Structuri de Date şi Algoritmi
Laborator II

Adrian Liţă

2015

1 Scopul laboratorului

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

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

2 Desfăşurarea lucrării

Lista în programare încearcă să modeleze lista reală de date. Practic, o listă este compusă din unul sau mai multe elemente. Tipul acestor elemente variază cu tipul listei: o listă de cumpărături va conţine ca elemente obiecte ce trebuiesc cumpărate, o listă de persoane va conţine ca elemente persoane, etc.

Ideea de la care pleacă o listă este aceea că lista va avea un număr variabil de elemente, şi operaţiile uzuale şi des întâlnite într-o listă vor fi cele de adăugare (oriunde în listă, nu neapărat la sfârşit), şi de ştergere din listă (bineînţeles, oricare element).

2.1 Structura de date şi modul de funcţionare a unei liste

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

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

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

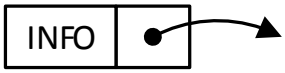

Figura 1: Structura de date a listei simplu înlănţuită

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 simplu înlănţuită este prezentat mai jos, şi reprezintă o listă de numere întregi.

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

În figura 1 este prezentată structura de date a unei liste simplu înlănţuite. Această unitate reprezintă, de fapt, un element al listei, şi în programare se mai numeşte şi **nod**.
Elementul listei este o structură **recurentă** în care se găseşte informaţia necesară şi un pointer către elementul următor din listă. Ultimul element al listei va conţine în câmpul **next** elementul **NULL**. O listă este compusă din mai multe astfel de elemente, legate între ele prin pointer-ul **next**.

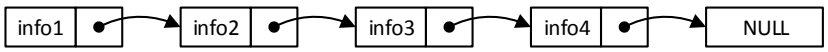

Figura 2: Listă simplu înlănţuită

În figura 2 este prezentată o listă simplu înlănţuită cu 4 elemente. Pentru a ţine minte lista este nevoie să ţinem minte doar **primul** element al listei, deoarece prin parcurgerea listei (capitolul următor) se poate ajunge la oricare din elementele listei. Practic, lista din figura 2 este completă şi funcţională dacă mai avem încă un pointer în care să ţinem minte primul element al listei. Vom numi acest 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 node *next;
} NODE, *PNODE;

int main()
```

```
{
  PNODE head = NULL;
  PNODE end = NULL;

  return 0;
}
```

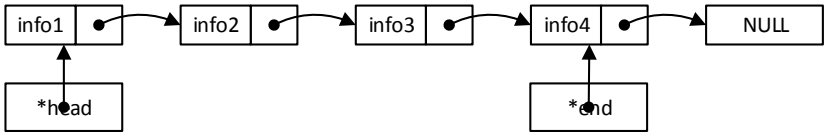

Figura 3: Listă simplu înlănţuită cu început şi sfârşit

2.2 Parcurgerea unei liste simplu înlănţuită

Parcurgerea unei liste simplu înlănţuită se face element cu element, pornind de la **head** şi sfârşind fie la **NULL** (pentru parcurgerea întregii liste) sau la un anumit element (spre exemplu în cazul căutarii în listă).

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

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

  PNODE temp;
  temp = head;  //temp copiaza adresa primului element
  while(temp != NULL)
  {
    //informatia utila este temp->info
    temp = temp->next;  //trecem la elementul urmator
  }

  return 0;
}
```

În exemplul de mai sus, se parcurge lista ce are ca prim element pe **head** (parcurgerea se realizază între liniile 14 şi 19). Deoarece nu avem voie să modificăm valoarea lui **head** (fiindcă apoi nu vom mai şti unde în memorie se află primul element, deci rezultă că pierdem memorie), se declară încă un pointer la listă temporar cu care vom face parcurgerea. În exemplul de mai sus parcurgem absolut toată lista, de la primul la ultimul element, urmând ca la sfârşitul parcurgerii **temp** să fie **NULL**. Parcurgerea în sine presupune ca pointerul **temp**

să fie pe rând, pointer la fiecare element din listă.

În exemplul următor vom folosi aceeaşi parcurgere pentru o exemplifica mult mai concret: afişăm toate elementele din listă.

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

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

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

  show(head); //afiseaza lista

  return 0;
}
```

Funcţia **show(PNODE)** primeşte ca parametru adresa elementului de unde începe afişarea (în majoritatea cazurilor acesta va fi primul element din listă). Funcţia copiază această adresă într-un pointer temporar, şi cât timp nu ajunge la ultimul element, afişează informaţia din element, după care trece la elementul următor (**temphead = temphead->next**). La sfârşit, funcţia trece la rândul următor.

2.3 Adăugarea unui element în listă

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

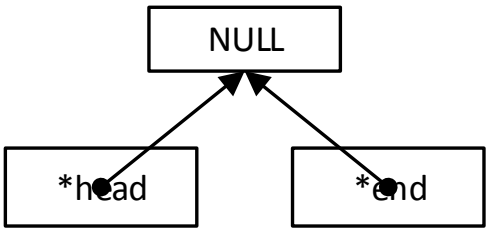

Figura 4: Lista goală

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 a cazului special din capitolul 2.3.1)
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 ***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)
6. actualizarea pointerului ***head** (dacă s-a modificat primul element)
7. 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**.

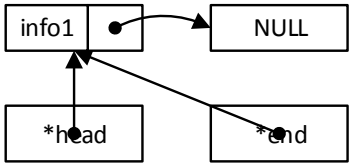

Figura 5: Adăugarea primului element din listă

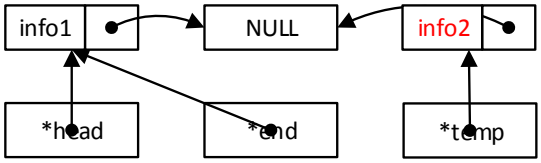

Figura 6: Alocarea de memorie şi completarea câmpului de informaţie pentru un element nou intordus pe poziţia 2

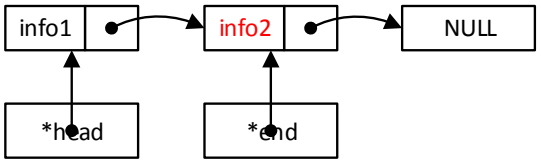

Figura 7: Legarea câmpurilor ***next** şi actualizarea pointerilor ***head** şi ***end**

```
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;

  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)
```

```
19      i = i->next;
20
21    i->next = temp;
22    return head;
23  }
24
25  PNODE addAfter(PNODE head, int info_search, int info_new)
26  {
27    PNODE temp = NULL;
28    temp = (PNODE)malloc(sizeof(NODE)*1);
29    if(temp == NULL)
30    {
31      printf("Eroare la alocare!\n");
32      exit(0);  //nu are sens sa continui
33    }
34    temp->info = info_new;
35    temp->next = NULL;
36
37    if(head == NULL)  //daca lista era goala
38      return temp;  //va fi primul element
39
40    //parcurgem pana gasim elementul cautat
41    //daca nu exista, parcurgem pana la ulimul element al listei
42    //si adaugam elementul nou acolo
43
44    PNODE i = head;
45    while( (i->next != NULL) && (i->info != info_search) )
46      i = i->next;
47
48    temp->next = i->next; //adaugam elementul temp intre
49                  //elem i si elem i->next
50    i->next = temp;
51    return head;
52  }
53
54  int main()
55  {
56    PNODE head = NULL;
57
58    head = addLast(head, 1);
59    addLast(head, 2);
60    addLast(head, 3);
61    addLast(head, 4);
62    addLast(head, 5);
63
64    //in momentul acesta lista arata asa
65    //1->2->3->4->5->NULL
66
67    head = addAfter(head, 1, 10);
68    addAfter(head, 2, 20);
69    addAfter(head, 3, 30);
70
71    //in momentul acesta lista arata asa
72    //1->10->2->2->3->30->4->5->NULL
73
74    return 0;
75  }
```

2.3.1 Caz special: când adăugarea se face la sfârşitul listei şi există pointerul *end

În cazul în care există şi pointerul ***end** (pointer către ultimul element din listă) adăugarea unui element la sfârşitul listei se simplifică considerabil (mai ales ca timp de procesare de către program) deoarece nu mai trebuie parcursă lista. Paşii pentru adăugarea elementului sunt exact paşii standard, cu excepţia parcurgerii listei. De asemenea, dacă lista nu era goală, pointerul ***head** nu trebuie modificat. Timpul de adăugare, în acest caz este constant indiferent de numărul de elemente din listă şi este minim.

```
PNODE addLast(PNODE head, PNODE *end, 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;

  //daca nu exista alt nod deocamdata
  if(head == NULL)
  {
      (*end) = temp;
      return temp;
  }

  //altfel
  (*end)->next = temp;
  (*end) = temp;

  return head;
}



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

  head = addLast(head, &end, 1);
  addLast(head, &end, 2);
  addLast(head, &end, 3);
  addLast(head, &end, 4);
  addLast(head, &end, 5);

  //in momentul acesta lista arata asa
  //1->2->3->4->5->NULL

  return 0;
}
```

2.4 Ştergerea unui element în listă

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** 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.

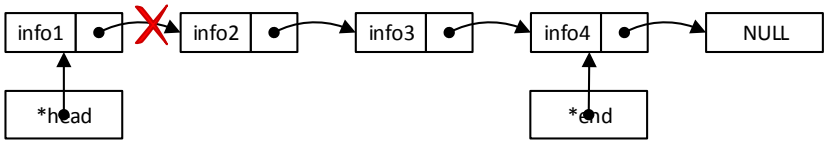

Figura 8: Primul pas în ştergerea celui de-al doilea element

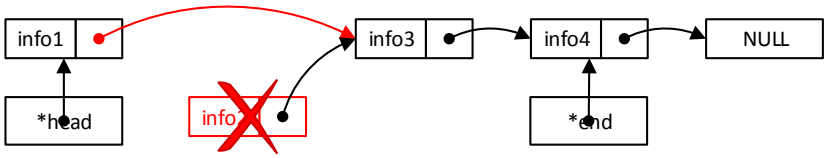

Figura 9: Ştergerea celui de-al doilea element

```
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;
    free(head);
    return n;
```

```
14    }
15
16    PNODE i = head;
17    while(i->next != NULL)
18    {
19      if(i->next->info == info_search)
20      {
21        PNODE n = i->next;
22        i->next = i->next->next;
23        free(n);
24        return head;
25      }
26      i = i->next;
27    }
28
29    return head;
30  }
31
32  int main()
33  {
34    PNODE head;
35
36    //presupunem ca lista arata asa
37    //1->2->3->4->5->NULL
38
39    deleteNode(head, 3);
40    //lista arata: 1->2->4->5->NULL
41
42    head = deleteNode(head,1);
43    //lista arata asa: 2->4->5->NULL
44
45    return 0;
46  }
```

2.5 Comparaţie între liste şi vectori

Diferenţele între liste şi vectori stau în performanţa de a executa anumite operaţii cu aceştia. Principalele diferenţe sunt:

- Este mult mai uşor de memorat date de dimensiuni diferite în liste. Un vector presupune ca fiecare element al său să fie de exact aceeaşi dimensiune.
- Dimensiunea unei liste poate creşte sau descreşte natural. Dimensiunea unui vector trebuie ştiută dinainte, sau vectorul trebuie re-creat în momentul în care este nevoie de expansiune.
- Amestecarea (sau schimbul) între elemente într-o listă se rezumă doar la a modifica pointerul ***next** al diferitelor elemente. Într-un vector este mai complicat şi operaţiile au nevoie de mai multă memorie.
- Timpul de acces în vector este mult mai mic decât într-o listă (unde trebuie făcută parcurgerea) dacă cunoaştem unde este localizat elementul dorit. Dacă elementul trebuie căutat, timpii de acces sunt similari.

3 Probleme

1. Afişaţi o listă începând cu al doilea element.
2. Creeaţi funcţia de adăugare a unui element înaintea unui element specificat.
3. Pornind de la o listă de **int** creeaţi o nouă listă care să conţină toate elementele primei liste în ordine inversă.
4. Într-o listă simplu înlănţuită de nume de persoane (deja populată), căutaţi o anumită persoană.
5. Creeaţi o listă de numere întregi şi populaţi-o cu numere între 1 şi 100. Ştergeţi apoi din listă toate numerele care nu sunt prime.
6. Realizaţi un program care realizează decimarea unei liste de întregi.