Structuri de Date şi Algoritmi
Laborator III

Adrian Liţă

2015

1 Scopul laboratorului

Laboratorul III al materiei Structuri de Date şi Algoritmi are ca scop introducerea structurii de date **listă dublu înlănţuită**, sau mai scurt, listă dublă.
În acest laborator se parcurg următoarele puncte:

- structura de date a unei liste dublu înlănţuită
- modul de funcţionare a unei liste dublu înlănţuită
- parcurgerea listei dublu înlănţuită
- adăugarea unui element în lista dublu înlănţuită
- ştergerea unui element din lista dublu înlănţuită

2 Desfăşurarea lucrării

Comparând lista dublu înlănţuită cu lista simplu înlănţuită, prima diferă doar prin faptul că ţine minte, pe lângă adresa următorului element, şi adresa elementului precedent. Acest lucru oferă mai multe avantaje, dintre care putem evidenţia o flexibilitate sporită în parcurgere (putând fi făcută în ambele sensuri) şi uşurinţă în operaţii.

2.1 Structura de date şi modul de funcţionare a unei liste duble

```
typedef struct node
{
  [tip de date informatie]

  struct node *prev;
  struct node *next;
} NODE, *PNODE;
```

Transpus în plan vizual, codul de mai sus arată în felul următor:

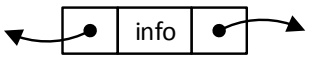

Figura 1: Structura de date a listei dublu înlănţuită

La fel ca şi în lista simplu înlănţuită, informaţia din interiorul structurii poate fi orice, inclusiv o altă structură. Poate, de asemenea, să nu fie doar o informaţie, ci mai multe (spre exemplu, toate datele necesare despre o persoană). Un exemplu concret de listă dublu înlănţuită este prezentat mai jos, şi reprezintă o listă de numere întregi.

```
typedef struct node
{
  int info;
  struct node *prev;
  struct node *next;
} NODE, *PNODE;
```

În figura 1 este prezentată structura de date a unei liste dublu înlănţuite. La fel ca în lista simplu înlânţuită, această unitate este un element al listei.

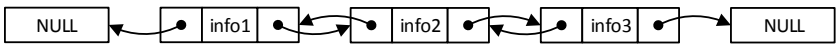

Figura 2: Listă dublu înlănţuită

În figura 2 este prezentată o listă dublu înlănţuită cu 3 elemente. Pentru a ţine minte lista este recomandat să ţinem minte **primul** element al listei, deoarece prin parcurgerea listei (capitolul următor) se poate ajunge la oricare din elementele listei. Mai mult, în lista dublu înlănţuită faţă de lista simplu înlănţuită, plecând de la **oricare** element al listei se poate ajunge oricare alt element (inclusiv primul). Similar cu lista simplă, Vom ţine minte primul element al listei printr-un pointer ***head**. Opţional se poate ţine minte şi ultimul element al listei (***end**). Un exemplu de listă completă este prezentat în figura 3 şi în codul următor:

```
typedef struct node
{
  int info;
  struct prev *next;
  struct node *next;
} NODE, *PNODE;

int main()
{
  PNODE head = NULL;
  PNODE end = NULL;
```

```
12
13   return 0;
14 }
```

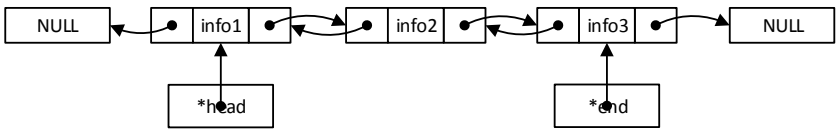

Figura 3: Listă dublu înlănţuită cu început şi sfârşit

2.2 Parcurgerea unei liste dublu înlănţuită

Parcugerea unei liste dublu înlănţuită se poate face în două feluri:

- prin pointerul ***next** (similar cu lista simplă)
- prin pointerul ***prev**

Similar cu parcurgerea unei liste simple, în programul următor sunt exemplificate cele două tipuri de parcurgere în lista dublu înlănţuită.

```
1  typedef struct node
2  {
3    int info;
4    struct node *prev;
5    struct node *next;
6  } NODE, *PNODE;
7
8  int main()
9  {
10   PNODE head;
11   PNODE end;
12   //presupunem ca lista are elemente in ea
13   //si head este pointer catre primul element
14   //si end este pointer catre ultimul element
15
16   PNODE temp;
17
18   //parcurgerea prin next (de la stanga la dreapta)
19   temp = head;  //temp copiaza adresa primului element
20   while(temp != NULL)
21   {
22     //informatia utila este temp->info
23     temp = temp->next;  //trecem la elementul urmator
24   }
25
26
27   //parcurgerea prin prev (de la dreapta la stanga)
28   temp = head;  //temp copiaza adresa ultimului element
29   while(temp != NULL)
30   {
```

```
    //informatia utila este temp->info
    temp = temp->prev;  //trecem la elementul anterior
  }

  return 0;
}
```

Un bun exemplu de parcurgere îl reprezintă funcţiile de afişare a listei. Pentru lista dublu înlănţuită putem face afişarea fie în ordine directă (prin pointerul **next**, de la stânga la dreapta - funcţia **show()**), sau în ordine inversă (prin pointerul ***prev** de la dreapta la stânga - funcţia **show_rev()**).

```
typedef struct node
{
  int info;
  struct node *prev;
  struct node *next;
} NODE, *PNODE;

void show(PNODE headlist)
{
  PNODE temphead = headlist;
  while(temphead != NULL)
  {
    printf("%d ", temphead->info);
    temphead = temphead->next;
  }
  printf("\n");
}

void show_rev(PNODE endlist)
{
  PNODE temphead = endlist;
  while(temphead != NULL)
  {
    printf("%d ", temphead->info);
    temphead = temphead->prev;
  }
  printf("\n");
}

int main()
{
  PNODE head;
  PNODE end;
  //presupunem ca lista are elemente in ea
  //si head este pointer catre primul element
  //si end este pointer catre ultimul element

  show(head); //afiseaza lista in ordine stanga-dreapta
  show_rev(end);  //afiseaza lista in ordine dreapta-stanga

  return 0;
}
```

2.3 Adăugarea unui element în lista dublu înlănţuită

Iniţial plecăm de la o listă goală, adică ambii pointeri ***head** şi ***end** sunt **NULL** (figura 4).

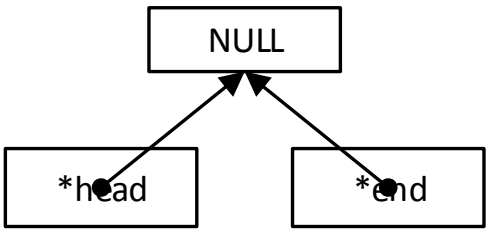

Figura 4: Lista goală

Indiferent de destinaţia unde trebuie adăugat noul element în listă (început, interior, sfărşit), paşii care trebuiesc urmaţi sunt următorii:

1. Alocarea de memorie pentru un element nou al listei
2. Completarea câmpului/câmpurilor de informaţie dorit
3. Parcurgerea listei până la poziţia pe care vrem să inserăm elementul nou creat (cu excepţia cazului în care elementul trebuie introdus la început sau la sfârşit şi avem pointeri către acele elemente)
4. Setarea pointerului ***next** al elementului nou adăugat corespunzător:
   - adresa următorului element din listă (când acesta există)
   - **NULL** dacă elementul este adăugat la sfârşitul listei (include cazul când elementul este primul adăugat într-o listă goală
5. Setarea pointerului ***prev** al elementului nou adăugat corespunzător:
   - adresa elementului precedent în listă (când acesta există)
   - **NULL** dacă elementul este adăugat la începutul listei (include cazul când elementul este primul adăugat într-o listă goală)
6. Setarea pointerului ***next** al elementului deja existent care-l precede pe cel nou adăugat către adresa elementului nou adăugat (cu excepţia cazului când elementul nou adăugat este primul)
7. Setarea pointerului ***prev** al elementului deja existent care-l succede pe cel nou adăugat către adresa elementului nou adăugat (cu excepţia cazului când elementul nou adăugat este ultimul)

8. actualizarea pointerului ***head** (dacă s-a modificat primul element)
9. actualizarea pointerului ***end** (dacă acesta există şi are nevoie de modificare)

În continuare sunt exemplificate prin figuri cazurile de adăugare când sunt disponibili ambii pointeri: ***head** şi ***end**.

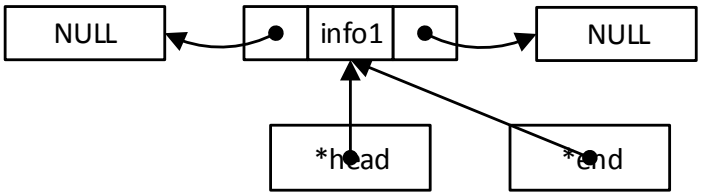

Figura 5: Adăugarea primului element din lista dublu înlănţuită

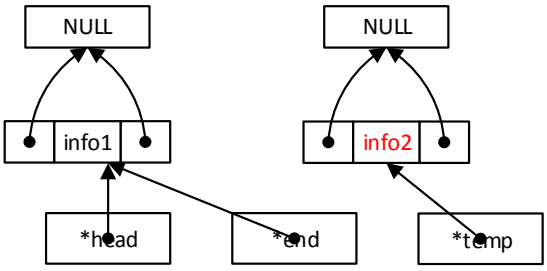

Figura 6: Alocarea de memorie şi completarea câmpului de informaţie pentru un element nou intordus pe poziţia 2

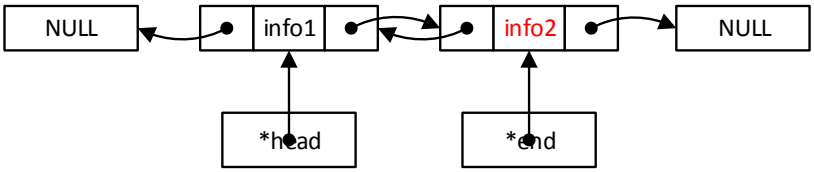

Figura 7: Legarea câmpurilor ***next**, ***prev** şi actualizarea pointerilor ***head** şi ***end**

```
PNODE addLast(PNODE head, int info_new)
{
  PNODE temp = NULL;
  temp = (PNODE)malloc(sizeof(NODE)*1);
  if(temp == NULL)
  {
    printf("Eroare la alocare!\n");
    exit(0);  //nu are sens sa continui
  }
  temp->info = info_new;
  temp->next = NULL;
  temp->prev = NULL;

  if(head == NULL)  //daca lista era goala
    return temp;  //va fi primul element

  //parcurgem pana la ultimul element al listei
  PNODE i = head;
  while(i->next != NULL)
    i = i->next;

  i->next = temp;
  temp->prev = i;
  return head;
}

PNODE addAfter(PNODE head, int info_search, int info_new)
{
  PNODE temp = NULL;
  temp = (PNODE)malloc(sizeof(NODE)*1);
  if(temp == NULL)
  {
    printf("Eroare la alocare!\n");
    exit(0);  //nu are sens sa continui
  }
  temp->info = info_new;
  temp->next = NULL;
  temp->prev = NULL;

  if(head == NULL)  //daca lista era goala
    return temp;  //va fi primul element

  //parcurgem pana gasim elementul cautat
  //daca nu exista, parcurgem pana la ulimul element al listei
  //si adaugam elementul nou acolo

  PNODE i = head;
  while( (i->next != NULL) && (i->info != info_search) )
    i = i->next;

  i->next->prev = temp; //elementul precedent elementului urmator
  temp->next = i->next; //adaugam elementul temp intre
              //elem i si elem i->next
  temp->prev = i;      //elementul precedent elementului nou
  i->next = temp;
  return head;
}
```

```
58
59 int main()
60 {
61   PNODE head = NULL;
62
63   head = addLast(head, 1);
64   addLast(head, 2);
65   addLast(head, 3);
66   addLast(head, 4);
67   addLast(head, 5);
68
69   //in momentul acesta lista arata asa
70   //NULL<-1<->2<->3<->4<->5->NULL
71
72   head = addAfter(head, 1, 10);
73   addAfter(head, 2, 20);
74   addAfter(head, 3, 30);
75
76   //in momentul acesta lista arata asa
77   //NULL<-1<->10<->2<->2<->3<->30<->4<->5->NULL
78
79   return 0;
80 }
```

2.4 Ştergerea unui element dîn lista dublu înlănţuită

Ştergerea unui element din listă se face în următorii paşi:

1. Se verifică dacă nu cumva lista este goală.
2. Se parcurge până se găseşte elementul precedent elementului dorit pentru a fi şters (dacă este cazul).
3. Se izolează elementul care trebuie şters.
4. Se modifică pointerii ***next** şi ***prev** corespunzător.
5. Se modifică pointerii ***head** şi ***end** (dacă este cazul).

Spre exemplu, plecând de la lista din figura 3, ştergerea celui de-al doilea element se face în felul următor.

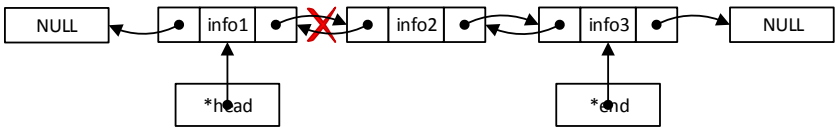

Figura 8: Primul pas în ştergerea celui de-al doilea element

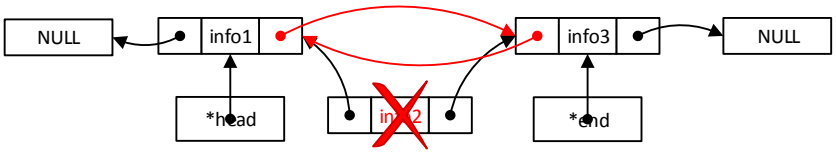

Figura 9: Ştergerea celui de-al doilea element

```
PNODE deleteNode(PNODE head, int info_search)
{
  //verificam daca nu cumva lista e goala
  if(head == NULL)
    return NULL;

  //testam cazul special in care
  //trebuie sters chiar elementul head
  if(head->info == info_search)
  {
    PNODE n = head->next;
    head->next->prev = NULL;
    free(head);
    return n;
  }

  PNODE i = head;
  while(i->next != NULL)
  {
    if(i->next->info == info_search)
    {
      PNODE n = i->next;
      n->next->prev = i;
      i->next = i->next->next;
      free(n);
      return head;
    }
    i = i->next;
  }

  return head;
}

int main()
{
  PNODE head;

  //presupunem ca lista arata asa
  //NULL<-1<->2<->3<->4<->5->NULL

  deleteNode(head, 3);
  //lista arata: NULL<-1<->2<->4<->5->NULL

  head = deleteNode(head,1);
  //lista arata asa: NULL<-2<->4<->5->NULL
```

```
46
47    return 0;
48 }
```

3 Probleme

1. Creaţi o funcţie care construieşte o listă dublu înlănţuită dintr-un vector dat.

2. Pornind de la o listă dublă în care nu ştim decât adresa unui element oarecare, să se găsească primul şi ultimul element.

3. Creaţi o funcţie de adăugare în lista dublă care să primească pe lângă parametrii standard încă un parametru care să specifice dacă elementul nou adăugat se va adăuga mai aproape de începutul sau de sfârşitul listei. Implementaţi adăugarea în mod eficient să dureze, în medie, mai puţin decât dacă nu ar exista acest parametru.

4. Implementaţi funcţiile de adăugare şi ştergere din lista dublă pe baza pointerului ***end**.

5. Implementaţi o listă **simplu** înlănţuită **ciclică**. Adaptaţi apoi codul pentru a funcţiona folosind o listă dublu înlănţuită.

6. Concatenaţi două liste dublu înlănţuite.

7. Plecând de la două liste dublu înlănţuite, căutaţi dacă prima listă se găseşte în a doua.