Gestiunea memoriei in C

Sunt trei functii C standard care se pot folosi pentru a aloca memorie: **malloc** , **calloc** , si **realloc** . Prototipurile lor, dupa cum sunt definite ın stdlib.h:

```
void calloc (size t nmemb, size_t size);

void malloc (size_t size);

void realloc (void *ptr, size_t size);
```

malloc e cea mai simpla alocare. Primeste ca argument marimea blocului de memorie ce se doreste a fi alocat ¸si returneaza un pointer la acesta. **calloc** primeste doua argumente, numarul de elemente(nmemb) si marimea fiecarui element(size). Marimea totala alocata este numar memb. De asemenea, calloc seteaza toata memoria pe care o returneaza la NULL. **realloc** este folosit pt a se realoca o sectiune de memorie care a fost alocata ınainte. Primeste un pointer (*ptr) ca prim argument. Al doilea argument(size) indica spatiul pe care pointerul nou trebuie sa il contina. Cateodata **realloc** trebuie sa mute memorie pentru a gasi spatiu pentru noul chunk. Astfel pointerul returnat poate fi diferit de pointerul trimis functiei. Se poate folosi **realloc** si pentru a reduce marimea unui bloc de memorie dandu-i o marime mai mica decat cea alocata initial. Functia **free**() are prototipul:

```
void free (void* ptr);
```

Este folosita pentru a dealoca memorie care a fost alocata inainte ın una din cele 3 metode mentionate precedent. Primeste un pointer ce refera memoria alocata. Dupa ce un bloc a fost eliberat, nu mai trebuie folosit.

Exemplu: Alocarea unei matrici de numere intregi

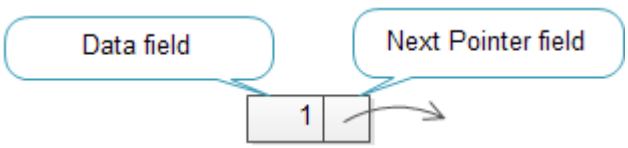
```
void main(){
    // number of rows and columns
    int n, m;
    // read in matrix dimensions
    printf("n=");
    scanf("%d",&n);
    printf("m=");
    scanf(""%d"",&m);
    // allocate space
    // this can be read as pointer to pointer
    int **a;
    a=(int **)calloc(n,sizeof(int *));
    if (a==NULL){
        perror("Insuficient memory");
```

```
            exit(1);
        }
        // allocate space for each uni-demensional array
        for(int i=0;i¡n;i++){
            a[i]=(int *)calloc(m,sizeof(int));
            if (a[i]==NULL){
            perror("Insuficient memory");
            exit(1);
        }
        }
        // read in matrix from keyboard
        for(i=0;i¡n;i++)
        {
            for(int j=0;j¡m;j++){
                printf("a[%d][%d]=\n",i,j);
                scanf("%d",&a[i][j]);
            }
        }
        // print array
        for(i=0;i¡n;i++)
        {
            for(int j=0;j¡m;j++){
                printf("a[%d][%d]=%d\n",i,j,a[i][j]);
            }
        }
        // free up memory. First the rows and then the full array
        a:
        for(int i=0;i¡n;i++) free(a[i]);
        free(a);
    }
```

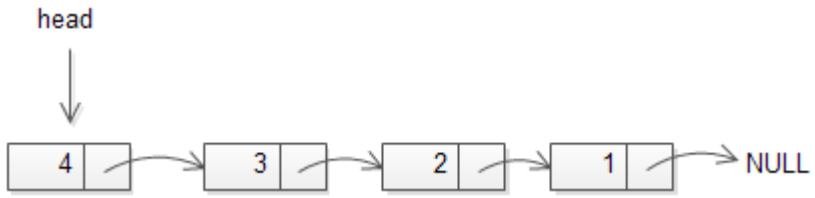
Exemplu: Lista Inlantuita

O lista inlantuita este o structura de date compusa dintr-o secventa de noduri, fiecare nod este (in general) compus din doua campuri:

- un camp de date (de tip int, double, char, o alta structura, etc)
- un pointer catre urmatorul nod



Pointer-ul **head** este folosit pentru a tine o referinta catre inceputul listei.



Putem defini un nod ca:

```
typedef struct node{
    int data;
    struct node* next;
}
```

Pentru a crea si popula un nod, putem folosi functia:

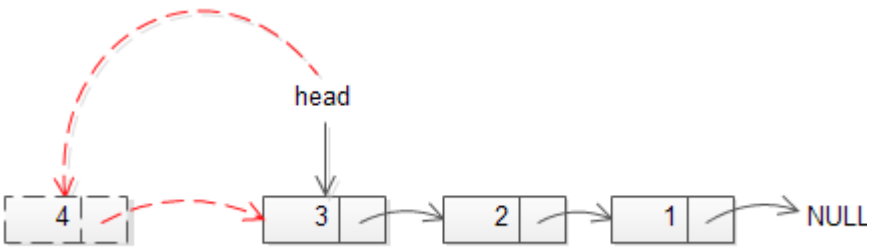
```
node* create(int data,node* next)
{
    node* new_node = (node*)malloc(sizeof(node));
    if(new_node == NULL)
    {
        printf("Error creating a new node.\n");
        exit(0);
    }
    new_node->data = data;
    new_node->next = next;

    return new_node;
}
```

Ca si parametri avem, continutul structurii, si un pointer catre urmatorul element din lista.

Pentru a creea un singur nod putem folosi:

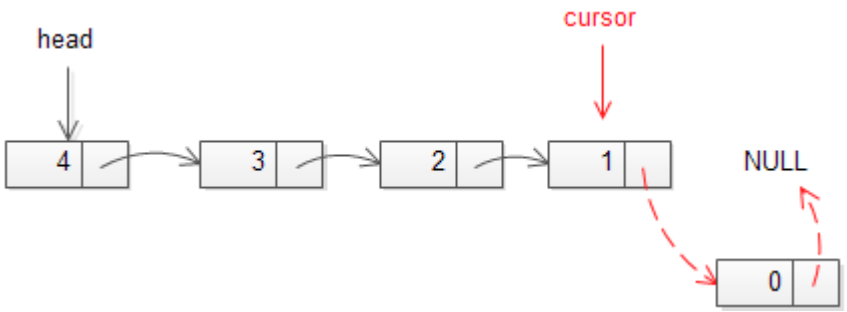
```
int data = 0 /* any number */
node* created_node = create(data,NULL)
```

Pentru a adauga un element nou la inceput de lista:



```
node* prepend(node* head,int data)
{
    // old head get's pushed back
    node* new_node = create(data,head);
    head = new_node;
    return head;
}
```

Pentru a adauga un element nou la sfarsit de lista:




```
node* append(node* head, int data)
{
    /* go to the last node */
    node *cursor = head;
    while(cursor->next != NULL)
        cursor = cursor->next;

    /* create a new node */
    node* new_node =  create(data,NULL);
    cursor->next = new_node;

    return head;
}
```

Pentru a printa lista:

```
void print(node* head)
{
    node* cursor = head;
    while(cursor != NULL)
    {
        printf("%d", cursor->data)
        cursor = cursor->next;
    }
}
```

Pentru a sterge lista din memorie:

```
void dispose(node *head)
{
    node *cursor, *tmp;
    // check that the list is not empty
    if(head != NULL)
    {
        // get the refferece to the next pointer before deleting it
        cursor = head->next;
        head->next = NULL;
        while(cursor != NULL)
        {
             // get the refferece to the next pointer before deleting it
            tmp = cursor->next;
            // free momory
            free(cursor);
            // go to next element in the list
            cursor = tmp;
        }
    }
}
```

Exemplu de uz:

```
int main()
{
    // initial pointer, not pointing to anything
    node* head = NULL;

    // create new element
    head = create(1,NULL);

    // prepend element to list, head pointer gets changed to point to newly
created     // node
    head = prepend(head, 2);
    head = append(head, 3);

    // print list
    print(head);

    // free up memory
    dispose(head);
    return 0;
}
```

Cerinte laborator:

Implementati o structura de lista dublu inlantuita (contine un camp de date, si doi pointeri, unul spre nodul urmator, si unul catre nodul anterior). Porniti de la structura:

```
typedef struct node{
    // data
    int data;
    // next element
    struct node* next;
    // previous element
    struct node* prev;
}
```

Adaugati functionalitatea de:

- Crearea unui nod
- Adaugare la inceputul listei
- Adaugare la sfarsitul litei
- Printare
- Stergere de memorie