Atelier de Programmation 2 : Langage C

Sami ZGHAL
sami.zghal@planet.tn

2011-2012

Plan

1. Récursivité
2. Type structure
3. Pointeurs
4. Structures de données
5. Performance des algorithmes

Récursivité

Définition

Un sous programme récursif (fonction ou procédure) est un sous programme qui peut s'appeler lors de son exécution

Récursivité

Énoncé

Écrire une fonction qui permet de calculer la somme des n premiers entiers $s = \sum_{i=1}^{n} i$

Fonction itérative

```
int somme (int x)
{
        int r;
        r=0;
        for(i=0;i<=n; i++)
        {r=r+i;}
        return r;
}
```

Récursivité

Énoncé

Écrire une **fonction récursive** qui permet de calculer la somme des n premiers entiers $s = \sum_{i=1}^{n} i$

Fonction récursive

```
int somme (int x)
{
        int r;
        if(x==0)
                r=0;
        else
                r=x+somme(x-1);
        return r;
}
```

Récursivité

Énoncé

Écrire une fonction qui permet de calculer le factoriel d'un entier n $(n \geq 0)$ $Fact(n) = 1 * 2 * 3 * ... * (n-1) * n$ et $Fact(0) = 1$

Fonction itérative

```
int Fact(int n)
{
        int r,i;
        r=1;
        for(i=1;i<=n;i++)
        {r=r*i;}
        return r;
}
```

Récursivité

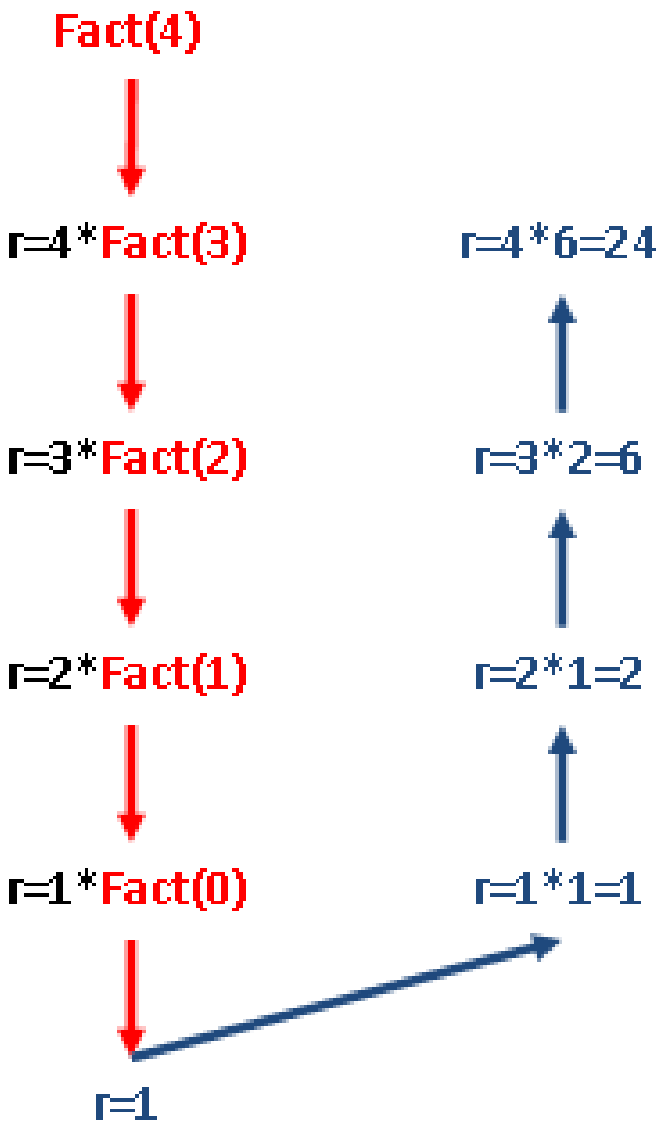
Énoncé

Écrire une **fonction récursive** qui permet de calculer le factoriel d'un entier n ($n \geq 0$) $Fact(n) = 1 * 2 * 3 * ... * (n-1) * n$ et $Fact(O) = 1$

Fonction récursive

```
int Fact(int n)
{
        int r;
        if(n==0)
                r=1;
        else
                r=n*Fact(n-1);
        return r;
}
```

Récursivité



Plan

1. Récursivité
2. Type structure
3. Pointeurs
4. Structures de données
5. Performance des algorithmes

Type structure

Définition

1. Type structure permet de regroupant des objets (des variables) de différents types au sein d'une même entité repérée par un seul nom de variable
2. Objets contenus dans une structure sont appelés champs de la structure

Type structure

Exemple

- Type personne : nom (chaîne de caractères), prénom (chaîne de caractères) et age (entier)
- Type complexe : partie réelle (réel) et partie imagiaire (réel)
- Type voiture : type (chaîne de caractères), marque (chaîne de caractères) et puissance (entier)
- etc.

Type structure

Déclaration type structure

```
typedef struct
{
        type_champ1 Nom_Champ1;
        type_champ2 Nom_Champ2;
        type_champ3 Nom_Champ3;
        type_champ4 Nom_Champ4;
        type_champ5 Nom_Champ5;
        ...
} NomType;
```

Type structure

Exemple : type personne

- Nom : chaîne de caractères
- Prénom : chaîne de caractères
- Age : entier

Déclaration : type personne

```
typedef struct
{
        char nom[15];
        char prenom[20];
        int age;
} personne;
```

Type structure

Exemple : type nombre complexe

- Patie réelle : réel
- Partie imaginaire : réel

Déclaration : type nombre complexe

```
typedef struct
{
        float pr;
        float pi;
} complexe;
```

Type structure

Exemple : type étudiant

- Nom & prénom : chaîne de caractères
- Notes : tableau de 4 notes (réel)
- Moyenne : réel

Déclaration : type étudiant

```
typedef struct
{
        char nom[15];
        char prenom[15];
        float notes[4];
        float moyenne;
} etudiant;
```

Type structure

Décalaration d'une variablede type structure

NomTypeStructure NomVariable;

Déclaration : type étudiant

```
personne Paul, Ali;
personne T[10]; // Tableau de 10 personnes


complexe c1, c2, c3;
complexe T1[15]; // Tableau de 15 nombres complexes

etudiant e1, e2;
etudiant liste[12]; // Tableau de 12 etudiants
```

Type structure

Manipulation d'une variable de type structure

- Chaque variable de type structure possède des champs repérés avec des noms uniques
- Nom des champs ne suffit pas pour y accéder : n'ont de contexte qu'au sein de la variable structurée
- Pour accéder aux champs d'une structure : l'opérateur de champ (un simple point) est placé entre le nom de la variable structurée et le nom du champ

Nom_Variable.Nom_Champ;

Type structure

Exemple de manipulation

```
#include<stdio.h>
typedef struct
{
        float pr;
        float pi;
} complexe;
void main()
{
        complexe c1,c2,c3;
        c1.pr=12;
        c1.pi=43;
        c2=c1;
        c3.pr=c1.pr+c2.pr;
        c3.pi=c1.pi+c2.pi;
}
```

Plan

1. Récursivité
2. Type structure
3. Pointeurs
4. Structures de données
5. Performance des algorithmes

Pointeurs

1. Définition
2. Notion d'adresse
3. Adresse d'une variable
4. Intérêt des pointeurs
5. Déclaration d'un pointeur

Définition

Pointeur

Variable contenant l'adresse d'une autre variable d'un type donné

Utilisation

Technique de programmation très puissante, permettant de définir des structures dynamiques qui évoluent au cours du temps (par opposition aux tableaux par exemple qui sont des structures de données statiques, dont la taille est figée à la définition)

Notion d'adresse

Mémoire centrale

- Constituée de plein de petites cases de 8 bits (un octet)
- Variable, selon son type (donc sa taille), va ainsi occuper une ou plusieurs de ces cases (une variable de type char occupera une seule case, tandis qu'une variable de type long occupera 4 cases consécutives)
- Chacune de ces "cases" (appelées blocs) est identifiée par un numéro : adresse

Notion d'adresse

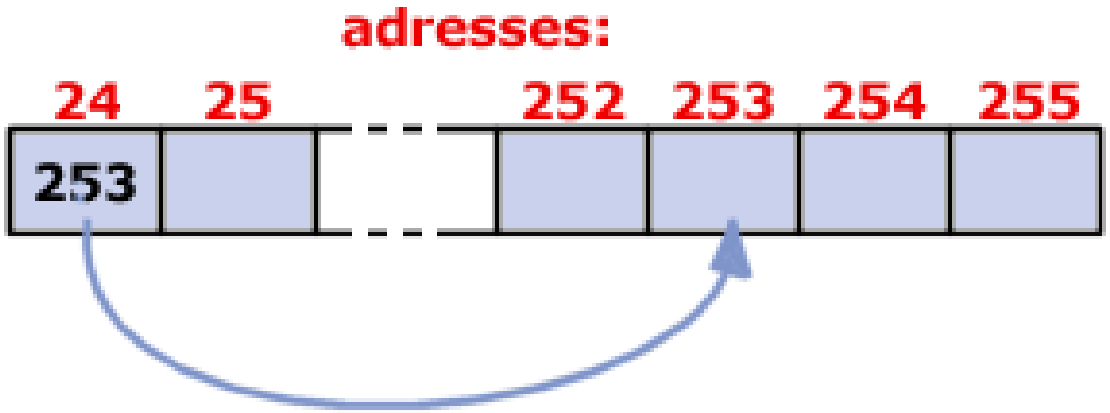
Variable

Une variable peut être accédé de 2 façons :

1. Grâce à son nom
2. Grâce à l'adresse du premier bloc alloué à la variable

Il suffit donc de stocker l'adresse de la variable dans un pointeur afin de pouvoir accéder à celle-ci (on dit que l'on "pointe vers la variable").

Notion d'adresse



Explication

- Le schéma ci-dessus montre par exemple par quel mécanisme il est possible de faire pointer une variable (de type pointeur) vers une autre
- Ici le pointeur stocké à l'adresse 24 pointe vers une variable stockée à l'adresse 253

Adresse d'une variable

Adresse

- En réalité vous n'aurez jamais à écrire l'adresse d'une variable, d'autant plus qu'elle change à chaque lancement de programme étant donné que le système d'exploitation alloue les blocs de mémoire qui sont libres, et ceux-ci ne sont pas les mêmes à chaque exécution
- Existe une syntaxe permettant de connaître l'adresse d'une variable, connaissant son nom : il suffit de faire précéder le nom de la variable par le caractère & pour désigner l'adresse de cette variable : &Nom_de_la_variable

Intérêt des pointeurs

Utilisation

Pointeurs possèdent un grand nombre d'intérêts :

- Permettent de manipuler de façon simple des données pouvant être importantes
- Tableaux ne permettent de stocker qu'un nombre fixé d'éléments de même type : en stockant des pointeurs dans les cases d'un tableau
- Possiblité de créer des structures chaînées : comportant des maillons

Déclaration d'un pointeur

Déclaration

- Variable qui doit être définie en précisant le type de variable pointée : type * Nom_du_pointeur;
- Type de variable pointée peut être aussi bien un type simple (int, char, float, etc.) ou un type complexe (struct)
- Grâce au symbole "*" : le compilateur sait qu'il s'agit d'une variable de type pointeur et non d'une variable ordinaire
- Étant donné que vous précisez (obligatoirement) le type de variable : compilateur saura combien de blocs suivent le bloc situé à l'adresse pointée

Déclaration d'un pointeur

Remarque

- Un pointeur doit préférentiellement être typé
- définir un pointeur sur "**void**" : sur quelque chose qui n'a pas de type prédéfini (void * toto)
- Ce genre de pointeur sert généralement de pointeur de transition, dans une fonction générique, avant un transtypage permettant d'accéder

Déclaration d'un pointeur

Initialisation d'un pointeur

```
int a = 2;

char b='Z';

int *p1;

char *p2;

p1 = &a;

p2 = &b;
```

Plan

1. Récursivité
2. Type structure
3. Pointeurs
4. Structures de données
5. Performance des algorithmes

Pointeurs

1. Liste
2. Pile
3. File
4. Arbre
5. Graphe

Liste

Définition

Suite d'éléments de même type

Exemples

- Tableau, pile, file, etc.
- Fichier

Organisation

- contiguë
- chaînée

Liste

Opérations sur les listes

- Initailiser
- Taille
- Ajouter à la fin
- Ajouter au début
- Chercher un élément
- Chercher la position d'un élement
- Ajouter après un élément
- Ajouter dans une position
- Vide
- Pleinne

Liste

Organisation contiguë : déclaration de type

```
#define max 50
typedef struct
{
        int T[max];
        int NE;
}liste;
```

Liste

Intialiser une liste

Procédure qui indique que la liste est constituée d'aucun élément

Implémentation

```
void initialiser(liste *l)
{
        l->NE=0;
}
```

Liste

Taille d'une liste

Fonction qui indique le nombre d'éléments appartenant à une liste

Implémentation

```
int taille(liste l)
{
 return l.NE;
}
```

Liste

Ajouter à la fin d'une liste

Procédure qui permet d'ajouter un élément à la fin de la liste

Implémentation

```
void AjouterFin(liste *l, int elt)
{
        if(l->NE<max)
        {
                l->T[l->NE]=elt;
                l->NE++;
        }
}
```

Liste

Ajouter au début d'une liste

Procédure qui permet d'ajouter un élément au début de la liste

Implémentation

```
void AjouterDebut(liste *l, int elt)
{
        int i;
        if(l->NE<max)
        {
                for(i=l->NE;i>=1;i--)
                {l->T[i]=l->T[i-1];}
                l->T[0]=elt;
                l->NE++;
        }
}
```

Liste

Chercher un élément dans une liste

Fonction qui permet de vérifier si un élément apparteient à la liste ou non

Liste

Implémentation

```
int ChercherElement(liste l,int elt)
{
        int i,trouve;

        i=0;
        trouve=0;
        while((i<=l.NE-1)&&(!trouve))
        {
                if(l.T[i]==elt)
                {trouve=1;}
                else
                {i++;}
        }
        return trouve;
}
```

Liste

Chercher la position d'un élément

Fonction qui permet de retourner la position d'un élément dans une liste (s'il existe) sinon -1

Liste

```
int ChercherPositionElement(liste l,int elt)
{
        int i,trouve,position;

        position=-1;
        i=0;
        trouve=0;
        while((i<=l.NE-1)&&(!trouve))
        {
                if(l.T[i]==elt)
                {
                        trouve=1;
                        position=i+1;
                }
                else
                {i++;}
        }
        return position;
}
```

Liste

Ajouter une valeur aprés un élément

Procédure qui permet d'ajouter une élement aprés une valeur donnée dans la liste

Liste

```
void AjouterApres(liste *l,int val,int elt)
{
        int i,trouve,position;
        if(l->NE<max)
        {       i=0; trouve=0;
                while((i<=l->NE-1)&&(!trouve))
                {   if(l->T[i]==val)
                        {       trouve=1; position=i;}
                        else
                        {i++;}
                }
                if(trouve)
                {       //décalage à droite
                        for(i=l->NE;i>=position+1;i--)
                        {l->T[i]=l->T[i-1];}
                        l->T[position]=elt;
                        l->NE++;
                }
        }
}
```

Liste

Ajouter une valeur dans une position

Procédure qui permet d'ajouter une élement dans une position donnée dans la liste

Implémentation

```
void AjouterPosition(liste *l, int position, int elt)
{
        if(l->NE<max)
        {
                for(i=l->NE;i>=position;i--)
                {l->T[i]=l->T[i-1];}
                l->T[position-1]=elt;
                l->NE++;
        }
}
```

Liste

Liste vide

Fonction qui permet de vérifier si la liste est vide ou non

Implémentation

```
int vide(liste l)
{
        return (l.NE==0);
}
```

Liste

Liste pleinne

Fonction qui permet de vérifier si la liste est pleinne ou non

Implémentation

```
int pleinne(liste l)
{
        return (l.NE==max);
}
```

Liste

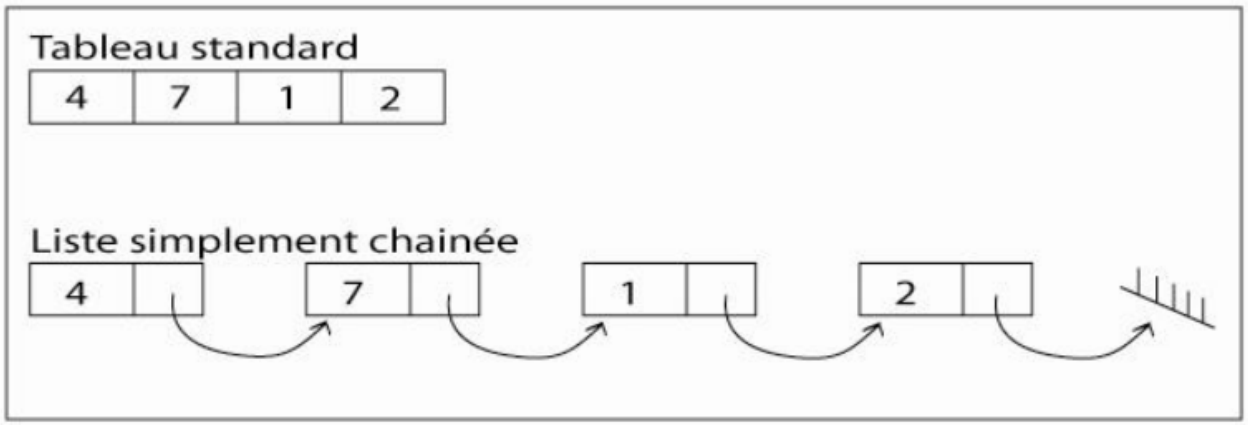
Inconvénients

- Gaspillage au niveau de le mémoire centrale : tableau réservé plus grand par rapport au nombre de cases exploitées
- Incapacité d'ajouter des valeurs : tableau réservé plus petit par rapport au nombre de cases exploitées

Solution

Exploitation d'une organisation chaînée

Liste



Liste chaînée

- Taille inconnue : peut avoir autant d'éléments que votre mémoire le permet
- Déclaration : créer le pointeur qui va pointer sur le premier élément de votre liste (aucune taille n'est donc à spécifier)
- Manipulation des pointeurs : ajouter, supprimer, intervertir des éléments de la liste

Liste

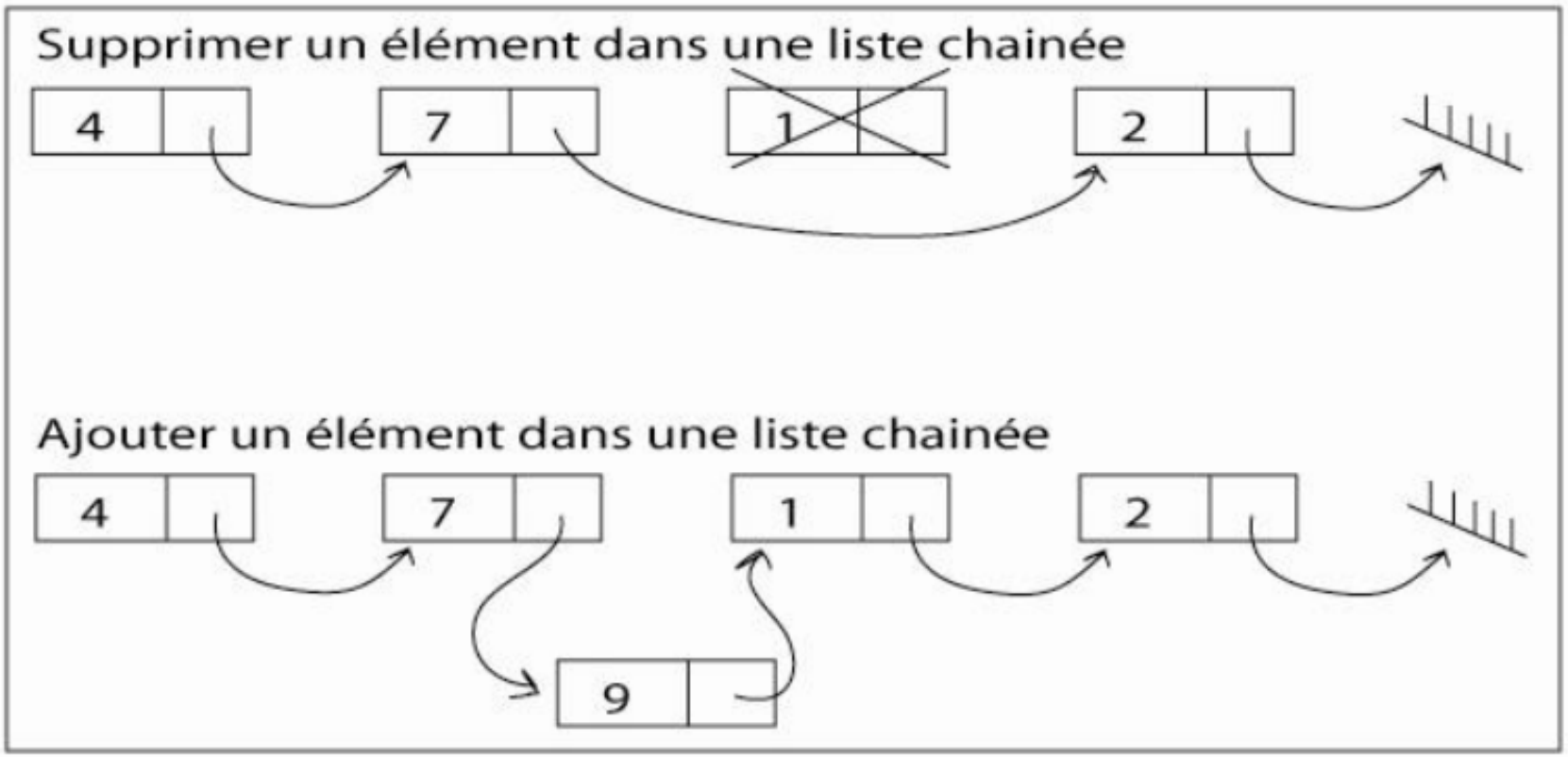
Élément de la liste

Chaque élément de la liste est composé de 2 parties :

1. Valeur à stocker
2. Adresse de l'élément suivant (s'il exist et dans le cas contraire l'adresse est NULL, et désignera le bout de la chaîne)

Liste

Ajout & supression dans une liste



Liste

Déclaration d'une liste chaînée

```
typedef struct element element;
struct element
{
    int val;
    struct element *nxt;
};

typedef element* llist;
```

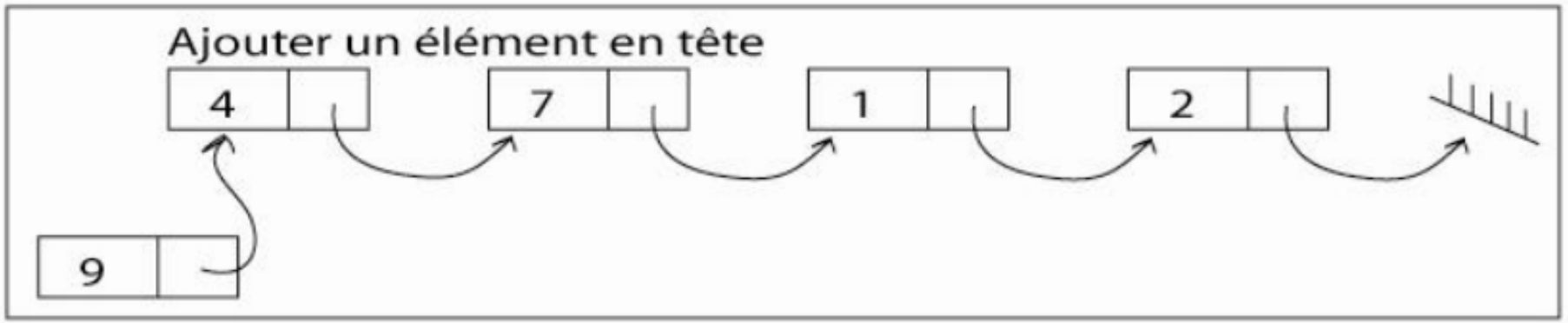
Liste

Initaliser la liste

```
llist initialiser()
{
        return NULL;
}
```

Liste

Ajouetr en tête



Liste

Ajouter en tête

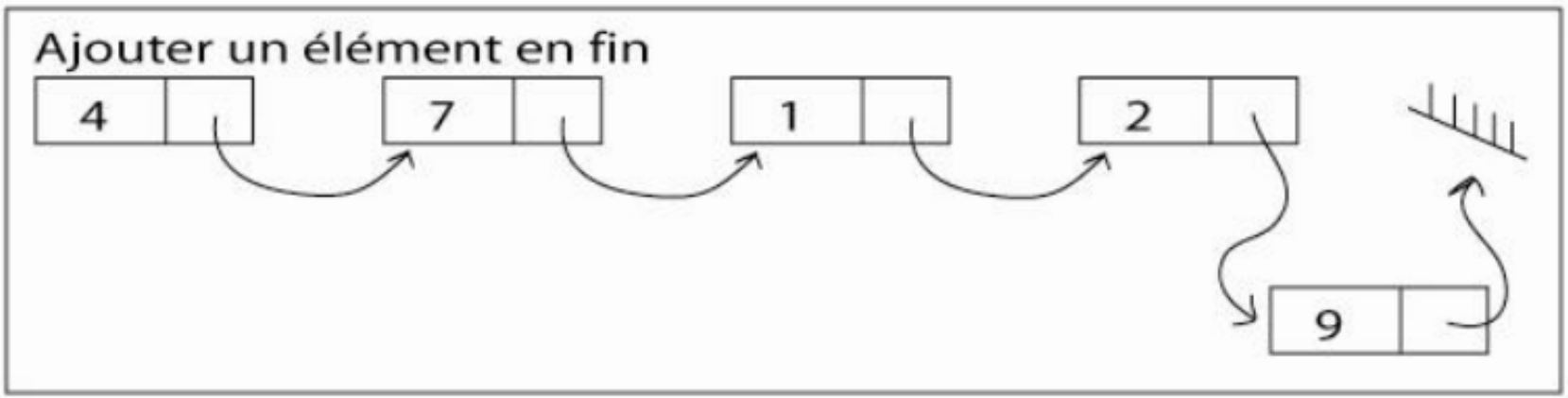
```
llist ajouterEnTete(llist liste, int valeur)
{

    element* nouvelElement;

        /* On crée un nouvel élément */
        nouvelElement = (element *)malloc(sizeof(element));
    /* On assigne la valeur au nouvel élément */
    nouvelElement->val = valeur;
    /* On assigne l'adresse de
        l'élément suivant au nouvel élément */
    nouvelElement->nxt = liste;
    /* On retourne la nouvelle liste,
        i.e. le pointeur sur le premier élément */
    return nouvelElement;
}
```

Liste

Ajouetr à la fin



Liste

Ajouter à la fin

```
llist ajouterEnFin(llist liste, int valeur)
{
    element* nouvelElement,*temp;
        nouvelElement = (element *)malloc(sizeof(element));
    nouvelElement->val = valeur;
    nouvelElement->nxt = NULL;
    if(liste == NULL)
    {   /* Si liste vide : renvoyer l'élément créé */
        return nouvelElement;
    }
    else
    {   /* Sinon, on parcourt la liste*/
        temp=liste;
        while(temp->nxt != NULL)
        { temp = temp->nxt; }
        temp->nxt = nouvelElement;
        return liste;
    }
}
```

Liste

Afficher une liste

```
void afficherListe(llist liste)
{
        llist tmp;
        tmp =liste;
    /* Tant que l'on n'est pas au bout de la liste */
    while(tmp != NULL)
    {
        /* On affiche */
        printf("%d ", tmp->val);
        /* On avance d'une case */
        tmp = tmp->nxt;
    }
}
```

Liste

Effacer la liste

```
llist effacerListe(llist liste)
{
    llist tmp,tmpnxt;

        tmp = liste;
    /* Tant que l'on n'est pas au bout de la liste */
    while(tmp != NULL)
    {   /* On stocke l'élément suivant pour
                pouvoir ensuite avancer */
        tmpnxt = tmp->nxt;
        /* On efface l'élément courant */
        free(tmp);
        /* On avance d'une case */
        tmp = tmpnxt;
    }
    /* La liste est vide : on retourne NULL */
    return NULL;
}
```

Liste

Effacer la liste

```
llist effacerListe(llist liste)
{
    llist tmp;
        if(liste == NULL)
    {   /* Si la liste est vide, il n'y
                a rien à effacer, on retourne
        une liste vide i.e. NULL */
        return NULL;
    }
    else
    {   /* Sinon, on efface le premier élément
                et on retourne le reste de la
        liste effacée */
        tmp = liste->nxt;
        free(liste);
        return effacerListe(tmp);
    }
}
```

Liste

Élément de la liste

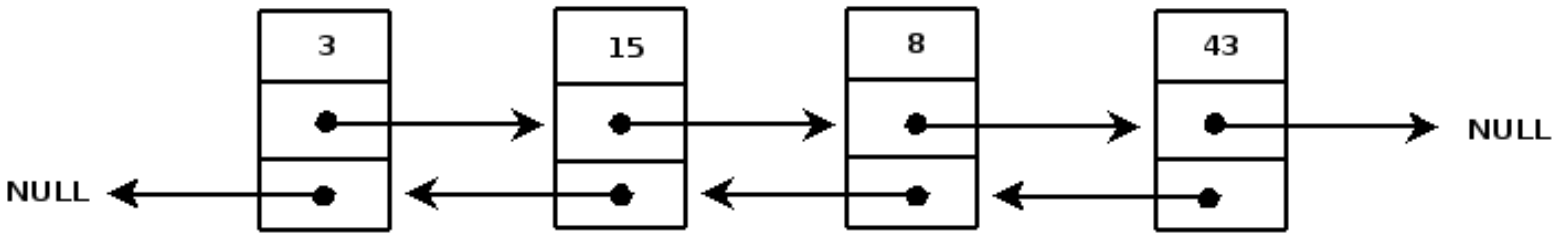
Chaque élément de la liste est composé de 3 parties :

1. Valeur à stocker
2. Adresse de l'élément suivant (s'il exist et dans le cas contraire l'adresse est NULL, et désignera le bout de la chaîne)
3. Adresse de l'élément précédent (s'il exist et dans le cas contraire l'adresse est NULL, et désignera le le début de la chaîne)

Liste

Représentation d'une liste doublement chaînée

Liste doublement chaînée de 4 valeurs



Liste

Déclaration d'une liste doublement chaînée

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

typedef struct dlist
{
    size_t length;
    struct node *debut;
    struct node *fin;
} Dlist;
```

Liste

Description

- length : représente le nombre d'éléments dans la liste (type size_t : correspond à un entier non signé - entier positif)
- p_tail : pointe vers le premier élément de la liste
- p_head : pointe vers le dernier élément de la liste

Liste

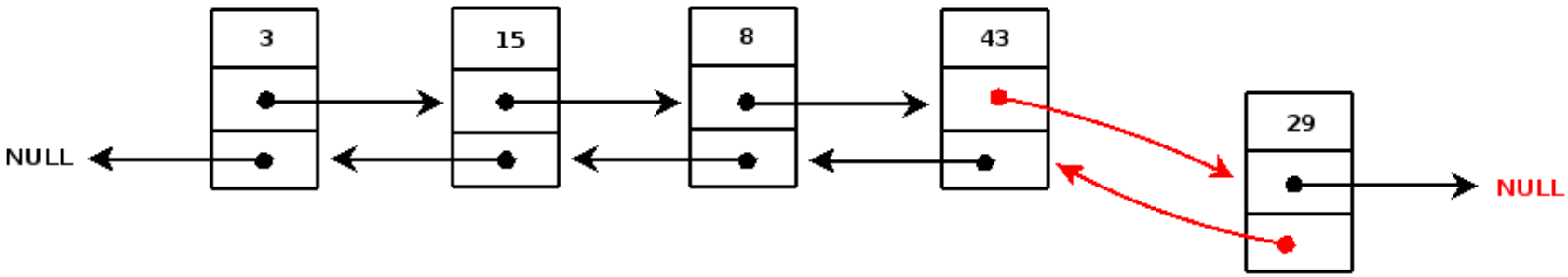
Initaliser la liste

```
Dlist * initialiser(void)
{
    Dlist *ld;
        ld = malloc(sizeof *ld);
    if (ld != NULL)
    {
        ld->length = 0;
        ld->debut = NULL;
        ld->fin = NULL;
    }
    return ld;
}
```

Liste

Ajouter en fin

Ajout d'un élément en fin de liste



Liste

Ajouter en fin (1/2)

```
Dlist * AjouterFin(Dlist *ld, int val)
{
        node * elt:
        /* Création d'un nouveau node */
    elt = malloc(sizeof *elt);
    /*Stocker la valeur*/
        elt->data = data;
    /*Pointer p_next vers NULL*/
        elt->next = NULL;
        if (ld->fin == NULL)
    { /*Cas où liste est vide*/
      /*Pointer p_prev vers NULL */
          e->prev = NULL;
      /*Pointer la tête de liste vers le nouvel élément*/
          ld->debut = elt;
          /*Pointer la fin de liste vers le nouvel élément*/
      ld->p_tail = elt;
    }
```

Liste

Ajouter en fin (1/2)

```
    else
      {   /* Cas où liste n'est pas vide*/
          /*Relier le dernier élément de la liste vers le nouvel élém
          ld->fin->pnext = elt;
              /*Pointer p_prev vers le dernier élément de la liste */
          elt->prev = ld->fin;
              /* Pointer la fin de liste vers notre nouvel élément (f
          ld->fin = elt;
      }
      ld->length++; /* Incrémentation de la taille de la liste */

      return p_list; /* on retourne notre nouvelle liste */
}
```

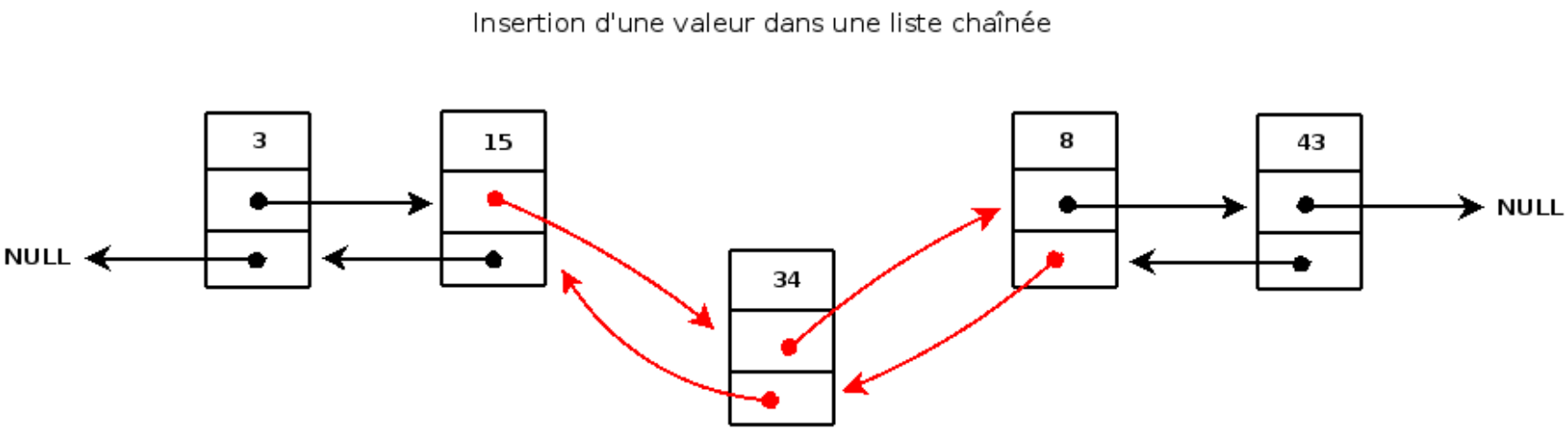
Liste

Ajouter au début

```
Dlist * AjouterDebut(Dlist *ld, int val)
{   node * elt;
    if (ld != NULL)
    {   elt = malloc(sizeof *elt);
            elt->data = val; elt->prev = NULL;
        if (ld->fin == NULL)
                    {   elt->next = NULL;
            ld->debut = elt;
            ld->fin = elt;
          }
          else
          {   ld->debut->prev = elt;
            elt->next = ld->debut;
            ld->debut = elt;
          }
          ld->length++;
    }
    return ld;
}
```

Liste

Ajouter dans une position



Liste

Ajouter dans une position (1/2)

```
Dlist *AjouterPosition(Dlist *ld, int data, int position)
{
        node * temp,* elt;
        int i;
    if (ld != NULL)
    {
        temp = ld->debut;
        i = 1;
        while (temp != NULL && i <= position)
        {
            if (position == i)
            {
                if (temp->next == NULL)
                {
                    ld = AjouterDebut(ld, data);
                }
                else if (temp->prev == NULL)
                {
                    ld = AjouterFin(ld, data);
```

Liste

Ajouter dans une position (2/2)

```
                else
                {
                    elt = malloc(sizeof *elt);
                    elt->data = data;
                    elt->next->prev = elt;
                    elt->prev->next = elt;
                    elt->prev = temp->prev;
                    elt->next = temp;
                    ld->length++;
                }
            }
            else
            {
                temp = temp->next;
            }
            i++;
        }
    }
    return ld;
```

Liste

Afficher une liste

```
void AfficherListe(Dlist *ld)
{
    node * temp;
        if (ld != NULL)
    {
        temp = ld->debut;
        while (temp != NULL)
        {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
    }
    printf("NULL\n");
}
```

Pile

Définition

- Structure de données permettant de stocker les données dans l'ordre LIFO (Last In First Out)
- Récupération des données se fait dans l'ordre inverse de leur insertion

Pile

Chaque élément de la pile

- Champ donnée
- Pointeur suivant

Pour permettre les opérations sur la pile

- Premier élément

Pile

Définition d'une pile

```
typedef struct element element;
struct element
{
    int valeur;
    struct element *suivant;
};

typedef element* pile;
```

Pile

Opérations sur la pile

- Initialiser
- Empiler : ajouter un élément au début (tête)
- Dépiler : retirer la tête
- Taille : nombre d'éléments
- Tête : donner la valeur en tête
- Afficher : affichage le contenu de la pile

Pile

Initialiser

```
pile initialiser()
{
        return NULL;
}
```

Pile

Empiler

```
pile empiler(pile p, int v)
{

    element* e;

        e = (element *)malloc(sizeof(element));
    e->valeur = v;
    e->suivant = p;
    return e;
}
```

Pile

Dépiler

```
pile depiler(pile p)
{
    element *e;

    if(p != NULL)
    {
        e=p;
                p=p->suivant;
        free(e);
    }

    return p;
}
```

Pile

Taille

```
int taille (pile p)
{
        int n;
        pile p1;

        p1=p;
        n=0;
        while (p1 != NULL)
        {
                n++;
                p1=p1->suivant;
        }
        return n;
}
```

Pile

Tête

```
int tete (pile p)
{
        if (p != NULL)
                return p->valeur;
}
```

Pile

afficher

```
void afficher(pile p)
{
        pile tmp;
        tmp = p;
    while(tmp != NULL)
    {
        printf("%d ", tmp->valeur);
        tmp = tmp->suivant;
    }
}
```

File

Définition

- Structure de données permettant de stocker les données dans l'ordre FIFO (First In First Out)
- Récupération des données se fait dans l'ordre de leur insertion

File

Chaque élément de la file

- Champ donnée
- Pointeur suivant

Pour permettre les opérations sur la file

- Premier élément

File

Définition d'une file

```
typedef struct element element;
struct element
{
    int valeur;
    struct element *suivant;
};

typedef element* file;
```

File

Opérations sur la file

- Initialiser
- Emfiler : ajouter un élément à la fin
- Dépifer : retirer la tête
- EstVide : vérifier si la file est vide
- Tête : donner la valeur en tête
- Afficher : affichage le contenu de la file

File

Initialiser

```
file initialiser()
{
        return NULL;
}
```

File

Emfiler

```
file emfiler(file f, int v)
{

    element* e, *temp;

        e = (element *)malloc(sizeof(element));
    e->valeur = v;
    e->suivant = NULL;
        if(f==NULL)
        {return e;}
    else
        {
                temp=f;
                while(temp->suivant!=NULL)
                {temp=temp->suivant;}
                temps->suivant=e;
                return f;
        }
}
```

File

Défiler

```
file defiler(pile f)
{
    element *e;

    if(f != NULL)
    {
        e=f;
                f=f->suivant;
        free(e);
    }

    return f;
}
```

File

Est vide

```
int EstVide (file f)
{
        if (f==NULL)
                return 1;
        else
                return 0;
}
```

File

Tête

```
int tete (pile f)
{
        if (f != NULL)
                return f->valeur;
}
```

File

Afficher

```
void afficher(file f)
{
        file tmp;
        tmp = f;
    while(tmp != NULL)
    {
        printf("%d ", tmp->valeur);
        tmp = tmp->suivant;
    }
}
```

Arbre

Graphe

Plan

1. Récursivité
2. Type structure
3. Pointeurs
4. Structures de données
5. Performance des algorithmes

Récursivité

Définition

Un sous programme récursif (fonction ou procédure) est un sous programme qui peut s'appeler lors de son exécution

Récursivité

Énoncé

Écrire une fonction qui permet de calculer la somme des n premiers entiers $s = \sum_{i=1}^{n} i$

Fonction itérative

```
int somme (int x)
{
        int r;
        r=0;
        for(i=0;i<=n;i++)
        {r=r+i;}
        return r;
}
```

Récursivité

Énoncé

Écrire une **fonction récursive** qui permet de calculer la somme des n premiers entiers $s = \sum_{i=1}^{n} i$

Fonction récursive

```
int somme (int x)
{
        int r;
        if(x==0)
                r=0;
        else
                r=x+somme(x-1);
        return r;
}
```

Récursivité

Énoncé

Écrire une fonction qui permet de calculer le factoriel d'un entier n $(n \geq 0)$ $Fact(n) = 1 * 2 * 3 * ... * (n-1) * n$ et $Fact(0) = 1$

Fonction itérative

```
int Fact(int n)
{
        int r,i;
        r=1;
        for(i=1;i<=n;i++)
        {r=r*i;}
        return r;
}
```

Récursivité

Énoncé

Écrire une **fonction récursive** qui permet de calculer le factoriel d'un entier n ($n \geq 0$) $Fact(n) = 1 * 2 * 3 * ... * (n-1) * n$ et $Fact(O) = 1$

Fonction récursive

```
int Fact(int n)
{
        int r;
        if(n==0)
                r=1;
        else
                r=n*Fact(n-1);
        return r;
}
```

Récursivité