Structuri de date și algoritmi
Liste

Lect. Dr. Ing. Cristian Chilipirea – cristian.chilipirea@mta.ro

RECAPITULARE

pointeri
vectori
matrici
struct
Limbaj C

Recapitulare - pointeri

Variabilă ce reține adresa în memorie a unui obiect.

Declarație

- type *nume;
- int *myPointer;
- char *myPointer;

Extragere valoare

- int a = *myPointer;

Extragere pointer

- int *muPointer = &a;

Pointeri – operații

pointerNou = pointer + întreg

- Pointerul mai dreapta cu un întreg număr de elemente
- Ține cont de dimensiunea elementelor

întreg = pointerA – pointerB

- numărul de elemente între cei doi pointeri

pointer++

pointer—

pointerNou = pointer + pointer ?

pointerNou = pointer * pointer ?

Pointeri – operații

- pointerNou = pointer + întreg
  - Pointerul mai dreapta cu un întreg număr de elemente
  - Ține cont de dimensiunea elementelor
- întreg = pointerA – pointerB
  - numărul de elemente între cei doi pointeri
- pointer++
- pointer--
- ~~pointerNou = pointer + pointer ?~~
- ~~pointerNou = pointer * pointer ?~~

Vectori – alocare statică

type myVector[N];

char myVector[10];

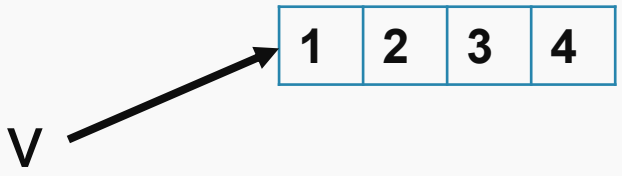
- 10 elemente alocate
- Numerotate de la 0

[0]	[1]	[2]	[3]	[4]	[5]	[6] …
1	2	3	4	5	6	7

myVector[3] == 4

Vectori - alocare dinamică

```
int* v;
v = (int*)malloc(N * sizeof(int));
```



v[3] == 4

[0]	[1]	[2]	[3]
1	2	3	4

Matrici

O matrice reprezintă un set de variabile grupate

Poate fi:

- 1D – vector (N elemente)
- 2D – matrice (N*M elemente)
- 3+D – matrice multidimensională (N*M*....*Z elemente)

Elementele sunt adresabile direct (caz 2D):

- matrice[i][j]; Elementul de pe poziția i*N+j
- matrice[3][4]; Elementul de pe poziția 3*N+4

Matrice

type myMatrix[N][M];

int myMatrix[10][20];

- 10*20 elemente allocate
- numerotare de la 0

	[][0]	[][1]	[][2]	[][3]	[][4]	[][5]	[][6] …
[0][]	1	2	3	4	5	6	7
[1][]	8	9	10	11	12	13	14
[2][]	15	16	17	18	19	20	21
[3][]	22	23	24	25	26	27	28
[4][]	29	30	31	32	33	34	35

…

myMatrix[3][4] == 26

Matrice – alocare statică

	[][0]	[][1]	[][2]	[][3]
[0][]	1	2	3	4
[1][]	8	9	10	11
[2][]	15	16	17	18
[3][]	22	23	24	25

int myMatrix[4][4]

- Rezultă în zonă continuă de memorie, rândurile sunt așezate unul după altul.

1	2	3	4	8	9	10	11	15	16	17	18	22	23	24	25

Matrice – alocare statică

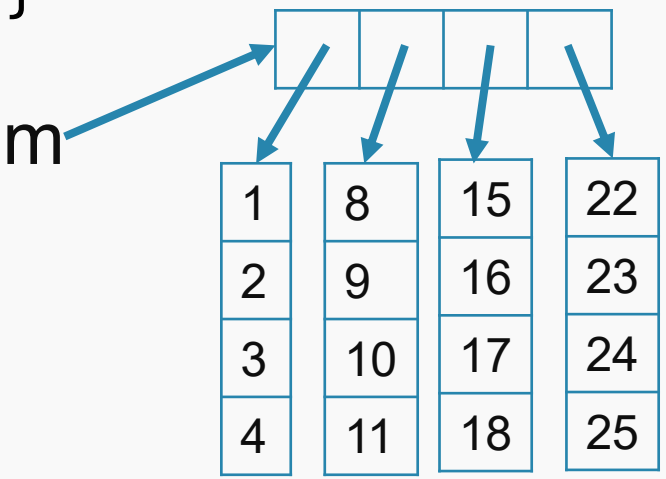
Cache friendly

Limitată ca spațiu (suntem pe stack)

1	2	3	4	8	9	10	11	15	16	17	18	22	23	24	25

Matrice - alocare dinamică

```
int** m;
m = (int**)malloc(N * sizeof(int *));
for (int i = 0; i < N; i++) {
    m[i] = (int*)malloc(N * sizeof(int));
}
```



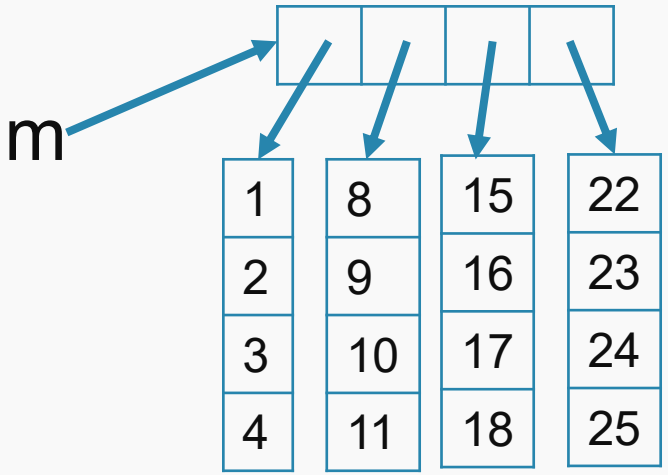
	[][0]	[][1]	[][2]	[][3]
[0][]	1	2	3	4
[1][]	8	9	10	11
[2][]	15	16	17	18
[3][]	22	23	24	25

Matrice - alocare dinamică

Spațiu foarte mare (suntem pe heap)

Posibilitate ca fiecare rând să aibă altă mărime (periculos)

Nu este cache friendly



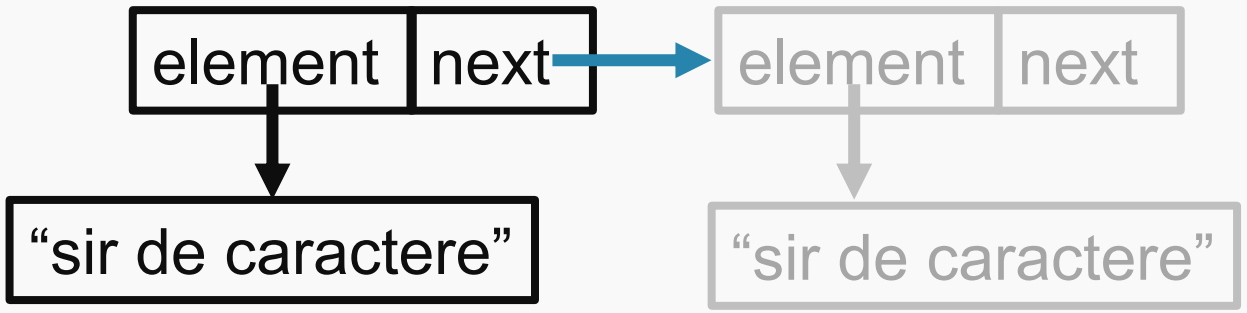
	[][0]	[][1]	[][2]	[][3]
[0][]	1	2	3	4
[1][]	8	9	10	11
[2][]	15	16	17	18
[3][]	22	23	24	25

Struct

```
struct name {
        type1 name1;
        type2 name2;
        …
};
typedef struct name {
        type1 name1;
        type2 name2;
        …
} newName;
```

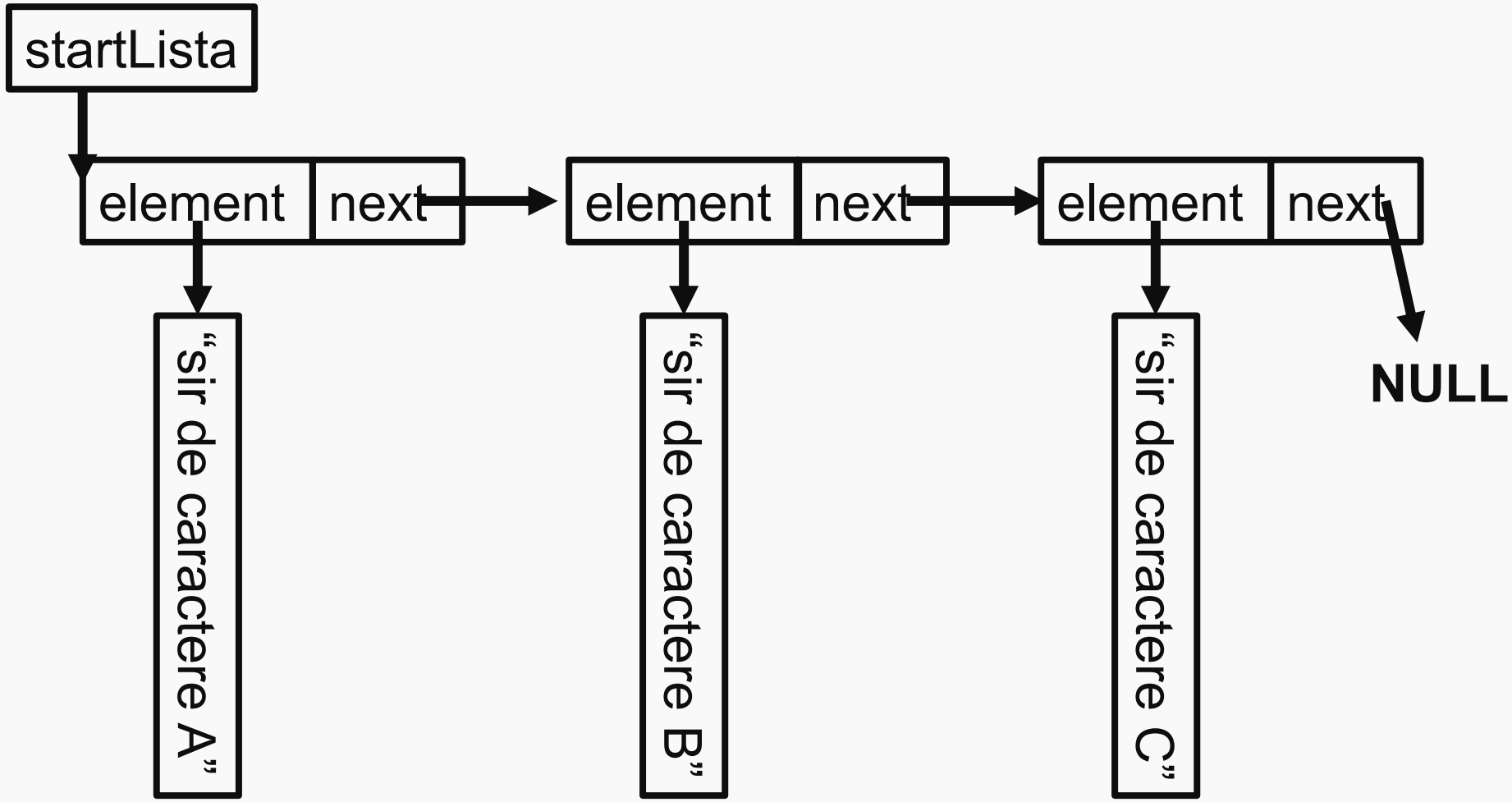
Liste înlănțuite

```
struct node {
    char* element;
    struct node* next;
};
```

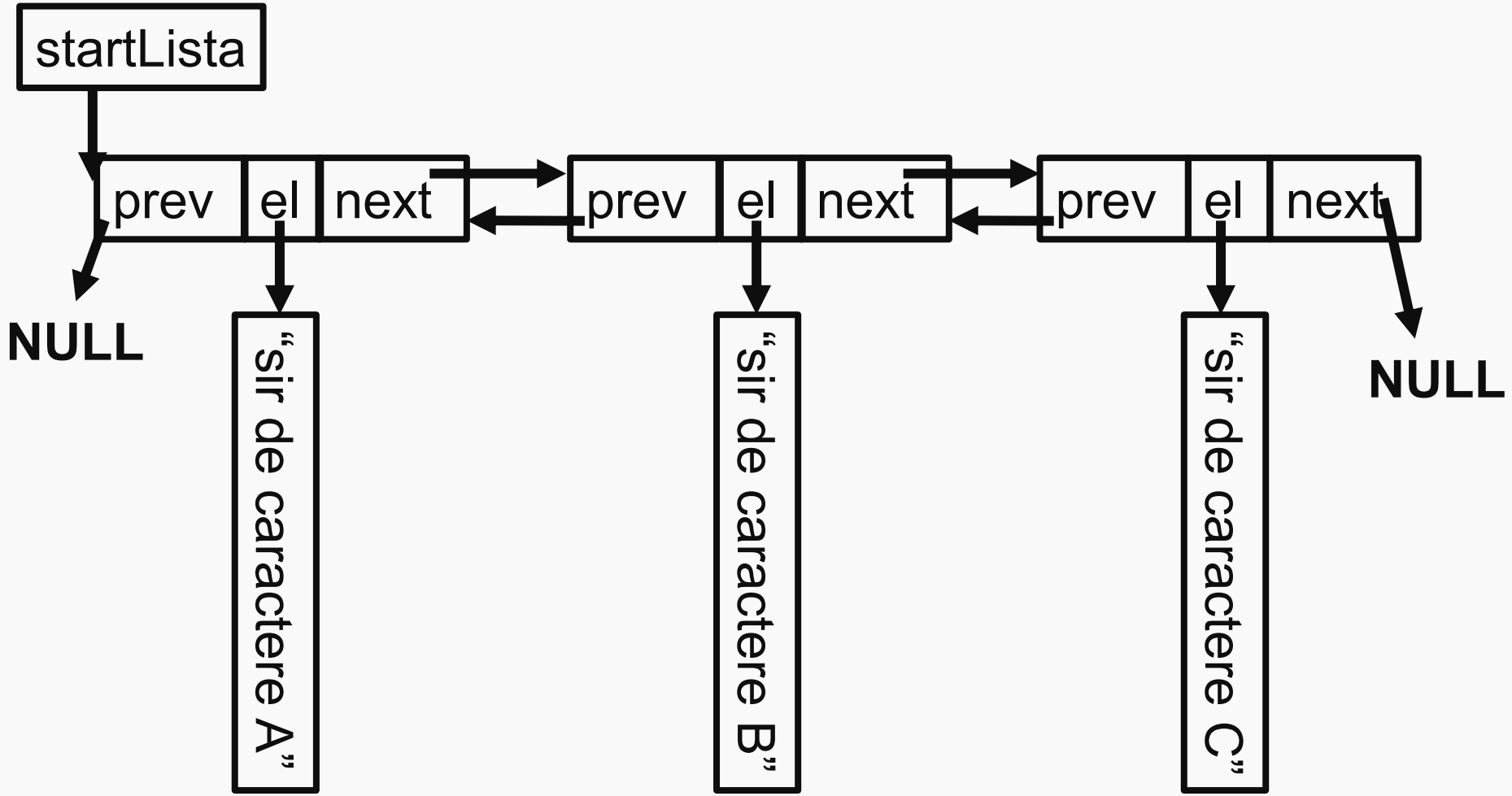



```
typedef struct node {
    char* element;
    struct node* next;
}listNode;
```

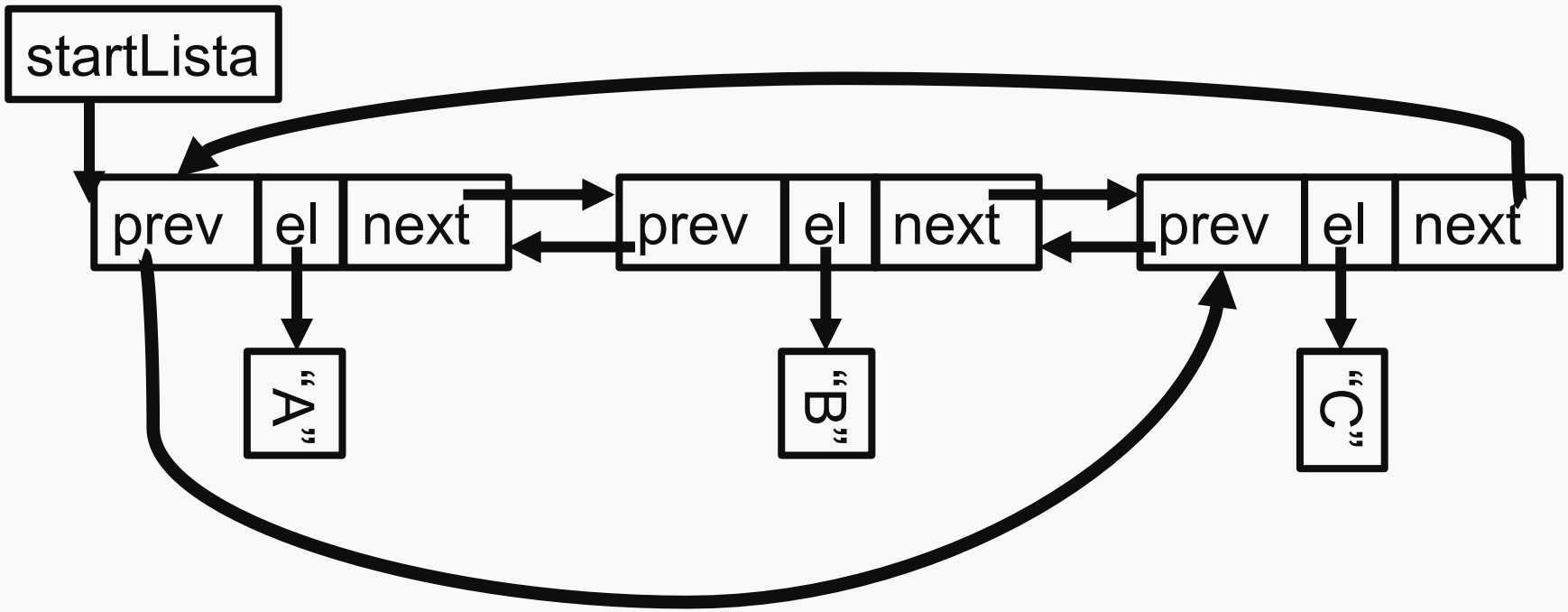
Liste înlănțuite



Liste dublu-înlănțuite



Liste circulare



Operații cu liste

Accesarea unui element

Inserare element

Ștergere element

Accesarea unui element din listă

```
listNode *getNode(listNode *listNode, int poz)
{
    if (poz < 0) return NULL;
    if (listNode == NULL) return NULL;

    for (int i = 0; i < poz; i++) {
        if (listNode->next == NULL)
            break;
        listNode = listNode->next;
    }
    return listNode;
}
```

Inserare element

```
void insertNodeInList(listNode **headNode, char *element, int poz)
{
    if (headNode == NULL) return;
    if (poz < 0) return;
    listNode *newNode = (listNode *)malloc(sizeof(listNode));
    if (newNode == NULL) return;
    newNode->element = element;
    listNode *prevNode = getNode(*headNode, poz - 1);
    if (prevNode == NULL) {
        newNode->next = *headNode;  //headNode may be NULL
        *headNode = newNode;
    } else {
        newNode->next = prevNode->next;
        prevNode->next = newNode;
    }
}
```

Ștergere element din listă

```
void removeNodeFromList(listNode **listStart, int poz)
{
    if (listStart == NULL) return;
    if (*listStart == NULL) return;
    if (poz < 0) return;
    listNode *prevNode = getNode(*listStart, poz - 1);
    if (prevNode == NULL) {
        listNode *aux = (*listStart);
        *listStart = (*listStart)->next;
        free(aux);
        return;
    } else if (prevNode->next == NULL) { return;
    } else {
        listNode *aux = prevNode->next;
        prevNode->next = prevNode->next->next;
        // free(aux->element);
        free(aux);
    }
}
```