_node *find_spot(struct dll_node *front, int data)
{
    struct dll_node *prev = NULL;
    while ((NULL != front) && (front->data > data))
    {
        prev = front;
        front = front->next;
    }
```

```
38      return prev;
39  }
40
41  void insert_after(struct dll_node *node, struct dll_node *new_node)
42  {
43      new_node->prev = node;
44      new_node->next = node->next;
45      node->next->prev = new_node;
46      node->next = new_node;
47  }
48
49  void insert_back(struct dll_node *back, struct dll_node *new_node)
50  {
51      back->next = new_node;
52      new_node->prev = back;
53  }
54
55  struct dll_node *insert_node(struct dll_node *front, int data)
56  {
57      if (NULL == front)
58          return NULL;
59
60      struct dll_node *new_node = (struct dll_node *)
61          malloc(sizeof(struct dll_node));
62      if (NULL != new_node)
63      {
64          new_node->data = data;
65          new_node->prev = new_node->next = NULL;
66          if (front->data <= data)
67              return insert_front(front, new_node);
68          else
69          {
70              struct dll_node *node = find_spot(front, data);
71              if (NULL != node->next)
72                  insert_after(node, new_node);
73              else
74                  insert_back(node, new_node);
75          }
76      }
77      return front;
78  }
79
80  struct dll_node *delete_front(struct dll_node *front)
81  {
82      struct dll_node *next = front->next;
83      if (NULL != next)
84          next->prev = NULL;
85      free(front);
86      return next;
87  }
88
89  struct dll_node *find_node(struct dll_node *front, int data)
90  {
91      while ((NULL != front) && (front->data != data))
92          front = front->next;
93      return front;
94  }
95
```

```
96  void delete_within(struct dll_node *node)
97  {
98      node->next->prev = node->prev;
99      node->prev->next = node->next;
100     free(node);
101 }
102
103 void delete_back(struct dll_node *back)
104 {
105     back->prev->next = NULL;
106     free(back);
107 }
108
109 struct dll_node *delete_node(struct dll_node *front, int data)
110 {
111     if (NULL == front)
112         return NULL;
113
114     if (front->data == data)
115         return delete_front(front);
116
117     struct dll_node *node = find_node(front, data);
118     if (NULL != node)
119     {
120         if (NULL != node->next)
121             delete_within(node);
122         else
123             delete_back(node);
124     }
125     return front;
126 }
127
128 void print_list(struct dll_node *front)
129 {
130     for (; NULL != front; front = front->next)
131         printf("%d ", front->data);
132     printf("\n");
133 }
134
135 void print_list_backwards(struct dll_node *front)
136 {
137     if (NULL != front)
138     {
139         for (; NULL != front->next; front = front->next);
140         for (; NULL != front; front = front->prev)
141             printf("%d ", front->data);
142     }
143     printf("\n");
144 }
145
146 void remove_list(struct dll_node **front)
147 {
148     while (NULL != *front)
149     {
150         struct dll_node *next = (*front)->next;
151         free(*front);
152         *front = next;
153     }
```

```
}

int main()
{
    struct dll_node *front = create_list(1);
    int i;

    for (i=2; i<10; i++)
        front = insert_node(front, i);
    printf("List elements:\n");
    print_list(front);
    printf("List elements printed backwards:\n");
    print_list_backwards(front);

    front = insert_node(front, 0);
    printf("List elements after insertion of 0:\n");
    print_list(front);
    front = insert_node(front, 5);
    printf("List elements after insertion of 5:\n");
    print_list(front);
    front = insert_node(front, 10);
    printf("List elements after insertion of 10:\n");
    print_list(front);

    front = delete_node(front, 0);
    printf("List elements after deletion of 0:\n");
    print_list(front);
    front = delete_node(front, 1);
    printf("List elements after deletion of 1:\n");
    print_list(front);
    front = delete_node(front, 1);
    printf("List elements after deletion of 1:\n");
    print_list(front);
    front = delete_node(front, 5);
    printf("List elements after deletion of 5:\n");
    print_list(front);
    front = delete_node(front, 10);
    printf("List elements after deletion of 10:\n");
    print_list(front);

    remove_list(&front);
    return 0;
}
```

Kod źródłowy 4: Szablon programu dwukierunkowej listy liniowej z wartownikami

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct dll_node
{
    int data;
    struct dll_node *prev, *next;
};

struct dll_with_guards
```

```
12 {
13     struct dll_node *front, *back;
14 };
15
16 bool create_list(struct dll_with_guards *dll)
17 {
18     struct dll_node *front = (struct dll_node *)
19         malloc(sizeof(struct dll_node));
20     struct dll_node *back = (struct dll_node *)
21         malloc(sizeof(struct dll_node));
22     if ((NULL != front) && (NULL != back))
23     {
24         dll->front = front->prev = back->prev = front;
25         dll->back = front->next = back->next = back;
26         return true;
27     }
28     free(front);
29     free(back);
30     return false;
31 }
32
33 void remove_list(struct dll_with_guards *dll)
34 {
35     while (dll->front != dll->back)
36     {
37         struct dll_node *next = dll->front->next;
38         free(dll->front);
39         dll->front = next;
40     }
41     free(dll->back);
42     dll->front = dll->back = NULL;
43 }
44
45 int main()
46 {
47     struct dll_with_guards dll = {NULL, NULL};
48
49     if (create_list(&dll))
50     {
51         // Wykonaj jakieś operacje na liście.
52         remove_list(&dll);
53     }
54     return 0;
55 }
```

4 Zadania

Uwaga! Programy należy napisać z podziałem na funkcje z parametrami oraz nie można w nich korzystać ze zmiennych globalnych.

1. Zmodyfikuj przykładowy program z instrukcji z listą jednokierunkową, tak aby do funkcji `insert_node()` i `delete_node()` początek listy był przekazywany przez parametr będący podwójnym wskaźnikiem, oraz żeby zwracały one wartość typu `bool`

sygnalizującą poprawność wykonanej operacji. Uwaga! Zmiany w ww. funkcjach pociągną za sobą modyfikacje innych części programu.

2. W przykładowym programie z listą jednokierunkową występują dwie funkcje o bardzo podobnym kodzie: `find_spot()` i `find_prev_node()`. Różnią się one jedynie warunkiem sprawdzanym w pętli `while`. Można je połączyć w jedną, wykonując sprawdzenie odpowiedniego warunku w odrębnej funkcji, przekazanej do niej przez parametr będący wskaźnikiem na funkcję. Zaimplementuj to rozwiązanie.

3. Napisz program, w którym odwrócisz kolejność elementów w jednokierunkowej liście liniowej. Znajdź jak najprostsze rozwiązanie. Uwaga! W tym zadaniu nie chodzi o to, aby wypisać wartości elementów listy w odwrotnej kolejności, ale aby zmienić „kierunek" pól wskaźnikowych `next` poszczególnych elementów listy.

4. Napisz program, który sprawdzi, czy litery umieszczone w elementach jednokierunkowej listy liniowej tworzą palindrom.

5. Napisz program, w którym zaimplementujesz jednokierunkową listę liniową z użyciem dwóch wartowników. Jeden z nich powinien znajdować się na początku listy, a drugi na końcu. Oprócz standardowych operacji wspieranych przez listę dodaj nową funkcję o nazwie `clear_list()`, która będzie usuwać wszystkie elementy listy za wyjątkiem wartowników (opróżniać listę).

6. Zmodyfikuj przykładowy program z instrukcji z listą dwukierunkową, tak aby do funkcji `insert_node()` i `delete_node()` początek listy był przekazywany przez parametr będący podwójnym wskaźnikiem, oraz żeby zwracały one wartość typu `bool` sygnalizującą poprawność wykonanej operacji. Uwaga! Zmiany w ww. funkcjach pociągną za sobą modyfikacje innych części programu.

7. Napisz program, który zapisze w dwukierunkowej liście liniowej wyrazy podane przez użytkownika i wypisze je w takiej kolejności, w jakiej on je wprowadził, oraz w kolejności odwrotnej. Wprowadzanie wyrazów należy zakończyć po tym, jak użytkownik poda wyraz „koniec". Przyjmij, że wszystkie wyrazy podane przez użytkownika będą zawierały nie więcej niż 50 znaków.

8. Napisz program, który sprawdzi, czy litery umieszczone w elementach dwukierunkowej listy liniowej tworzą palindrom.

9. Napisz program, w którym zaimplementujesz dwukierunkową listę liniową z użyciem dwóch wartowników. Jeden z nich powinien znajdować się na początku listy, a drugi na końcu. Oprócz standardowych operacji wspieranych przez listę dodaj nową funkcję o nazwie `clear_list()`, która będzie usuwać wszystkie elementy listy za wyjątkiem wartowników (opróżniać listę).

10. Napisz program, w którym operacje na dwukierunkowej liście liniowej będą zaimplementowane w postaci rekurencyjnej.