Structuri de date

Laboratorul 4

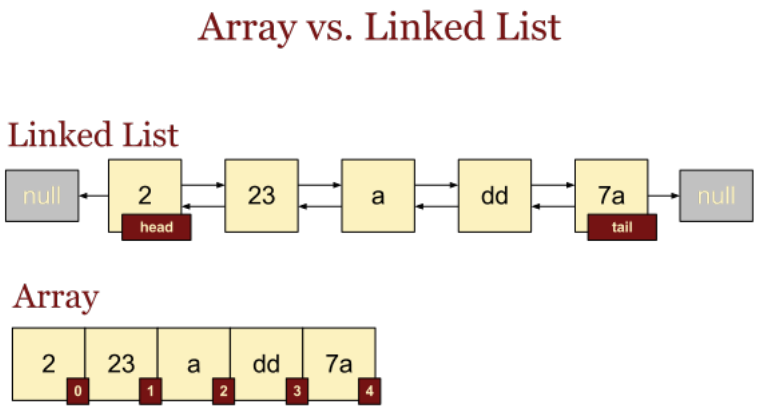
Mihai Nan
mihai.nan.cti@gmail.com
Grupa 314CC

Facultatea de Automatica şi Calculatoare
Universitatea Politehnica din Bucureşti
Anul universitar 2017 - 2018

1 Liste dublu inlantuite

1.1 Introducere

Intr-o ***lista liniara dublu inlantuita***, fiecare element contine doua adrese de legatura: una catre elementul urmator si alta catre elementul precedent. Aceasta structura permite accesul mai rapid la elementul precedent celui curent (necesar la eliminarea din lista) si parcurgerea listei in ambele sensuri (inclusiv existenta unui iterator in sens invers).



1.2 Implementarea structurii de date

Observatie

Pentru simplitate si omogenitate, exercitiile din cadrul acestui laborator se vor efectua uzitand reprezentarea, definita in blocul de cod de mai jos, pentru o lista liniara dublu inlantuita alocata dinamic.

Cod sursa C

```c
typedef struct list {
  int value;
  struct list* next;
  struct list* prev;
}*List;
```

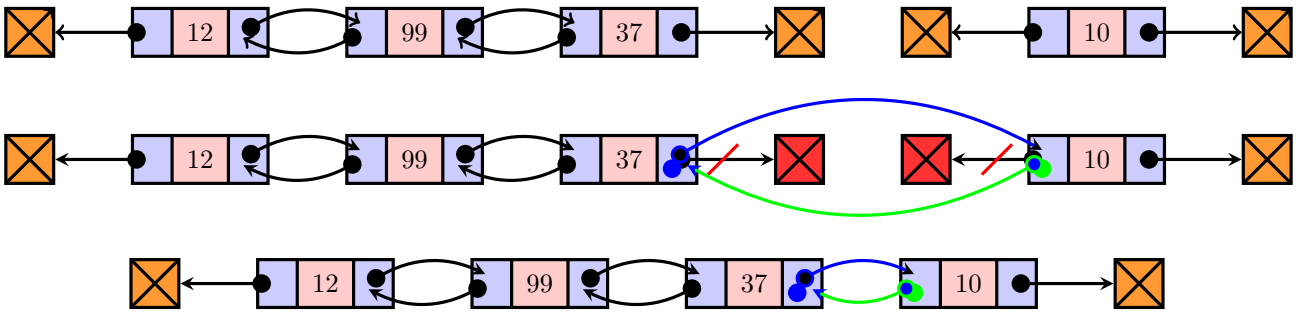
1.3 Operatii cu liste

1.3.1 Initializarea listei

Operatia de initializare a unei liste stabileste adresa de inceput a listei, cu alte cuvinte, construieste o lista cu un singur element. Crearea unui nou element de lista necesita alocarea de memorie, prin functia ***malloc***. Verificarea rezultatului cererii de alocare (***NULL*** - daca alocarea este imposibila) se poate face printr-o instructiune ***if***. Dupa ce i s-a alocat memorie, se retine valoarea si se stabilesc adresele elementelor urmator si anterior ca ***NULL*** – pentru a marca sfarsitul listei si inceputul ei.

1.3.2 Adaugarea la sfarsitul listei

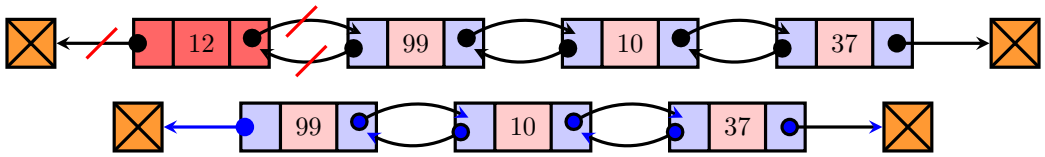
Avem de analizat cele doua cazuri: lista vida sau lista care contine elemente. Daca lista nu este vida, adaugam nodul la sfarsitul listei. Alocam memorie pentru noul nod si scriem informatia pe care dorim sa o adaugam in nod tot cu ajutorul functiei pentru initializare. Daca lista nu este vida, trebuie sa legam ultimul nod al listei de noul nod, devenind ultimul nod al listei, iar, in caz contrar, doar sa returnam noul nod initializat. Pentru a putea parcurge lista, trebuie sa folosim o varibaila auxiliara ***temp*** tot de tip ***List***, precum si o structura repetitiva – ***while***.



1.3.3 Stergerea unui element

Ca si adaugarea, stergerea unui element dintr-o lista este de mai multe feluri. Prin operatia de stergere se intelege scoaterea unui element din inlantuire. Elementul care a fost izolat de lista trebuie sa fie procesat in continuare, cel putin pentru a fi eliberata zona de memorie pe care o ocupa, de aceea, adresa lui trebuie salvata.

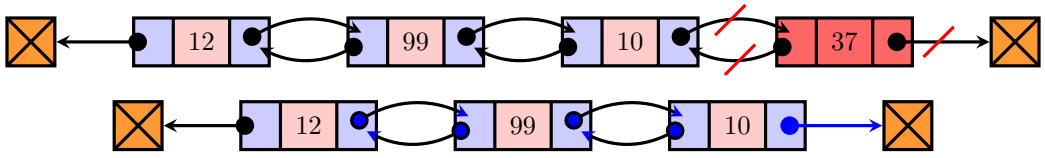
Stergerea de la inceputul listei



Sunt cateva cazuri pe care trebuie sa le luam in cosiderare: lista este vida sau lista are un singur element. Oricum ar fi, algoritmul general este urmatorul: salvez primul nod intr-un nod auxiliar ***temp***, succesorul primului nod (*al doilea nod din lista initiala*) devine primul nod (setez campul pentru elementul anterior ca fiind **NULL**) si eliberez memoria ocupata de obiectul salvat in ***temp***.

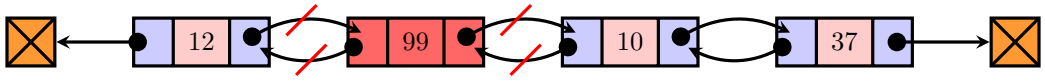
Stergerea de la sfarsitul listei

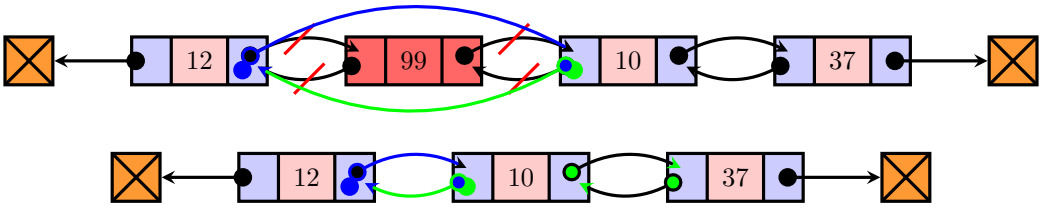
Pentru stergerea utlimului element din lista, trebuie sa cautam predecesorul sau – care va deveni ultimul nod din lista – sa eliberam memoria alocata pentru ultimul nod din lista si sa refacem legaturile.



Stergerea din interiorul listei

Operatia de stergere a unui nod plasat in interiorul listei este catalogata ca fiind o operatie mai grea, deoarece trebuie sa fie refacute 2 legaturi.




12
99
10
37
12
10
37

2 Probleme de laborator

Observatie

In arhiva laboratorului, gasiti un schelet de cod de la care puteti porni implementarea functiilor propuse, avand posibilitatea de a le testa functionalitatea.
Punctajul maxim pentru acest laborator este 10 si se va acorda doar daca rezolvarea este insotita si de explicatii.

2.1 Probleme standard

Problema 1 - 0.5 puncte

Pornind de la definitia structurii de date ***List*** care reprezinta un element al unei liste dublu inlantuite ce va retine valori de tip ***int***, implementati operatia de initializare a unei liste. Aceasta operatie stabileste adresa de inceput a listei, aloca memoria necesara pentru retinerea unui element, se adauga valoarea elementului ***value*** in lista si se stabileste adresa elementelor ***urmator*** si **precedent** ca fiind ***NULL*** - pentru a marca sfarsitul si inceputul listei.

```
List initList(int value);
```

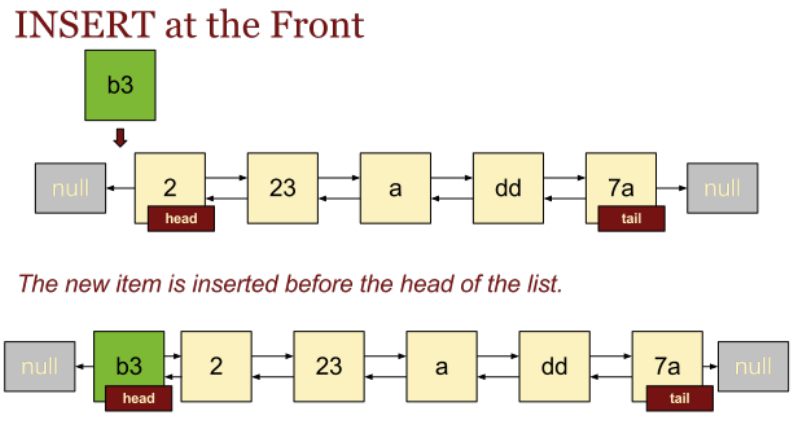
Observatie

In implementarea functiilor pentru rezolvarea urmatoarelor 3 probleme, analizati separat cazul listei vide!

Problema 2 - 1 punct

Implementati operatia de adaugare a unui element la inceputul listei liniare dublu inlantuite.

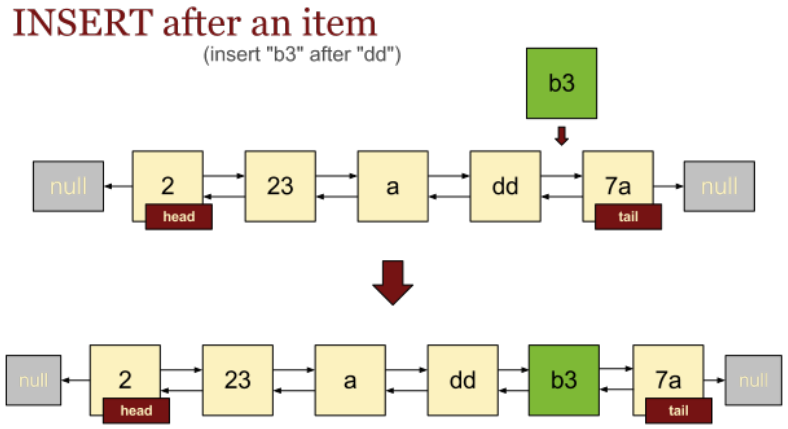
```
List addFirst(List l, int value);
```



Problema 3 - 1 punct

Implementati operatia de adaugare a unui nod intr-o lista dublu inlantuita, plasandu-l la finalul listei.

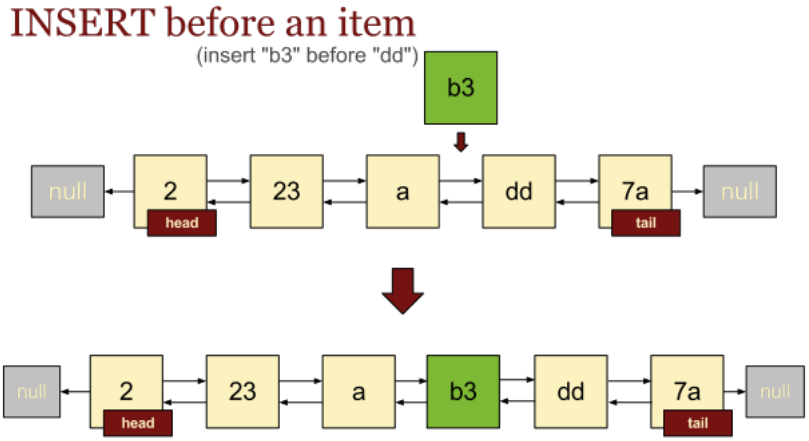
```
List addLast(List l, int value);
```



Problema 4 - **1 punct**

Implementati operatia de adaugare a unui nod intr-o lista dublu inlantuita, plasandu-l dupa un anumit element **x** al listei.

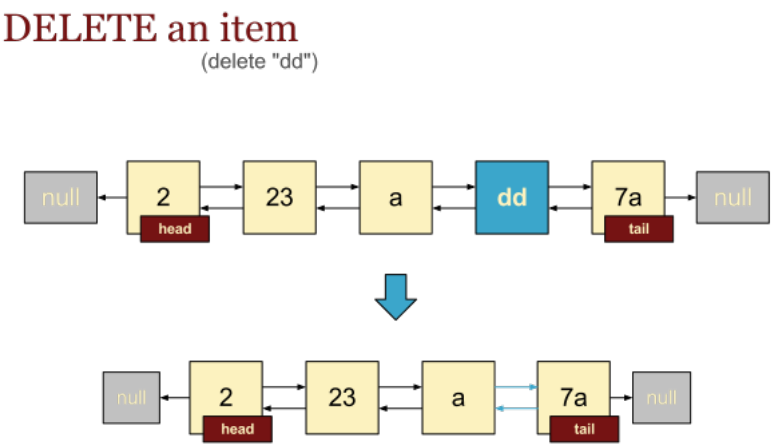
```
List addItem(List l, int x, int value);
```



Problema 5 - **2 puncte**

Sa se scrie o functie care poate fi uzitata pentru a sterge dintr-o lista liniara dublu inlantuita nodul care contine valoarea ***value***.

```
List deleteItem(List l, int value);
```



⚠ IMPORTANT !

⚠ Analizati toate cazurile posibile, raportandu-va la pozitia pe care o poate avea un nod intr-o lista si luand in considerare cazul listei vide!

Problema 6 - 1 punct

Definiti o functie care implementeaza operatia de stergere a listei. Cu alte cuvinte, functia va sterge fiecare nod din lista, dealocand memoria necesara reprezentarii sale, returnand lista vida.

Problema 7 - 1 punct

Scrieti o functie care inverseaza o lista dublu inlantuita, fara a folosi memorie auxiliara.

Algorithm 1 Reverse

```
1:  procedure REVERSE(list)
2:      temp ← NULL
3:      current ← list
4:      while current ≠ NULL do
5:          temp ← current → prev
6:          current → prev ← current → next
7:          current → next ← temp
8:          current ← current → prev
9:      if temp ≠ NULL then
10:         list ← temp → prev
        return list
```

Problema 8 - 2.5 puncte

Dându-se o listă dublu înlănuită cu elementele pozitive i sortate crescător, determinai toate perechile de celule a căror suma a valorilor este egală cu o valoare indicată (x).

```
void print_pairs(List lst, int x);
```

2.2 Probleme bonus

Problema 9 - 2 puncte

Sa se scrie doua functii care calculeaza suma si produsul a doua polinoame reprezentate prin doua liste dublu inlantuite. Sa se redefineasca tipul de date astfel incat sa contina doua valori: coeficientul si gradul unui termen.

```
List suma(List pol1, List pol2);
List produs(List pol1, List pol2);
```

3 Interviu

Observatie

Aceasta sectiune este una optionala si incearca sa va familiarizeze cu o serie de intrebari ce pot fi adresate in cadrul unui interviu tehnic. De asemenea, aceasta sectiune poate fi utila si in pregatirea pentru examenul final de la aceasta disciplina.

Intrebari interviu

1. Se ofera trei tipuri de liste: o lista simplu inlantuita, o lista dublu inlantuita si o lista ordonata. Pentru care din acest tip concatenarea a doua liste se realizeaza in ***O(1)***?
2. Cum putem verifica daca o lista dublu inlantuita este sau un o lista palindrom?
3. Se poate realiza operatia de accesare a unui nod dintr-o lista liniara dublu inlantuita in ***O(1)***?
4. Se ofera o lista liniara dublu inlantuita pentru care prima referinta pointeaza catre elementul urmator, iar a doua catre un nod ales aleator din lista. Se poate realiza operatia de clonare a unei astfel de liste fara a folosi memorie auxiliara? Argumentati!
5. Este mai eficienta operatia de reordonare in cazul unei liste ordonate decat in cazul unui vector sortat?
6. Oferiti cateva argumente pentru a demonstra ca operatia de adaugare a unui element la o lista ordonata este mai rapida si mai simpla decat la un vector ordonat.

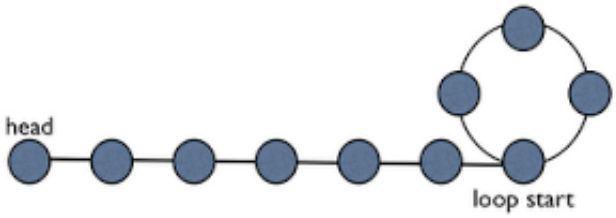
3.1 Probleme interviu

Problema 1: Se da un numar intreg N si o permutare a multimii 1, 2, ..., N. O subsecventa [i, j] (i $\leq$ j) contine toate elementele permutarii aflate intre pozitiile i si j inclusiv. Se numeste ***interval compact*** o subsecventa ale carei elemente formeaza o multime de valori consecutive, nu neaparat in ordine din permutare. De exemplu, pentru permutarea *1 2 6 4 5 3*, subsecventele *6, 4, 5* si *2 6 4 5 3* sunt intervale compacte, in timp ce subsecventele *1 2 6* si *2 6 4 5* nu sunt intervale compacte. Sa se determine numarul de intervale compacte din permutarea data.

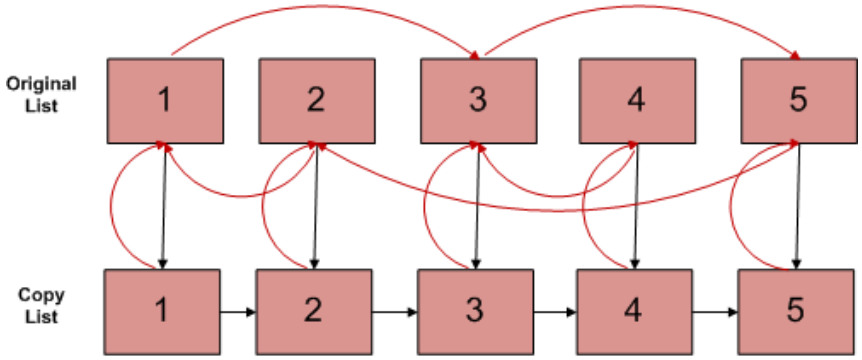
Sursa infoarena

Problema 2: Implementati o functie care primeste ca argument o lista liniara dublu inlantuita si determina daca lungimea acesteia este un numar par sau impar. Nu se va folosi un contor in implementare pentru calculul lungimii.

Problema 3: Danduse o lista liniara dublu inlantuita, sa se scrie o functie care determina daca aceasta contine sau nu un ciclu. In cazul in care lista contine un ciclu, se va afisa lungimea acestui ciclu.



Problema 4: Se ofera o lista dublu inlantuita care contine un pointer spre elementul urmator si unul care pointeaza catre un nod aleator, nu neaparat precedent. Scrieti un program care sa realizeze o copie a unei astfel de liste in ***O(n)***.



Problema 5: Sa se scrie un program care implementeaza algoritmul ***Strand Sort***, uzitand ca structura de date o lista dublu inlantuita.

Algorithm 2 StrandSort

```
1:  procedure StrandSort(globalList)
2:      subList ← NULL
3:      sortedList ← NULL
4:      while globalList ≠ NULL do
5:          subListHead ← NULL
6:          val ← topList(globalList)
7:          globalList ← globalList → next
8:          subListHead ← initList(val)
9:          subList ← subListHead
10:         if globalList ≠ NULL then
11:             sortedList ← mergeLists(subListHead, sortedList)
12:             continue
13:         tmp ← globalList
14:         while tmp ≠ NULL do
15:             if tmp → value ≥ val then
16:                 new ← newList(tmp → value)
17:                 subList → next ← new
18:                 subList ← new
19:                 val ← tmp → value
20:                 globalList ← deleteItem(globalList, tmp → value)
21:             tmp ← tmp → next
22:         sortedList ← mergeLists(sortedList, subListHead)
        return sortedList
```

Feedback

Pentru imbunatatirea constanta a acestui laborator, va rog sa completati formularul de feedback disponibil aici.

De asemenea, va rog sa semnalati orice greseala / neclaritate depistata in laborator pentru a o corecta.

Va multumesc anticipat!