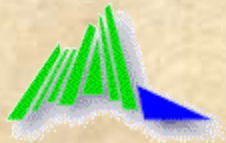


Module IC2
Algorithmique Avancée

Manuel Munier
IUT des Pays de l'Adour - Mont de Marsan
Département Réseaux Télécommunications
2012-2013

PPN

- Compétences minimales (être capable de)
  - Choisir et manipuler les structures de données avancées
  - Choisir et mettre en œuvre des algorithmes de recherche et de tri
- Contenu
  - S.D. avancées: listes, files, piles,…
  - Tris
  - Récursivité
  - Notions de complexité → « sensibilisation »

Plan

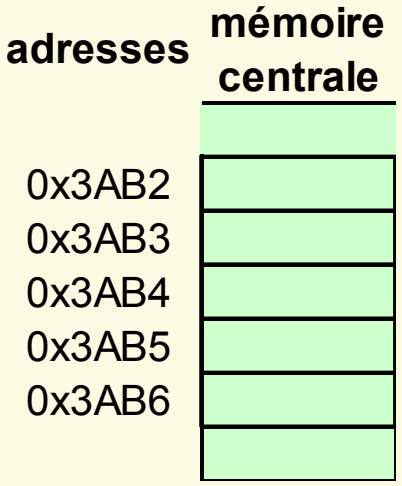
IC2

- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

Adresses & Pointeurs

- Adressage ?
  - mémoire centrale ⇔ suite d'octets ⇔ «tableau»
  - adresse ⇔ localisation ⇔ «indice»
  - valeur (d'une adresse) = fonction de la taille de l'espace d'adressage
    - Ex: espace de 64ko

    ⇒ adresses codées sur 4 octets

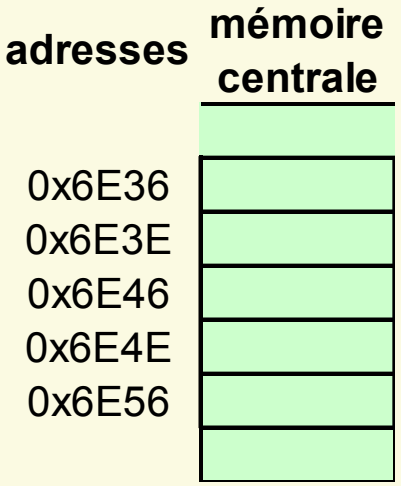


Adresses & Pointeurs

- Opérateur **&**
  - Permet de récupérer l'adresse d'une **lvalue**

```
...
double tab[5];
int i;
...
puts("Adresses des 5 éléments de tab:");
for (i=0; i<5; i++)
   printf("\tAdr elt %d: %X\n",i,&tab[i]);
...
```

```
Adresses des 5 éléments de tab
    Adr elt 0: 6E36
    Adr elt 1: 6E3E
    Adr elt 2: 6E46
    ...
```



type double sur 8 octets

Adresses & Pointeurs

- Notion de pointeur
  - pointeur = objet dont la valeur est une adresse
    - variables et constantes pointeurs, « pointeur constant », constante **`NULL`**
  - type pointeur ? ⇒ plusieurs types dérivés
    - « pointeur sur ‹*un_type*› »
    - ‹*un_type*› est dit « type pointé »

Cette terminologie sous-entend qu'un pointeur contient une adresse à laquelle est implanté un objet d'un type bien particulier (celui désigné par ‹*un_type*›).

En C, un pointeur fait référence **A LA FOIS à une adresse en mémoire ET à un type (pointé)**.

Adresses & Pointeurs

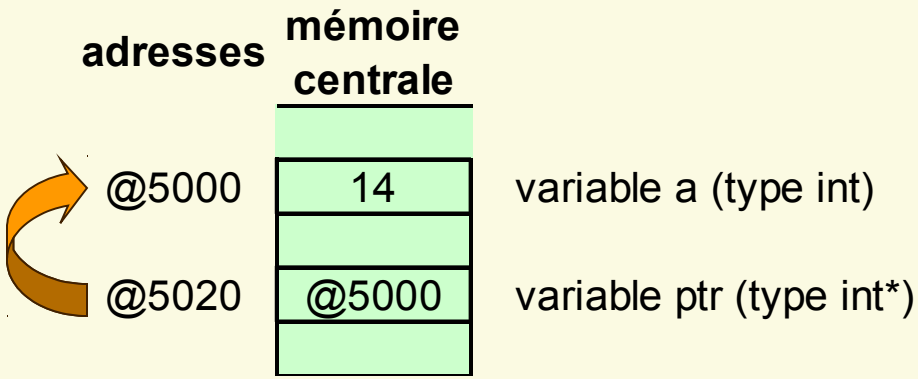
- Variables pointeurs et déclarateur *
  - **Déclaration**: *<id_type> *<id_var_ptr>;*
  - Exemples
    - `float *p;` `/* p = variable pointeur sur float */`
    - `char *t;` `/* t = pointeur sur caractère */`
    - `int *ptr;` `/* ptr = pointeur sur entier */`

    Remarque: ces 3 variables ont la même taille en mémoire centrale (ce sont 3 adresses) !

Adresses & Pointeurs

- Variables pointeurs et déclarateur *
  - **Initialisation** lors de la déclaration

```
int a=14;     /* a = var entière initialisée à 14 */
...
int *ptr=&a; /* ptr est un pointeur sur entier
                initialisé par un pointeur constant
                qui est l'adresse de la variable a */
```



Adresses & Pointeurs

- Opérations sur les adresses
  - **Affectation** (pointeurs de même type)

```
float rayon, *ptr;
...
ptr = &rayon;
```

    - Remarque: seule une lvalue peut être affectée

```
&rayon = NULL;    /* INTERDIT !!! */
```

  - **Comparaison** (pointeurs de même type)

```
int *ptr1, *ptr2, *ptr;
...
if (ptr1>ptr2) ...
...
while (ptr!=NULL) ... /* idem while (!ptr) */
```

Adresses & Pointeurs

- Opérations sur les adresses
  - **Ajout/retrait d'un entier** à un pointeur

    `P + val ⇔ P + (val * sizeof(<type_pointé_par_P>)`

    ➔adresse du val^{ième} élément après celui pointé par P

    ➔notion de déplacement
  - **Opérateurs**
    - `+, -` : lvalue et constante
    - `+=, -=, ++, --` : lvalue uniquement

Adresses & Pointeurs

- Opérations sur les adresses
  - **Conversion** d'un pointeur

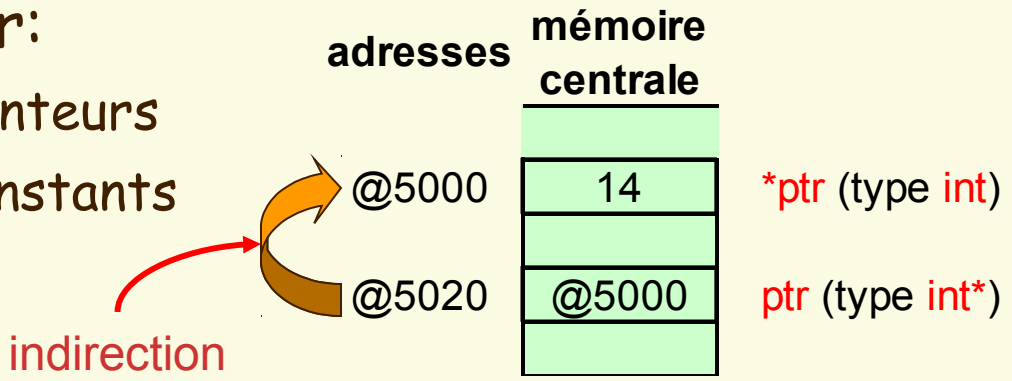
```
void *ptr;
int tab[10];
...
ptr = (int*)&tab[0];
```

➔ Pour information uniquement

Adresses & Pointeurs

- Opérateur d'indirection *
  - Permet d'accéder à l'élément pointé

    <pointeur>
  - Type du résultat: type pointé
  - Applicable sur:
    - variables pointeurs
    - pointeurs constants



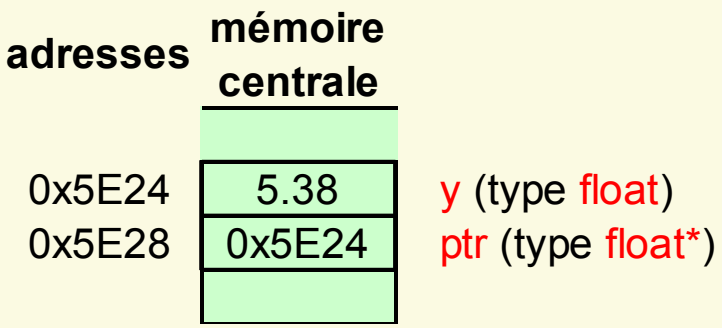
Adresses & Pointeurs

- Récapitulatif & et *
  - Programme

```
#include <stdio.h>
main()
{ float y=5.38;
  float *ptr=&y;
  printf("y vaut %f\n&y vaut %x\n",y,&y);
  printf("ptr vaut %x\n&ptr vaut %x\n*ptr vaut %f\n",ptr,&ptr,*ptr);
}
```

  - Résultats

```
y vaut 5.38
&y vaut 5E24
ptr vaut 5E24
&ptr vaut 5E28
*ptr vaut 5.38
```



Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - Principe de base
    - les arguments sont **obligatoirement** passés par valeur
  - Problème
    - comment contourner ce mode de fonctionnement ?
  - Solution
    - ne pas passer la **valeur** de l'argument mais son **adresse**

Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - Règles à respecter
    - déclarer tout paramètre formel correspondant à un argument à modifier comme étant un pointeur
    - dans le corps de la fonction, se souvenir que ces paramètres sont des pointeurs
      - ➔ * pour obtenir leur valeur réelle
    - à l'appel, considérer que les paramètres effectifs sont des pointeurs
      - ➔ & pour transmettre leur adresse

Pointeurs & Fonctions

- Modifier les arguments dans une fonction

```
#include <stdio.h>

void permut(int a,int b);

main()
{ int x=10,y=-5;
  printf("Avant x=%d et y=%d\n",x,y);
  permut(x,y);
  printf("Après x=%d et y=%d\n",x,y);
}

void permut(int a,int b)
{ int aux;

  aux=a;
  a=b;
  b=aux;
}
```

a et b sont des copies des **valeurs** de x et de y ➔ pas d'effet de bord sur x et y

Pointeurs & Fonctions

- Modifier les arguments dans une fonction

```
#include <stdio.h>

void permut(int *a,int *b);

main()
{ int x=10,y=-5;
  printf("Avant x=%d et y=%d\n",x,y);
  permut(&x,&y);
  printf("Après x=%d et y=%d\n",x,y);
}

void permut(int *a,int *b)
{ int aux;

  aux=*a;
  *a=*b;
  *b=aux;
}
```

a et b contiennent les **adresses** de x et de y ➔ x et y seront effectivement modifiés

Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **avant l'appel**

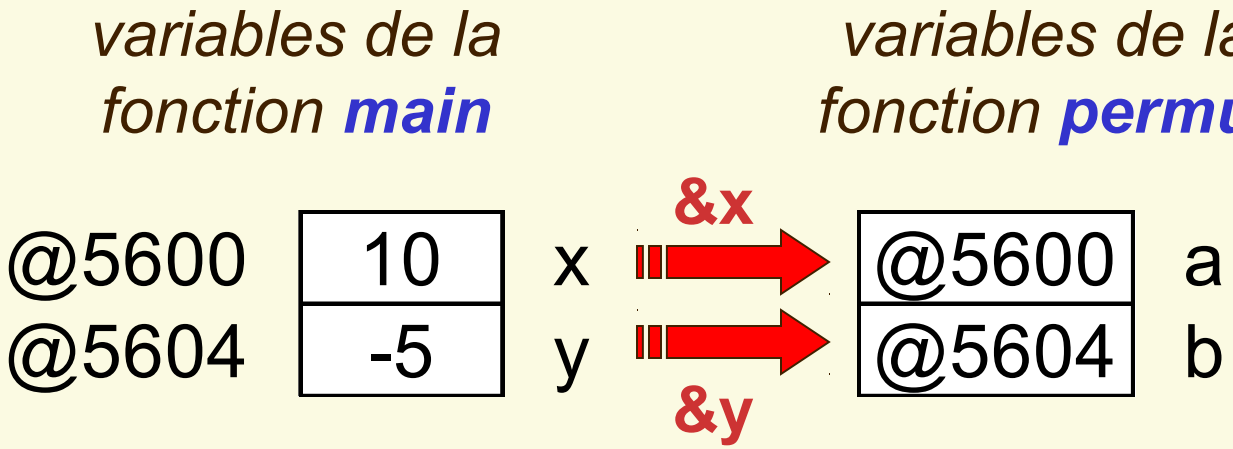
*variables de la fonction **main***

@5600	10	x
@5604	-5	y

*variables de la fonction **permut***

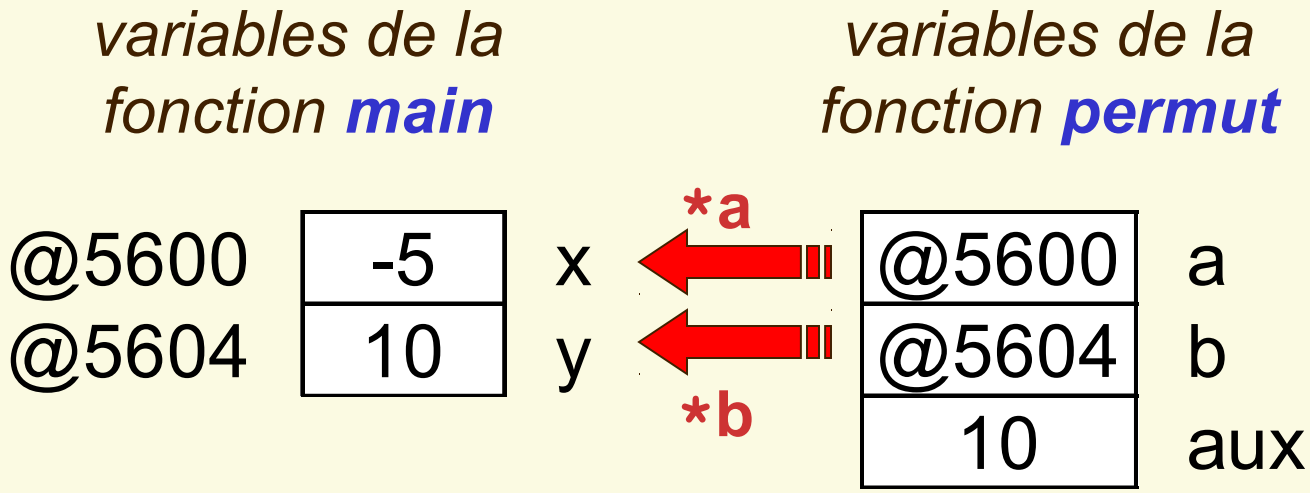
Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **lors de l'appel**



Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **pendant l'appel**



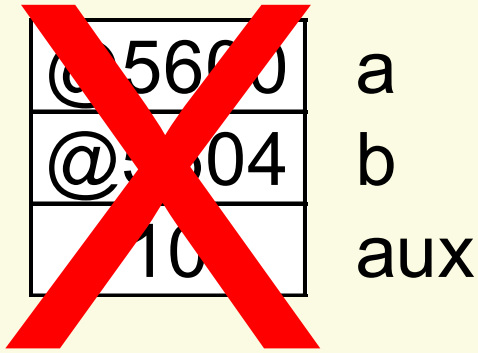
Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **après l'appel**

*variables de la fonction **main***

@5600	-5	x
@5604	10	y

*variables de la fonction **permut***



Pointeurs & Fonctions

- Récapitulatif
  - On fait toujours du passage par valeur
    - au lieu de passer (par valeur) l'information elle-même on passe (par valeur) l'adresse à laquelle se trouve cette information
  - C'est exactement le fonctionnement de la fonction `scanf`
    - on doit bien passer l'adresse (`&`) de la variable dans laquelle on veut stocker l'information saisie
  - **Attention aux effets de bord !!!**

Pointeurs & Tableaux

- Nom d'un tableau ⇔ pointeur constant
  - Définition: `<un_type> Tab[100];`
    - ➔ `Tab ⇔ &Tab[0]`
  - Au niveau des adresses:
    - `Tab ⇔ Tab+0 ⇔ &Tab[0]`
    - ➔ `Tab+i ⇔ &Tab[i]`
  - Au niveau des valeurs:
    - `*Tab ⇔ *(Tab+0) ⇔ *(&Tab[0]) ⇔ Tab[0]`
    - ➔ `*(Tab+i) ⇔ Tab[i]`

Pointeurs & Tableaux

- Nom d'un tableau ⇔ pointeur constant
  - Définition: `<un_type> Tab[100];`
    - ➔ `*(Tab+i)` ⇔ `Tab[i]`
  - Remarques:
    - **`*Tab + i` ≠ `*(Tab+i)`**
      - cf. règles de priorité sur les opérateurs
    - **`Tab` n'est pas une lvalue**
      - donc `Tab=...` ou `Tab++` sont interdits !

Plan

- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

Structures

- Principe et notions
  - **Structure** ⇔ ensemble de variables de types
    agrégation de variables en une seule entité identifiée de façon unique
    - **le modèle d'une structure** décrit le type des variables membres d'une structure ➔ c'est un **type**
    - **une instance d'une structure** a une existence réelle en mémoire déduite du modèle correspondant
      ➔ **variable** structurée (elle contient des valeurs)
  - **Champ** ⇔ identificateur d'une variable membre
    - chaque champ a son propre type (cf. modèle)
    - chaque champ a une valeur de ce type (cf. instance)

Structures

- Déclaration d'un modèle de structure

de la vision logique...

numInsc
numInsee
nom
prenom
adresse
moyenne

...à la déclaration du modèle

```
struct etudiant {
    long numInsc;
    char numInsee[14];
    char nom[30],
    char prenom[30];
    char adresse[100];
    float moyenne;
};
/* aucune réservation en mémoire */
```

- selon la syntaxe C:

```
struct <id_type_struct> { <id_type_champ1> <id champ1>;
                          ...
                          <id_type_champn> <id champn>;
                        };
```

Structures

- Déclaration de variables structurées

struct <*id_type_struct*> <*id_variable*>;

```
/* déclaration de trois variables de type étudiant */
struct etudiant unetu;
struct etudiant premier, dernier;

/* déclaration d'une structure de données et déclaration
   de trois variables */
struct date {
    short jour;
    short mois;
    short annee;
} today, tomorrow, yesterday;
```

Structures

- Initialisation lors de la déclaration

```
/* initialisation totale */
struct etudiant unetu = { 12563,
                          "1820564513025",
                          "MUNIER",
                          "Manuel",
                          "Mont-de-Marsan",
                          19.5 };

/* initialisation partielle */
struct etudiant unautre { 524,
                          "1810933201856",
                          "GALLON",
                          "Laurent",,};
```

Structures

- Imbrication de structures

```
/* une personne en général */
struct personne {
     char numInsee[14];
     char nom[30];
     char prenom[30];
     char adresse[100];
};

/* un étudiant est une personne particulière */
struct etudiant {
     long umInsc;
     struct personne individu;
     float moyenne;
};
```

définition récurrente
INTERDITE

```
struct A { int i;
           struct A j; };
```

Structures

- Imbrication de structures

```
/* un employé est une personne particulière */
struct employe {
    long numSal;
    struct personne salarie;
    float salaire;
    struct date embauche;
};

/* déclaration d'une variable */
struct employe unemp = { 256,
                         { "2710531592048",
                           "TRONE",
                           "Paule",
                           "Toulouse" },
                         12800.0,
                         { 1, 1, 2002 }
                       };
```

Structures

- Utilisation « **globale** » d'une structure
  - Affectation globale: opérateur **=**
    - ceci n'est possible qu'entre deux variables déclarées à partir du même modèle

```
tomorrow = today;
```

    - il s'agit d'une recopie des valeurs champs par champs
  - Adresse d'une structure: opérateur **&**

```
/* déclaration d'une variable pointeur sur
   une structure du modèle date */
struct date *ptrdate;
...
ptrdate = &today;
```

Structures

- **Accès aux champs** d'une structure
  - Accès à la valeur d'un champ: opérateur .

    <id_variable> . *<id_champ>*

    - le type de cette expression est celui du champ
  - Exemples:

```
unemp.salaire = 325000;
tomorrow.jour = today.jour + 1;
strcpy(unetu.prenom, "Bill");
unemp.embauche = today;
yesterday.mois--;
scanf("%f", &unetu.moyenne);
```

Structures

- **Accès aux champs** d'une structure
  - Imbrication de structures:
    - si un champ est lui-même une structure, on peut à son tour lui appliquer l'opérateur . pour accéder à un de ses champs
  - Exemples:

```
gets(unemp.salarie.adresse);
unemp.embauche.mois = 2;
```

Structures

- Tableaux de structures
  - Déclaration

```
/* déclaration d'une var tableau de 100 employés */
struct employe tab_sal[100];
```

  - Accès aux éléments
    - au niveau global
      - `tab_sal[i] = tab_sal[j];` `/* indiciation */`
      - `*(tab_sal+i) = *(tab_sal+j);` `/* indirection */`
    - au niveau des champs
      - `strcpy(tab_sal[i].salarie.nom, "DUPONT");`
      - `strcpy(*(tab_sal+i).salarie.prenom, "Pierre");`

Structures

- Pointeurs et structures
  - Déclaration de pointeurs sur structure
    - `struct employe *ptr_emp;`
    - `struct employe *cour = tab_sal;`
  - Au niveau des adresses
    - `ptr_emp = &unemp;`
    - `cour++;`

Structures

- Pointeurs et structures
  - Déclaration de pointeurs sur structure
    - `struct employe *ptr_emp;`
    - `struct employe *cour = tab_sal;`
  - Au niveau des valeurs
    - au niveau global
      - `unemp = *cour;`
      - `*ptr_emp = tab_sal[8];`
    - au niveau des champs
      - `(*cour).salaire = 8000.0;`
      - `strcpy((*cour).salarie.nom, "DURAND");`

Structures

- Pointeurs et structures
  - Accès à la valeur d'un champ: opérateur ->

    <adr_structure> -> *<id_champ>*
  - Exemples
    - le type de l'expression est celui du champ

```
(&unemp)->salaire = 32500.0;
ptrdate->mois--;
scanf("%f",&(cour->salaire));
(ptr_emp->embauche).mois = 2;
```

Structures

- Pointeurs et structures
  - Accès à la valeur d'un champ: opérateur ->

    <adr_structure> -> *<id_champ>*

  - Équivalence de notation

```
struct date today;
struct date *ptr_today = &today;

today.jour
⇔ ptr_today->jour
⇔ (&today)->jour
⇔ (*ptr_today).jour
```

Structures

- Structures auto-référentielles
  - **Définition:** Structure dont au moins un des champs est de type **pointeur sur cette structure**. Un tel champs peut contenir **l'adresse** d'une autre variable structurée de même type…

  ➔ Structures de données dynamiques (liste, arbres, …)

```
struct noeudEtudiant {
    long numInsc;
    struct personne individu;
    float moyenne;
    char resultat;
    /* lien vers un autre étudiant */
    struct noeudEtudiant *ptr_etu;
};
```

Structures

- Structures et fonctions
  - Passer une structure en argument:
    - passage par valeur
    - passage par adresse
      - modification possible de l'argument
      - plus performant (évite création + recopie)
  - Retourner une structure en résultat:
    - déclarer le type de retour de la fonction comme étant le type structuré

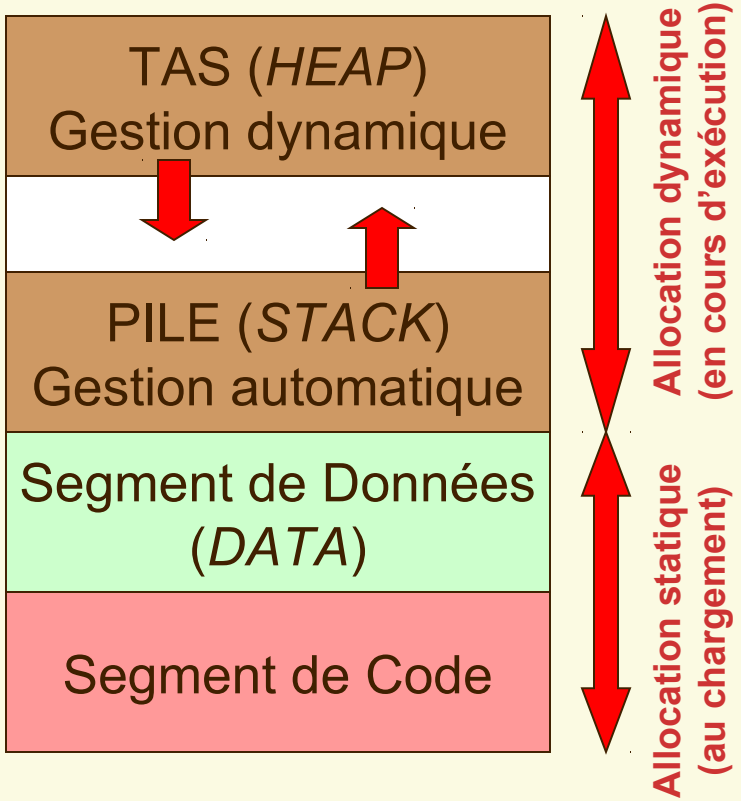
Plan

- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

Gestion Dynamique

- Chargement et exécution d'un programme

1. Chargement en MC du fichier exécutable
   - Segment de code
   - Segment de données
2. Exécution du processus
   - Utilisation dynamique de la pile
3. Utilité du tas ?
   - Gestion dynamique (et programmée) de la mémoire



Gestion Dynamique

- En résumé:

Catégorie de données	Zone mémoire	Durée de vie
statiques	données (*data*)	processus
automatiques	pile (*stack*)	exécution du bloc
dynamiques	tas (*heap*)	prévue par le programmeur

- Outils de gestion dynamique:
  - Allocation `malloc`
  - Libération `free`

Gestion Dynamique

- Allocation dynamique dans le tas

```
void *malloc(size_t taille)
```

```
/* Allocation dynamique d'une zone de 100 caractères */

...
char *nom;
...
nom = malloc(100);
puts("Entrez votre nom:");
gets(nom);
...
```

Gestion Dynamique

- Allocation dynamique dans le tas

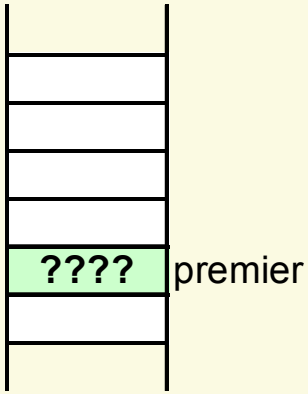
```
/* Allocation dynamique d'une zone de N réels */

...
int nb_temp;
float *temperatures, *ptr
...
puts("Entrez le nombre de températures:");
scanf("%d",&nb_temp);
temperatures = malloc(nb_temp*sizeof(float));
for (ptr=temperatures; ptr<temperatures+nb_temp; ptr++)
    {
      puts("Entrez une température:");
      scanf("%f",ptr);
    }
...
```

Gestion Dynamique

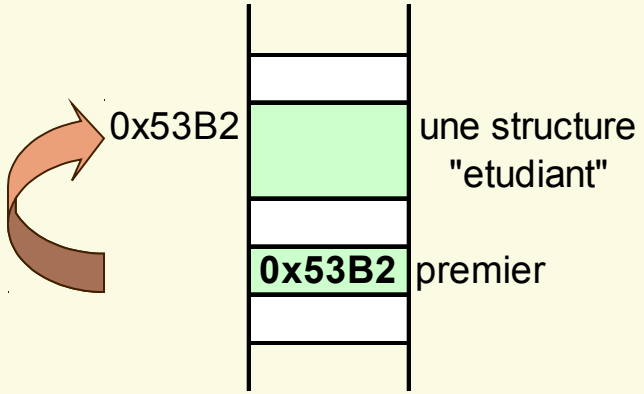
- Allocation dynamique: illustration

```
/* Allocation dynamique d'une variable structurée */
struct etudiant *premier;
...
premier = (struct etudiant)malloc(sizeof(struct etudiant));
/* Retypage de la valeur retournée (void*) par un cast */
```

Avant l'appel à malloc



Après l'appel à malloc



Gestion Dynamique

- Libération dynamique dans le tas

```
void free(void *adresse)
```

```
/* Libération des zones précédemment allouées */

...
free(nom);    /* libération des 100 caractères */
...
free(temperatures);    /* libération des N réels */
...
```

Plan

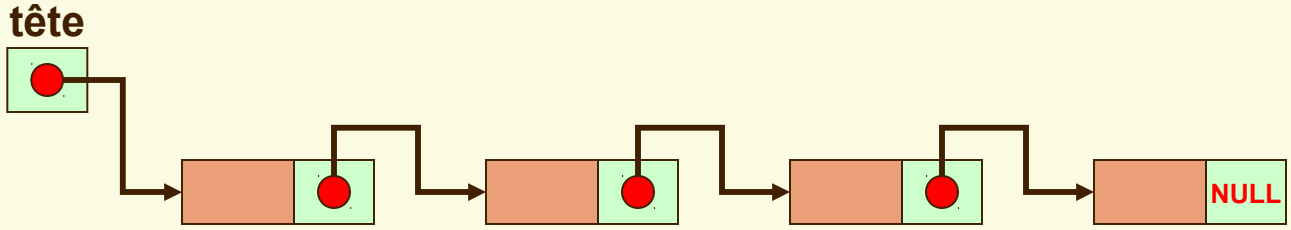
- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

S.D. Dynamiques

- Structures de données complexes
  - Relation d'organisation entre éléments de même nature
    - listes, files, piles, arbres, tas, graphes,... (tableaux ?)
- Structures de données dynamiques
  - **Structures auto-référentielles** (SDD)
  - Notion de **cellule** (ou boîte, nœud,...)
    - partie « **valeur** »
    - un (ou plusieurs) **liens** vers autre(s) cellule(s)
  - Struct. + pointeurs + gestion dyn.: **traitement**
  - **Modes de gestion**: politiques spécifiques

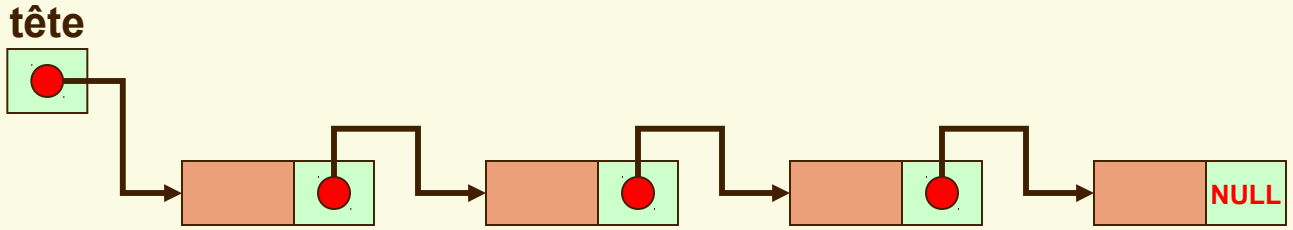
Listes

- Listes simplement chaînées
  - **Suite de cellules** liées entre elles par **un lien unique**
  - Une cellule = une **valeur** + un lien vers le **suivant**
  - Représentation logique



Listes

- Listes simplement chaînées
  - Représentation logique



  - Description en C

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* variable repérant la liste */
struct listeSC *tete;
```

Listes

- Listes simplement chaînées
  - Opérations sur les listes:
    - Fonctions à développer
    - Nouvelles classes d'algorithmes
  - Exemples:
    - initialiser
    - liste vide ?
    - insertion d'un élément
    - suppression d'un élément
    - édition des éléments de la liste
    - recherche d'un élément

Listes

- Listes simplement chaînées
  - Liste vide

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* construction d'une liste vide */
struct listeSC *listeConsVide()
{
  return NULL;
}

/* une liste est-elle vide ? */
int listeVide(struct listeSC *l)
{
  return (l == NULL);
}
```

Listes

- Listes simplement chaînées
  - Insertion d'un élément en tête de liste

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* insertion d'un élément en tête de liste */
struct listeSC *listeCons(<type_elt> elt, struct listeSC *l)
{
  struct listeSC *nouveau;
  nouveau = (struct listeSC*)malloc(sizeof(struct listeSC));

  nouveau->val = elt;
  nouveau->suivant = l;

  return nouveau;
}
```

Listes

- Listes simplement chaînées
  - Accès tête et queue d'une liste

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* valeur en tête de la liste */
<type_elt> listeTete(struct listeSC *l)
{
  return l->val;
}

/* queue de la liste, i.e. reste de la liste sans la tête */
struct listeSC *listeQueue(struct listeSC *l)
{
  return l->suivant;
}
```

Listes

- Listes simplement chaînées
  - Avec ces 5 fonctions d'accès aux listes on peut maintenant écrire des fonctions plus évoluées

```
void listeAfficher(struct listeSC *l)
{
  if (listeVide(l))
    printf("La liste est vide !!!\n");
  else
    {
      struct listeSC *temp = l; /* inutile car l passé par valeur */
      while (!listeVide(temp))
        {
          printf("...\n",listeTete(temp));      /* affichage tête */
          temp = listeQueue(temp);          /* on passe à la suite */
        }
    }
}
```

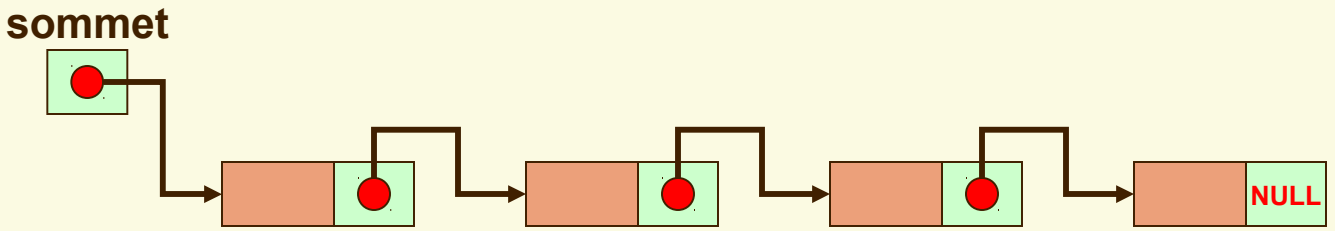
Listes

- Listes simplement chaînées
  - Utilisation de mot-clé **typedef**

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* définition d'un raccourci */
typedef struct listeSC* liste;

/* insertion d'un élément en tête de liste */
liste listeCons(<type_elt> elt, liste l)
{
  liste nouveau = (liste)malloc(sizeof(struct listeSC));
  nouveau->val = elt;
  nouveau->suivant = l;
  return nouveau;
}
```

Listes ➔ Piles

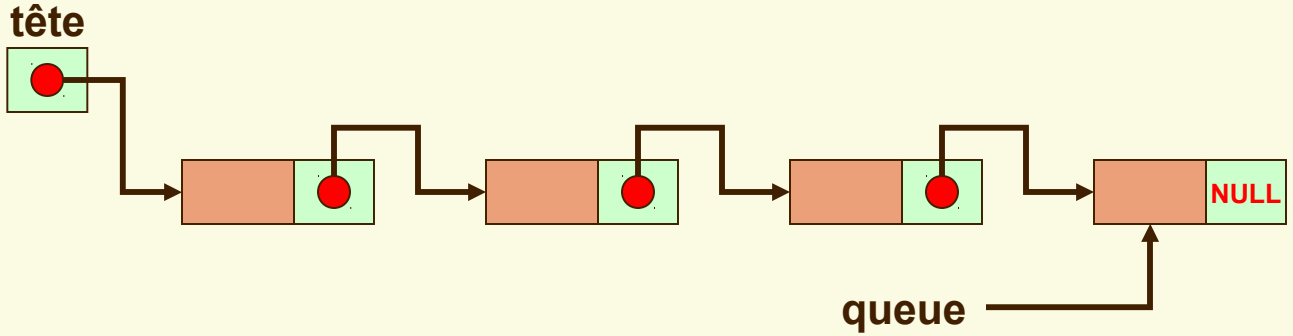
- Mode de gestion **LIFO**: pile
  - Pile d'objets ➔ **Last In First Out**
  - Représentation logique



  - Opérations particulières
    - **empiler** (insérer au sommet)
    - **dépiler** (extraire au sommet)

Listes ➔ Files

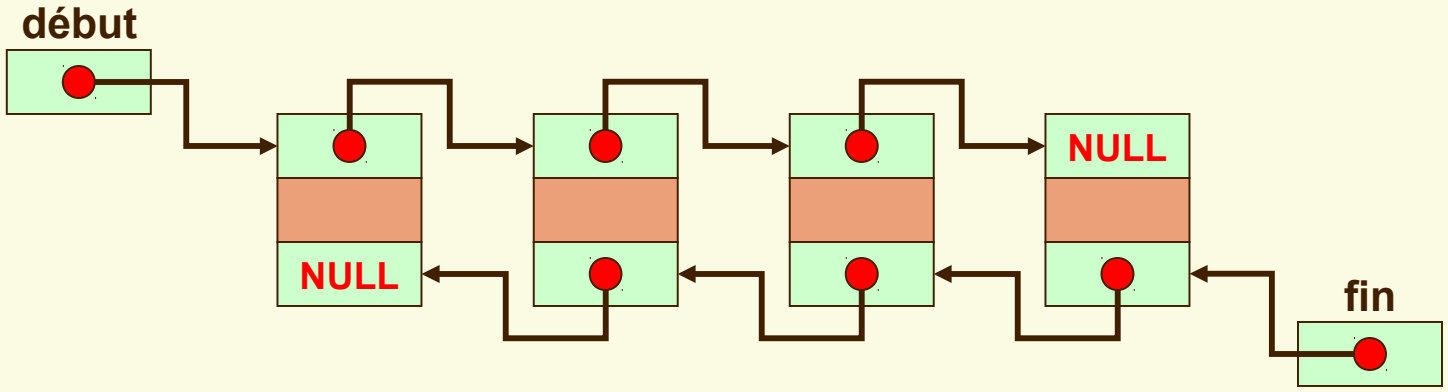
- Mode de gestion **FIFO**: file
  - File d'attente ➔ **First In First Out**
  - Représentation logique



  - Opérations particulières
    - **insérer** (toujours à la fin)
    - **extraire** (toujours au début)

Listes: pointeur arrière

- Listes doublement chaînées
  - **Suite de cellules** liées entre elles par **deux liens**
  - Une cellule = **précédent** + **valeur** + **suivant**
  - Représentation logique



Listes: pointeur arrière

- Listes doublement chaînées
  - Description en C

```
/* modèle de la structure auto-référentielle */
struct listeDC {
    struct listDC *precedent; /* lien arrière */
    <type_elt> val;
    struct listeDC *suivant; /* lien avant */
};

/* variable repérant la liste */
struct listeDC *debut, *fin;
```

Plan

- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

Idée

- Fonction récursive = fonction qui s'appelle elle-même
- Exemple: factorielle
  - Version itérative

```
int factorielle(int n)
{
  int res = 1;
  while (n>0) { res *= (n--); }
  return res;
}
```

boucle (itération)

  - Version récursive

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

appel récursif

Récursivité

- Avantages
  - Algo + clair (➔ mieux d'un point de vue GL)
  - Algo + proche de la conception (math, déf.,…)
- Inconvénients
  - Ne pas oublier la condition d'arrêt !!!
  - Conso. mémoire
    - On empile un appel de fonction à chaque fois

Définitions

- Récursivité croisée
  - Une fonction A appelle une fonction B qui appelle elle-même la fonction A
- Récursivité terminale
  - L'appel récursif a lieu à la fin du traitement
- Récursivité non terminale
  - Il y a des instructions après l'appel récursif
    - Ex: exploitation du résultat (cf. factorielle)

Récursivité multiple

- Tours de Hanoï
  - **But:** Le but du jeu est de déplacer n disques, initialement empilés sur une tour, vers une autre tour. Le nombre de tours est toujours 3. La seule action élémentaire est de déplacer un disque d'une tour vers une autre. Ce mouvement est possible s'il n'y a aucun disque sur cette tour ou si le dernier disque de la pile est plus grand que le disque que l'on veut déplacer.
  - **Analyse du problème:** Pour déplacer n disques de la tour 1 vers 3, une façon de faire est d'empiler les n – 1 plus petits sur la tour 2, de déplacer le plus grand disque de 1 vers 3, puis d'amener à leur tour les n–1 petits disques sur cette tour. Le tour est joué ! Cette analyse montre que si nous savons jouer à Hanoi avec n – 1 disques, nous savons jouer à Hanoi avec n disques.

Récursivité multiple

- Algorithme

```
// Algorithme déplaçant d'une pile de n disques
// depuis la tour origine vers la tour destination

Algorithme hanoï
Entrées n: un entier, origine: un entier, destination: un entier
Sortie
    Si n = 1
    Alors // Condition d'arrêt : un seul disque
        déplacer(origine, destination)
    Sinon // Récursivité
        autre := 6-origine-destination // indice de la 3ème tour
        hanoï(n-1, origine, autre)
        déplacer(origine, destination)
        hanoï(n-1, autre, destination)
    Fin Si

Fin hanoï
```

Dérécursiver

- Récursivité ➔ itérations
  - Il est toujours possible de dérécursiver un algo, i.e. remplacer les appels récursifs par des boucles
  - Comment ?
    - Appliquer des «procédés» selon le type de récursivité (terminale ou non, etc…)
  - Pourquoi ?
    - Limiter la consommation mémoire
    - Améliorer les performances

Listes

- Algo itératif
  - Nous savons parcourir une liste à l'aide d'une boucle…

```
void listeAfficher(struct listeSC *l)
{
  if (listeVide(l))
    printf("La liste est vide !!!\n");
  else
    {
      struct listeSC *temp = l; /* inutile car l passé par valeur */
      while (!listeVide(temp))
        {
          printf("...\n",listeTete(temp));      /* affichage tête */
          temp = listeQueue(temp);          /* on passe à la suite */
        }
    }
}
```

Listes

- Algo récursif
  - …mais, bien évidemment, les structures de données **récursives** telles que les listes se prêtent à merveille aux algorithmes **récursifs** !

```
void listeAfficher(struct listeSC *l)
{
  if (!listeVide(l))
    printf("...\n",listeTete(l));    /* affichage de la tête  */
    listeAfficher(listeQueue(l));    /* affichage de la queue */
}
```

➔ **Attention à la condition d'arrêt de la récursivité !!!**

De toute façon…

- Avec les arbres et les graphes, vos algos seront forcément récursifs
- Ce qui compte avant tout, c'est la clarté de vos algos (aspect GL)
  - Compréhension
  - Réutilisation
  - Évolution
  - Validation
- Performances & co, c'est du ressort de l'optimisation de code…

Plan

IC2

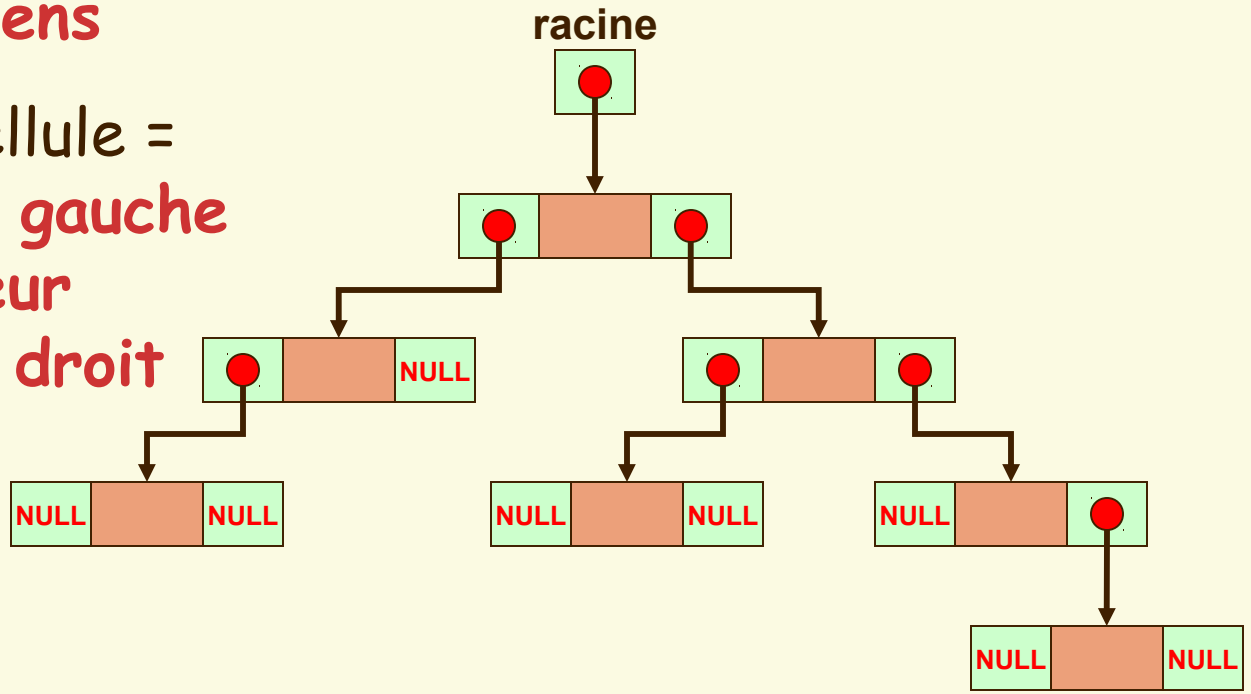
- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres

Principe

- Arbre
  - Un nœud contient plusieurs liens vers d'autres nœuds
  - Les branches sont disjointes (*sinon → graphe acyclique*)
  - Il n'y a pas de cycle (*sinon → graphe cyclique*)
- Terminologie
  - **Arbre binaire**: un nœud a au maximum 2 fils
  - **Arbre n-aire** : nombre quelconque de fils
  - **Racine** : 1er nœud (pas de père)
  - **Feuille** : nœud sans fils

Arbres binaires

- Arbres binaires
  - **Hiérarchie de cellules** liées entre elles par **deux liens**
  - Une cellule =
    fils gauche
    + valeur
    + fils droit



Arbres binaires

- Arbres binaires
  - Description en C

```
/* modèle de la structure auto-référentielle */
struct arbre {
    <type_elt> val;
    struct arbre *gauche; /* lien fils gauche */
    struct arbre *droit;  /* lien fils droit */
};

/* variable repérant l'arbre */
struct arbre *racine;
```

Arbres binaires

- Arbres binaires
  - Fonctions d'accès élémentaires
    - `struct arbre *arbreConsVide()`
    - `int arbreVide(struct arbre *a)`
    - `struct arbre *arbreCons( <type_elt> racine, struct arbre *fgauche, struct arbre *fdroit)`
    - `<type_elt> arbreRacine(struct arbre *a)`
    - `struct arbre *arbreFilsGauche(struct arbre *a)`
    - `struct arbre *arbreFilsDroit(struct arbre *a)`

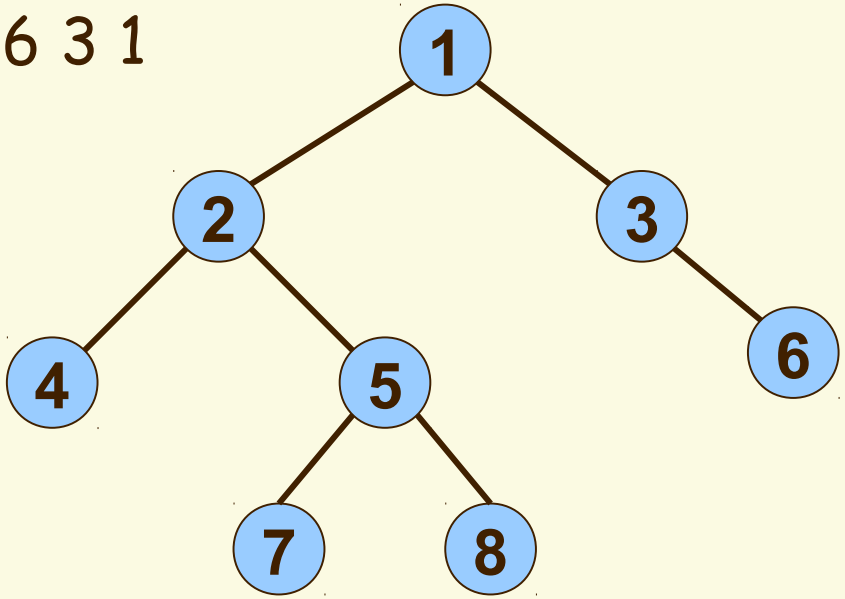
Parcourir un arbre

- Question
  - Comment parcourir (i.e. visiter) tous les nœuds d'un arbre ?
- 2 solutions:
  - Parcours en profondeur (d'abord)
  - Parcours en largeur

Parcours en profondeur

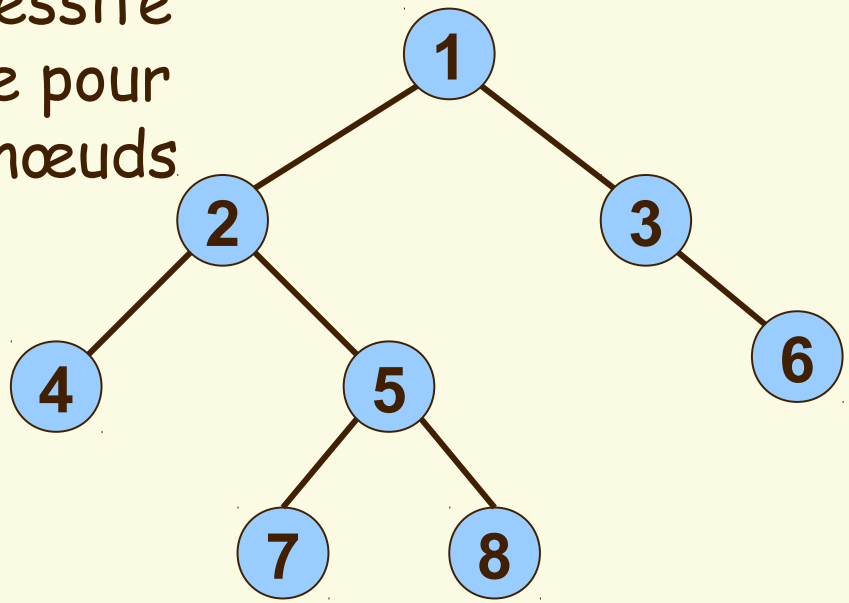
- 3 variantes:
  - G-N-D ➔ 4 2 7 5 8 1 3 6
  - N-G-D ➔ 1 2 4 5 7 8 3 6
  - G-D-N ➔ 4 7 8 5 2 6 3 1

G ≡ Gauche
D ≡ Droite
N ≡ Nœud



Parcours en largeur

- "Niveau par niveau"
  - ➔1 2 3 4 5 6 7 8
    - Techniquement, nécessite l'utilisation d'une file pour mémoriser tous les nœuds à parcourir



Algo de recherche

- Conditions
  - Arbre binaire
  - Arbre trié
    - Pour un nœud, toutes les valeurs contenues dans les nœuds du fils gauche sont plus petites que la valeur du nœud. Toutes celles du fils droit sont plus grandes.
- Principe (~dichotomie):
  - Discrimination par rapport à la valeur du nœud
    - Si la valeur que l'on recherche (clé) est plus petite que la valeur du nœud, on poursuit la recherche (récursivement) dans le fils gauche.
    - Sinon on poursuit la recherche (récursivement) dans le fils droit.
    - Optimisation: si la valeur recherchée est égale à celle du nœud, on stoppe la recherche

Optimisation

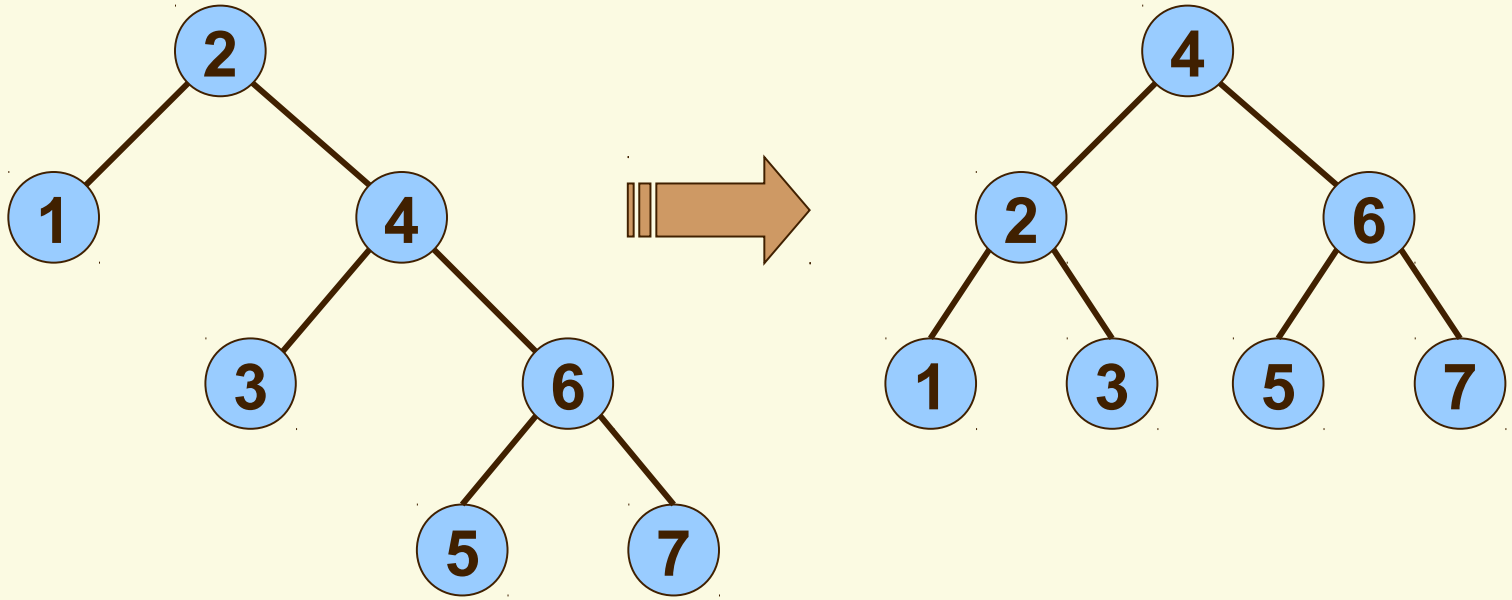
- Définition
  - Hauteur d'un arbre = nombre de niveaux
  - Arbre équilibré: pour chacun des nœuds, **hauteur(fils gauche) = hauteur(fils droit) +/- 1**
- La recherche sera d'autant plus efficace que l'arbre (binaire) sera équilibré
  - ➔Algo en **$\log_2$(nombre nœuds)**

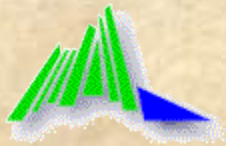
Ex: 1 024 valeurs ➔ 10 tests maxi
1 048 576 valeurs ➔ 20 tests maxi
1 073 741 824 valeurs ➔ 30 tests maxi

NB: population française au 01/01/2005 = 60 561 200 → 26 tests

Optimisation

- Arbre équilibré (ou bien balancé)
  - Algo d'équilibrage…
    - *Remarque: parcours en profondeur GND identique !*





Compléments

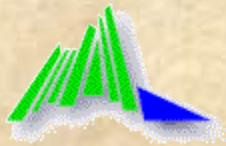
Graphes

Graphes

- Comme les arbres, mais, éventuellement
  - Branches non disjointes
  - Cycles dans les liens

➔ Algos de parcours d'arbres non adaptés

➔ Parcours (en largeur ou en profondeur) par **marquage** des nœuds pour éviter:
  - De traiter **plusieurs fois** le même nœud (branches non disjointes)
  - De partir en **boucle infinie** (cycles)



Compléments

Notions de Complexité

Complexité

- Qu'un algorithme parvienne au bon résultat est une chose, encore faut-il qu'il le fasse dans un temps raisonnable. Un algorithme peut ainsi demander beaucoup de ressources (mémoires, temps, espace disque...) pour parvenir à un résultat, tandis qu'un autre mieux conçu le ferait de manière plus efficace.
- Il faut dès lors choisir l'algorithme le plus approprié face à une situation : un code pour une calculatrice à énergie solaire disposera certainement de moins de ressources qu'un autre pour vérifier le bon déroulement du décollage de la fusée Ariane.
- Il s'agit donc d'évaluer les besoins en ressources d'un ou plusieurs algorithmes, afin de les rattacher à une courbe connue ou à les comparer entre eux. Étant donnés les prix en chute de la mémoire et de l'espace disque, la complexité la plus pertinente reste celle en temps. Par ailleurs, il est fréquent de devoir sacrifier l'efficacité en stockage pour privilégier le temps d'exécution.

Complexité

- On recense trois types de complexité :
  - Complexité "dans le pire des cas" : situations où les données obligent l'algorithme à un maximum d'itérations
  - Complexité "dans le meilleur des cas" : situation où les données autorisent l'algorithme à les traiter en un minimum d'itérations
  - Complexité moyenne : moyenne de toutes les situations possibles
- Chaque cas d'utilisation devra être étudié selon les données à traiter. Si l'on entre dans un cadre plus général, c'est la complexité moyenne qui sera à prendre en compte. L'objectif est bien évidemment que ce calcul de complexité se fasse indépendamment du processeur, du type de mémoire, etc…
- La comparaison de l'efficacité des algorithmes se fait grâce à la notation "grand *O*" ("*O*" signifiant "de l'ordre de") ou notation de Landon, qui décrit le comportement asymptotique des fonctions mathématiques, et par extension permet d'indiquer la rapidité avec laquelle elle (ou sa courbe descriptive) augmente ou diminue. Il existe également une notation "petit o".

Complexité

- En effet, les algorithmes ne peuvent décemment être testés sur 10 ou 100 possibilités, mais parfois dans des quantités astronomiques (si l'algorithme est crucial). Les algorithmes connus sont donc tous rattachés à l'une des notations grand O, qui permet de savoir de quelle manière évolue une courbe temps/possibilités. Cela permet de repérer d'un seul coup d'œil les algorithmes efficaces pour un très petit échantillon, ceux qui prennent constamment le même temps...
- Cette évolution des courbes peut suivre divers cadres connus :
  - x = y, quand l'augmentation est constante et quasiment rectiligne
  - x = log(y), quand elle correspond à celle du logarithme
- Ainsi, on dénombre un certain nombre de types de notations grand O :
  - O(1) : le traitement prend toujours le même temps
  - O(n) : le temps de traitement est constamment proportionnel au nombre de données, quasiment linéaire
  - O(n!) : le temps de traitement correspond à une factorielle du nombre de données
  - O(log(n)) : évolution logarithmique
  - O(n*log(n)) : évolution linéarithmique
  - O(c^n) : évolution exponentielle

Complexité

- Ainsi, en rapprochant son algorithme d'une des notations grand O, on est mesure de comparer directement son amplitude avec celle d'un autre algorithme, et donc de voir lequel est le plus approprié pour le travail à accomplir, ou le cadre dans lequel il se réalise.

Plan

- Rappels de C
  - Pointeurs
  - Structures de données
  - Gestion dynamique de la mémoire
- Structures de données dynamiques
  - Listes, files, piles
  - Récursivité
  - Arbres