

ALTE TIPURI DE LISTE

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare

Liste dublu înlănțuite

- Într-o **listă dublu înlănțuită**, fiecare element conține **două adrese de legătură**: una către elementul următor și alta către elementul precedent
- Această structură permite **accesul mai rapid** la elementul precedent celui curent (necesar la eliminare din listă) și parcurgerea listei în ambele sensuri (inclusiv existența unui iterator în sens invers pe listă)

Liste dublu înlănțuite

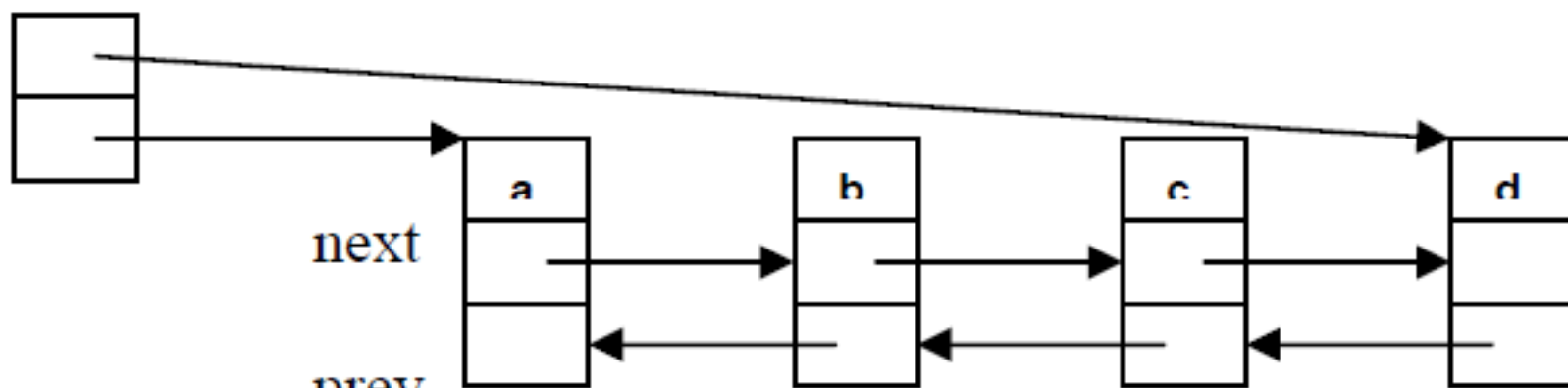
- Pentru acces rapid la ambele capete ale listei, se poate defini tipul *DList* ca o structură cu doi pointeri:
 - adresa primului element
 - adresa ultimului element
- Acest tip de listă este folosită pentru acces pe la ambele capete ale listei

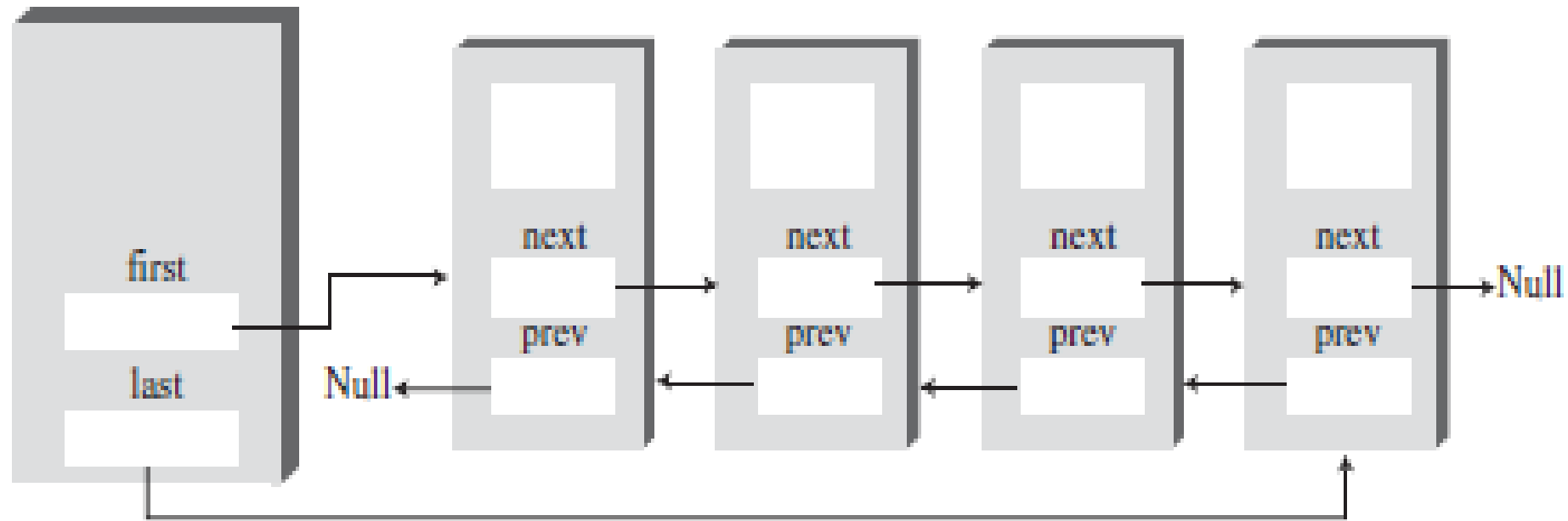
ultim

prim

next

prev





Exemplu de definire nod de listă dublu înlănțuită:

```
typedef struct nod {           // structură nod
    T val;                     // date
    struct nod * next;         // adresă nod următor
    struct nod * prev;         // adresă nod precedent
} Nod, * DList;
```



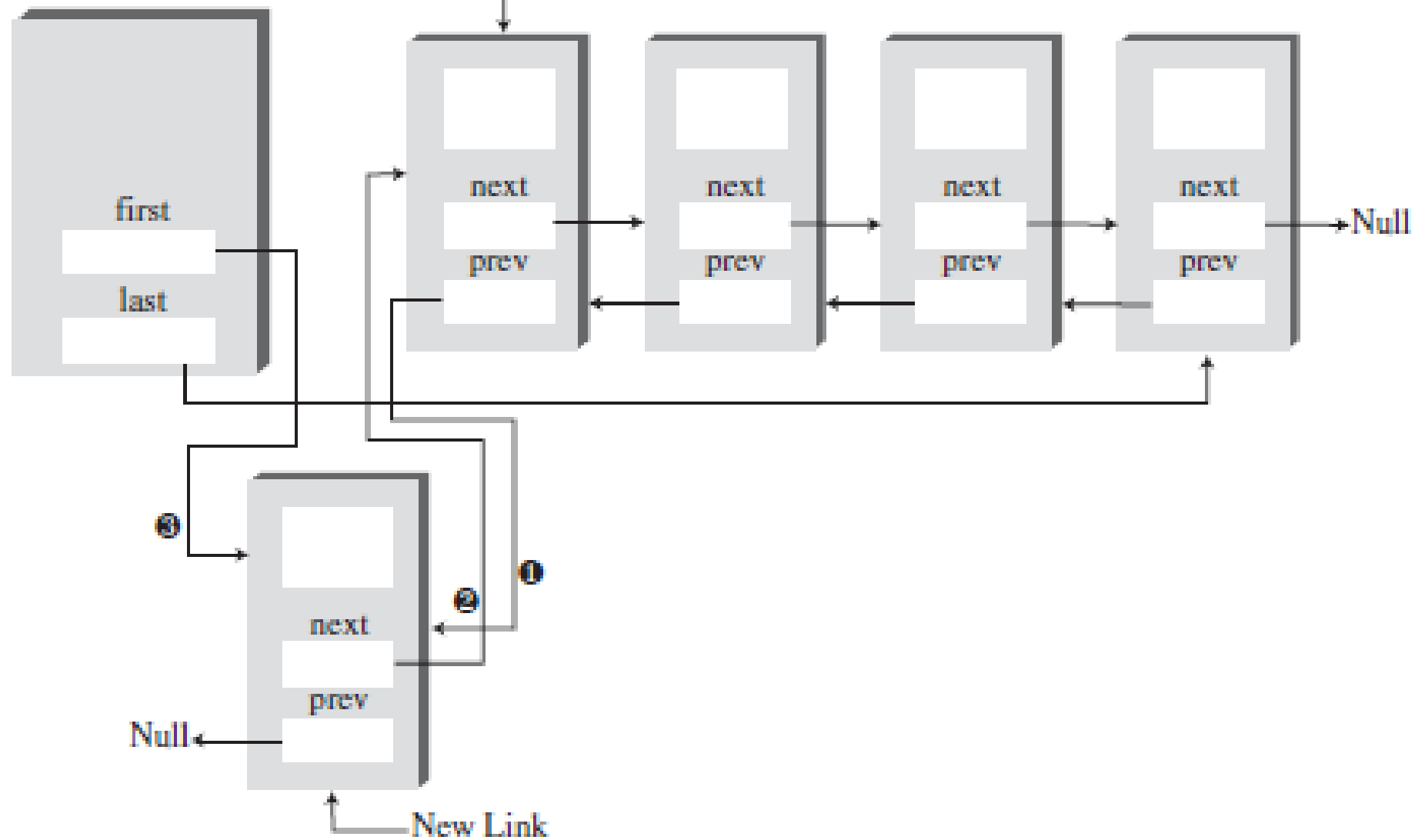
O altă variantă de listă dublu înlănțuită este o **listă circulară cu element santinelă**

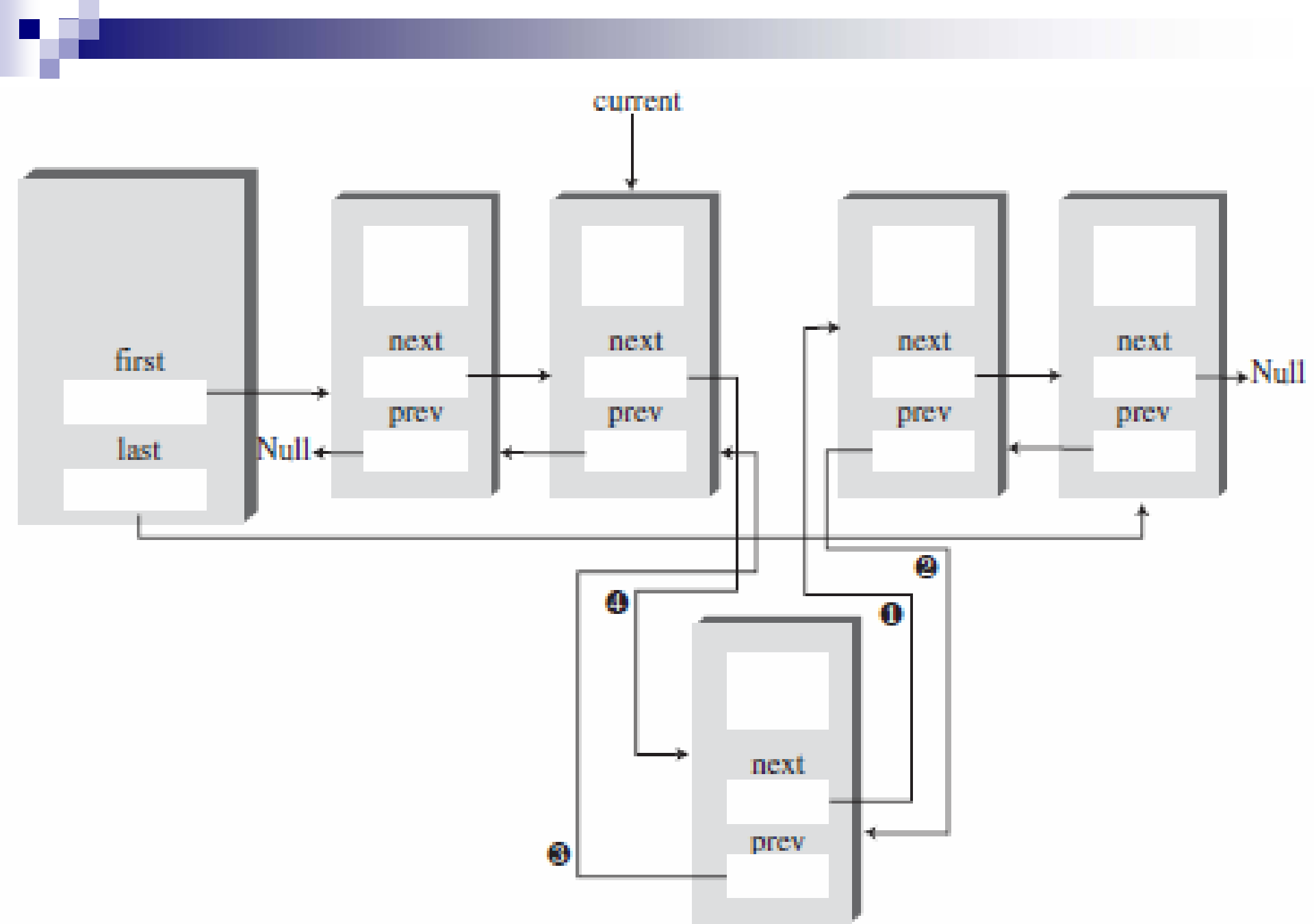
La crearea listei se alocă elementul santinelă

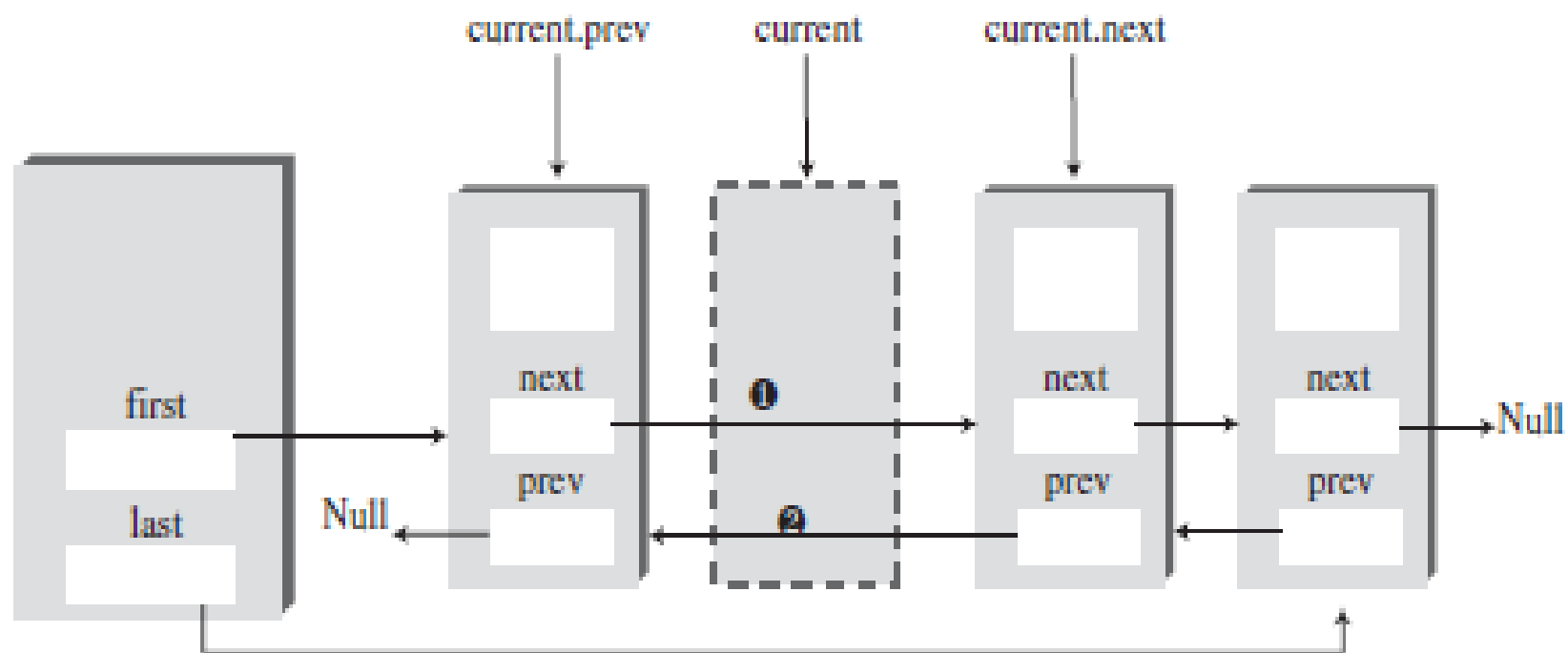
Exemplu de inițializare

```
void initDL (DList & lst) {  
    lst = (Nod*)malloc(sizeof(Nod));  
    lst->next = lst->prev = lst;  
}
```

Old first link







Observații

- Pentru a efectua în aceeași manieră operațiile de inserare și ștergere pentru fiecare element al listei, se folosesc două celule santinelă, așezate la începutul și la sfârșitul listei, înaintea primului element și, respectiv după ultimul element al listei
- În acest fel, toate elementele listei au un predecesor și un succesor




Operații cu liste dublu înlanțuite

- Inserarea unui element în listă
- Ștergerea unui element din listă
- Căutarea unui element în listă
- Parcurgerea listei, folosind una din cele două adrese de legătură

Exemplu cu liste dublu înlanțuite

- Program care citește de la tastatură șiruri de caractere, până la citirea șirului “end”
- Șirurile sunt memorate într-o listă dublu înlanțuită, cu celule santinelă
- Programul afișează pe ecran șirurile de caractere în ordine normală și în ordine inversă

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
struct lista {
    char str[20];
    lista *prev, *next;
} *p, *sant1, *sant2;
int main() {
    char s[20];
    sant1 = new lista;
    sant2 = new lista;
    sant1->next = sant2;
    sant2->prev = sant1;
    printf("Introduceti un cuvant: ");
    gets(s);
```



```
while(strcmp(s, "end") != 0) {  
    p = sant2;  
    //cuvantul se leaga in locul santinelei 2  
    sant2 = new lista;  
    strcpy(p->str, s);  
    p->next = sant2;  
    sant2->prev = p;  
    printf("Introduceti un cuvant: ");  
    gets(s);  
}
```



```
//traversarea listei pe adresele din next
```

```
p = sant1->next;
```

```
//se pleaca de la urmatorul santinelei 1
```

```
if (p == sant2) printf("Lista este vida\n");
```

```
else {
```

```
    printf("Lista parcursa de la inceput la sfarsit: ");
```

```
    do {
```

```
        printf("\n %s", p->str);
```

```
        p = p->next;
```

```
    } while (p != sant2);
```

```
}
```

```
printf("\n");
```



```
//traversarea listei pe adresele din prev
p = sant2->prev;
//se pleaca de la precedentul santinelei 2
if (p == sant1) printf("Lista este vida\n");
else {
    printf("Lista parcursa de la sfarsit la inceput:
");
    do {
        printf("\n %s", p->str);
        p = p->prev;
    } while (p != sant1);
}
getch();
}
```

Comparație între vectori și liste

- Un vector este folosit când se face frecvent un acces aleator (nu secvențial) la elementele listei, ca în algoritmi de sortare, sau când este necesară o regăsire rapidă pe baza poziției în listă sau pentru listele al căror conținut nu se mai modifică și trebuie menținute în ordine (fiind posibilă și o căutare binară)
- Inserția și eliminarea de elemente în interiorul unui vector au însă complexitatea dependentă de dimensiunea vectorului

Comparație între vectori și liste

- O listă înlănțuită se recomandă atunci când dimensiunea listei este greu de estimat, fiind posibile multe adăugări și/sau ștergeri din listă, sau atunci când sunt necesare inserări de elemente în interiorul listei
- Deși este posibil accesul pozițional, printr-un indice întreg, la elementele unei liste înlănțuite, utilizarea sa frecventă afectează negativ performanțele aplicației

Comparație între vectori și liste

- Dacă este necesară o colecție ordonată, atunci se va folosi o listă ordonată și prin procedura de adăugare la listă nu se face o reordonare a listei înlănțuite
- În cazul vectorilor, se adaugă la sfârșit și se ordonează numai la afișare sau când este necesar

Liste Skip

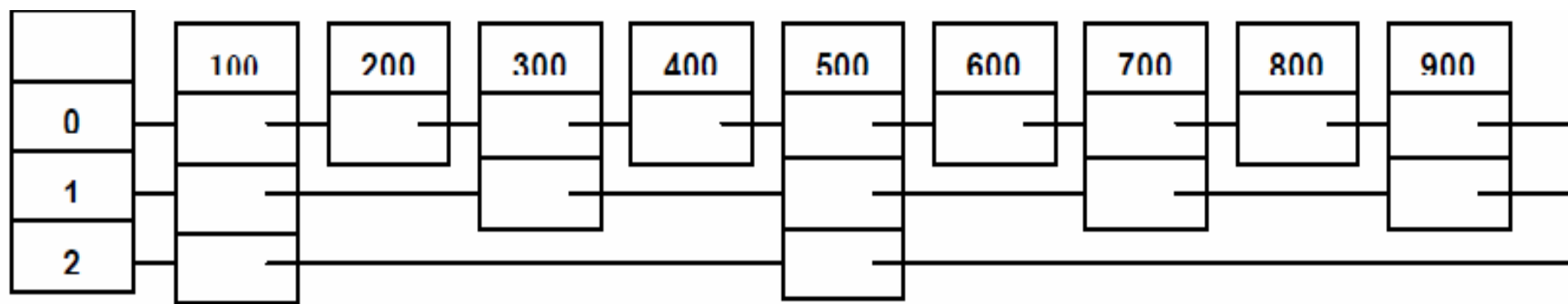
- Dezavantajul principal al listelor înlănțuite este timpul de căutare a unei valori date, prin acces secvențial
- Acest timp este proporțional cu lungimea listei
- De aceea, s-a propus o soluție de reducere a acestui timp prin utilizarea de pointeri suplimentari în anumite elemente ale listei

Liste Skip

- **Skip List** sunt **liste ordonate cu timp de căutare mai mic**
- Adresele de legătură între elemente sunt situate pe câteva niveluri: pe nivelul 0 este legătura la elementul imediat următor din listă, pe nivelul 1 este o legătură la un element aflat la o distanță d_1 , pe nivelul 2 este o legătură la un element aflat la o distanță $d_2 > d_1$ ș.a.m.d.

Liste Skip

- Adresele de pe nivelurile 1, 2, 3 și următoarele permit “salturi” în listă, pentru a ajunge mai repede la elementul căutat
- O listă skip poate fi privită ca fiind formată din mai multe liste paralele, cu anumite elemente comune




Liste Skip

- Căutarea începe pe nivelul maxim și se oprește la un element cu valoare mai mică decât cel căutat, după care continuă pe nivelul imediat inferior ș.a.m.d.
- Căutarea valorii 800 începe pe nivelul 2, “sare” direct și se oprește la elementul cu valoarea 500
- Se trece apoi pe nivelul 1 și se sare la elementul cu valoarea 700, după care se trece pe nivelul 0 și se caută secvențial între 700 și 900

Liste Skip

- Pointerii de pe nivelurile 1, 2 etc. împart lista în subliste de dimensiuni apropiate, cu posibilitatea de a sări peste orice sublistă, pentru a ajunge la elementul căutat
- Pentru simplificarea operațiilor cu liste skip, ele au un element santinelă (care conține numărul maxim de pointeri) și un element terminator cu o valoare superioară tuturor valorilor din listă sau care este același cu santinela (la liste circulare)



Fiecare nod conține un vector de pointeri la elementele următoare de pe câteva niveluri (dar nu și dimensiunea acestui vector) și un câmp de date

Exemplu de definire a unei liste cu salturi:

```
#define MAXLEVEL 11
// limita superioară pt nr max de pointeri pe nod
typedef struct Nod {
// structura care definește un nod de listă
int val;                // date din fiecare nod
struct Nod *leg[MAXLEVEL];
// legături la nodurile următoare
} Nod;
```

Liste neliniare

- Într-o **listă generală (neliniară)**, elementele listei pot fi de două tipuri:
 - elemente cu date (cu pointeri la date)
 - elemente cu pointeri la subliste
- O listă care poate conține subliste, pe oricâte niveluri de adâncime, este o listă neliniară

Liste neliniare

- Limbajul **Lisp** (“List Processing”) folosește liste neliniare, care pot conține atât valori atomice (numere, șiruri), cât și alte (sub)liste
- Listele generale se reprezintă în limbajul Lisp prin expresii
- O expresie Lisp conține un număr oarecare de elemente (posibil zero), încadrate între paranteze și separate prin spații
- Un element poate fi un atom (o valoare numerică sau un șir) sau o expresie Lisp

Liste neliniare

- În aceste liste se pot memora expresii aritmetice, propoziții și fraze dintr-un limbaj natural sau chiar programe Lisp
- Exemple de liste ce corespund unor expresii aritmetice în formă prefixată (operatorul precede operanzii):

$(- 5 3) \quad 5 - 3$

o expresie cu 3 atomi

$(+ 1 2 3 4) \quad 1 + 2 + 3 + 4$

o expresie cu 5 atomi

$(+ 1 (+ 2 (+ 3 4))) \quad 1 + 2 + 3 + 4$

o expresie cu 2 atomi și o

subexpresie

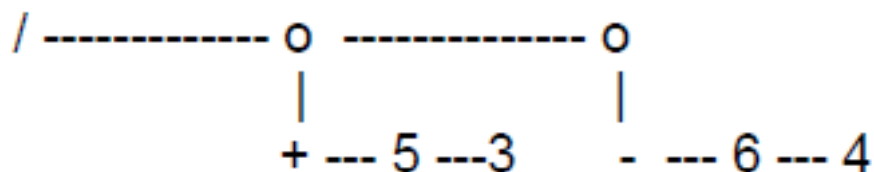
$(/ (+ 5 3) (- 6 4)) \quad (5 + 3) / (6 - 4)$


o expresie cu un atom și 2

subexpresii

Fiecare element al unei liste Lisp conține două câmpuri, numite **CAR** (primul element din listă) și **CDR** (celelalte elemente sau restul listei). Primul element dintr-o listă este de obicei o funcție sau un operator, iar celelalte elemente sunt operanzi.

Imaginea unei expresii Lisp ca listă neliniară, cu două subliste:





O implementare eficientă a unei liste Lisp folosește două tipuri de noduri: noduri cu date (cu pointeri la date) și noduri cu adresa unei subliste

Este posibilă și utilizarea unui singur tip de nod cu câte 3 pointeri: la date, la nodul din dreapta (din aceeași listă) și la nodul de jos (sublista asociată nodului)

Se consideră că elementele atomice memorează un pointer la date și nu chiar valoarea datelor, pentru a permite șiruri de caractere ca valori

