Cours “Algorithmique et Structure des Données”

Licence L2 Informatique, semestre S3

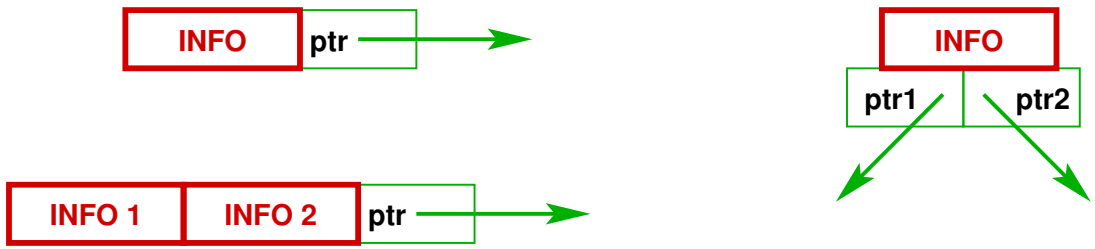
Année 2019-2022

Notion de cellule

Une **cellule** est une structure de donnée constituée de deux sortes de composants :

- un (ou plusieurs) élément(s) d'**information**
- un (ou plusieurs) **pointeur(s)** sur une (ou plusieurs) autres cellules

Exemples de cellules :



Structure de donnée "cellule"

Une cellule est donc une **donnée structurée**, puisqu'elle a des champs (au moins deux) de natures différentes.

Le champ "pointeur" d'une cellule pointe sur une cellule de même type $\Longrightarrow$ la définition du type de donnée est récursive (i.e. le type fait appel à lui-même) et nécessite de donner un nom à la structure :

```
typedef struct Celu cellule;
struct Celu
{   int info;        // si l'information est de type Entier
    cellule *ptr;    // pointeur vers un objet de même type
};
```

Déclaration et utilisation d'une cellule

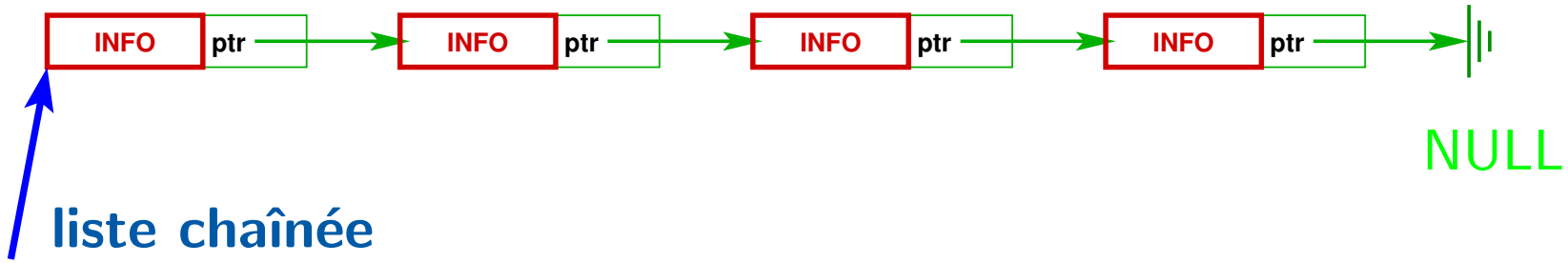
On accède à une cellule par un pointeur_sur_cellule ; a priori, si celui-ci n'est **que déclaré**, c'est un pointeur sur NULL.

La déclaration d'une cellule doit être accompagnée d'une allocation mémoire pour que l'on puisse travailler avec la cellule :

```
void main()
{
    cellule *cell = malloc(sizeof(cellule));
    if(cell != NULL)      // pour vérifier si pas eu de pb d'allocation
    { cell->info = 0;
      cell->ptr = NULL;
    }
    ecrit(cell);          // proc. qui fait : printf("info = %d\n",c->info);
    modifie(45,cell);     // proc. qui fait : c->info=m;
    ecrit(cell);
}
```

Définition d'une liste chaînée

Une liste chaînée est une succession de cellules ayant un seul pointeur :



On accède à la liste par un pointeur vers sa **première cellule** (en bleu sur la figure) et uniquement par cette *entrée*. Pour atteindre toutes les autres cellules, il faudra se déplacer **séquentiellement**, de pointeur en pointeur.

La dernière cellule de la liste pointe vers NULL (symbole "prise de terre").

Structure pour contrôler une liste chaînée

Pour contrôler une liste chaînée, on définit généralement une structure liste contenant, au moins, un pointeur sur la **tête** de liste, ainsi que d'autres informations éventuelles :

```
typedef struct Liste list_chn;
struct Liste
{
    cellule *tete;          // pointeur vers première cellule
    // facultatif :   cellule *queue;    // derniere cellule
    // facultatif :   int nbCell;        // nombre de cellules
};
```

Déclaration et initialisation

Pour déclarer et initialiser une variable de type liste chaînée, on crée **d'abord une cellule** que l'on initialise, puis on fait pointer la tête de liste sur cette cellule :

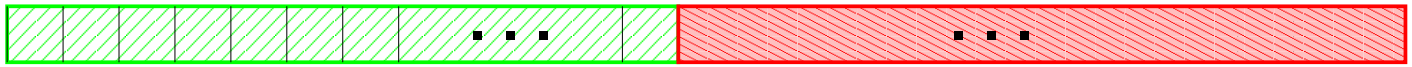
```
list_chn *liste = malloc(sizeof(list_chn));    // déclaration

cellule *cell = malloc(sizeof(cellule));
if (liste != NULL && cell != NULL)    // vérif. si pas pb
alloc.
{    cell->info = 0;
     cell->ptr = NULL;
}

liste->tete = cell;                          // initialisation
```

Tableaux vs listes chaînées

N.B. Les différentes cellules d'une liste chaînée n'ont aucune raison d'occuper des espaces mémoires contigus ; c'est donc une structure plus souple que les tableaux dynamiques.

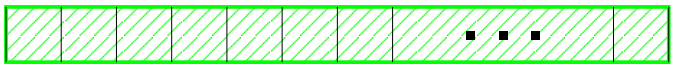
Pour récapituler, on a maintenant trois types structurés différents :

- Les tableaux statiques :

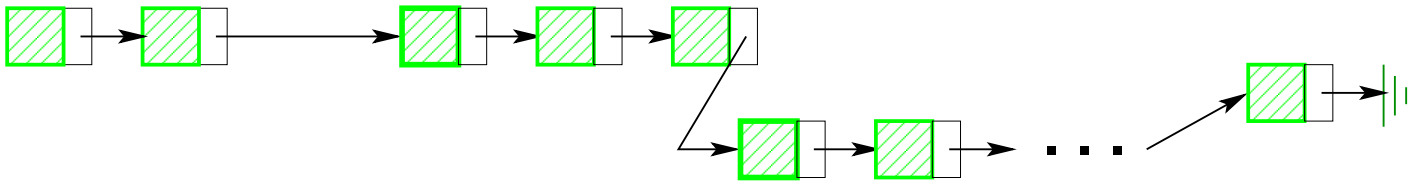


gaspillage d'espace memoire

- Les tableaux dynamiques :



- Les listes chaînées :



Afficher tous les éléments d'une liste

[FIGURE]

```
void affiche(list_chn *ll)
{
  cellule *pp = ll->tete;
  while (pp != NULL)   // parcours de la liste
  {
    ecrit(pp);  // appelle la procédure de lecture de cellule
    pp = pp->ptr;
  }
}
```

Insertion en tête de liste

[FIGURE]

```
void insere_tete(int val, lis_chn *ll)
{
  cellule *cc = malloc(sizeof(cellule));
  if (cc != NULL)     // création de la cellule à insérer
  {
    cc->info = val;
    cc->ptr = ll->tete;
  }
  ll->tete = cc;
}
```

Insertion en fin de liste

[FIGURE]

```
void insere_queue(int val, list_chn *ll)
{
  cellule *cc = malloc(sizeof(cellule));
  if (cc != NULL)     // création de la cellule à insérer
  {
    cc->info = val;
    cc->ptr = NULL;
  }
  cellule *pp = ll->tete;
  while (pp->ptr != NULL)   // parcours de la liste
  {
    pp = pp->ptr;
  }
  pp->ptr = cc;
}
```

Rechercher un élément dans une liste

[FIGURE]

```
void recherche(int xx, list_chn *ll)
{
  int present = 0;   // variable booléenne
  cellule *pp = ll->tete;
  while ((!present) && (pp != NULL))   // parcours de la liste
  {
    if (pp->info == xx)
      present = 1;
    pp = pp->ptr;
    }
  if (present)        // affichage du diagnostic
    printf("%d est bien la !\n", xx);
  else
    printf("%d n'est pas dans la liste...\n", xx);
}
```

Supprimer un élément d'une liste

[FIGURE]

WARNING : il ne faut pas perdre les wagons de queue !
Pour cela, il faut **anticiper** la lecture de la valeur à supprimer.

[FIGURE]

Supprimer un élément d'une liste

```
void supprime(int xx, list_chn *ll)
{ int fait = 0;   // variable booléenne
  if (ll->tete != NULL)
  { cellule *pp = ll->tete;   // si l'élément est en tête de la liste
    if (pp->info == xx)
    { ll->tete = pp->ptr;
      fait = 1;
    }
    else     // parcours de la liste si l'élément est n'importe où ailleurs
    { while ((!fait) && (pp->ptr != NULL))   // prévoir le cas ''élément
absent''
      { if ((pp->ptr)->info == xx)
        { pp->ptr = (pp->ptr)->ptr;
          fait = 1;
        }
      pp= pp->ptr;
      }
    }
  }
  if (!fait)
    printf("%d n'a pu être supprimé car il n'était pas dans la liste.\n",xx);
}
```

Listes avec pointeur de queue

Comme on vient de le voir, on peut faire toutes les opérations sur les listes chaînées avec un seul pointeur de tête.

Si l'on ajoute un pointeur de queue, cela peut simplifier certaines opérations (exemple : lecture ou suppression du dernier élément), mais aussi en compliquer certaines autres !

```
typedef struct Liste list_chn;
struct Liste
{
    cellule *tete;        // pointeur vers première cellule
    cellule *queue;       // pointeur vers derniere cellule
    int nbCell;           // compteur du nombre de cellules
};
```

On peut aussi ajouter un entier pour comptabiliser le **nombre de cellules**
$\Rightarrow$ ne pas oublier la mise à jour dans les opérations d'insertion ou de suppression.

Listes doublement chaînées

Une liste doublement chaînée est constituée de cellules ayant **deux pointeurs** : l'un vers la cellule suivante (comme dans les listes simples) et l'autre vers la cellule précédente.


```
typedef struct Celu cellule;
struct Celu
{   int info;          // si l'information est de type Entier
    cellule *ptr_suiv;   // pointeur vers la cellule suivante
    cellule *ptr_prec;   // pointeur vers la cellule précédente
};
```

Certains opérations sont plus faciles, mais, en moyenne, la programmation est plus compliquée car il faut gérer correctement les deux pointeurs.

Listes circulaires

L'inconvénient des listes doublement chaînées est la présence de deux pointeurs sur NULL :

- en queue de liste, pour `ptr_suiv` (comme pour les listes simples)
- et en tête de liste, pour `ptr_prec`

Une liste circulaire fait disparaître cet inconvénient en "recollant les deux bouts" : plus de NULL nulle part !

MAIS : par où est-ce qu'on accède à une liste circulaire ? ? ?

Racine d'une liste circulaire

Une cellule racine va jouer le rôle de *point à l'infini* (comme dans les espaces projectifs !) et servir de **point d'entrée** vers la liste circulaire. Les pointeurs `ptr_suiv` de queue et `ptr_prec` de tête vont tous deux pointer vers la racine. N.B. celle-ci ne contient pas d'information.

Une liste circulaire **vide** sera constituée d'une cellule racine rebouclée sur elle-même :

cellule "racine"

Cette astuce va très nettement simplifier l'ecriture des opérations, surtout celles d'insertion et de suppression, dans les listes chaînées.

Exemple d'opérations simplifiées

Suppression d'un élément quelconque

```
void supprimerElement (Liste_Circulaire_Doublement_Chainee* element)
{
  element->prec->suiv = element->suiv;
  element->suiv->prec = element->prec;
  free(element);      // on libère la mémoire allouée
}
```

Suppression en tête

```
void supprimerPremierElement (Liste_Circulaire_Doublement_Chainee* racine)
{
  if (racine->suiv != racine)
    supprimerElement (racine->suiv);
}
```

Suppression en queue

```
void supprimerDernierElement (Liste_Circulaire_Doublement_Chainee* racine)
{
  if (racine->prec != racine)
    supprimerElement (racine->prec);
}
```

Une pile : LIFO

Une **pile** est une liste d'objets auxquels on ne peut accéder **que par le dessus** : i.e. dans l'ordre **LIFO** = Last In First Out.

Penser à une pile d'assiettes . . .

Une pile peut être implémentée à l'aide d'une *liste chaînée :*

```
typedef struct pi pile
struct pi
{
  int info;
  struct pile *prec;
}
```

Opérations sur les piles

Il est pratique de considérer une **pile** comme un élément "sommet" qui surmonte une **pile** (constituée de tous les éléments qui sont en dessous).

Outre le fait de tester si une pile est vide ou non, les opérations sont :

- empiler un nouvel élément au sommet de la pile
- dépiler l'élément qui était au sommet de la pile
- afficher tous les éléments : le sommet, puis le reste de la pile...

Ce qui incite à écrire une procédure d'affichage **récursive**.

Une file : FIFO

Une **file** est une liste d'objets dans laquelle on entre en se plaçant à la queue et on ressort quand on arrive en tête de la file : i.e. dans l'ordre **FIFO** = First In First Out.

Penser à une file d'attente devant un guichet !

Une file peut être implémentée à l'aide d'une *liste chaînée :*

```
typedef struct ff file
struct ff
{
  int info;
  struct file *suiv;
}
```

Opérations sur les files

Deux éléments sont importants dans une file : le premier et le dernier.

Outre le fait de tester si une file est vide ou non, les opérations sont :

- ajouter un nouvel élément en queue, donc le dernier
- retirer l'élément de tête de la file, donc le premier
- afficher tous les éléments de la file : dans un ordre, ou dans l'autre.

Notion d'arbre

Un arbre peut être vu comme un ensemble de cellules ayant chacune **une** information et **plusieurs** pointeurs, chacun pointant sur une autre cellule.

Les arbres s'utilisent dans de multiples contextes *(voir illustrations)* :

- arbre syntaxique
- expression arithmétique
- arbre lexicographique
- arbre phylogénétique des espèces
- arborescence des fichiers gérés par un système d'exploitation
- hiérarchie des rubriques d'un site web
- arbre de jeu : le déplacement des pièces sur un échiquier
- arbre implicite dans une recherche dichotomique, un "tri rapide", etc.

Le vocabulaires des arbres

Un arbre informatique utilise à la fois le vocabulaire des arbres botaniques (racine, feuilles) et celui des arbres généalogiques (père, fils, frère)

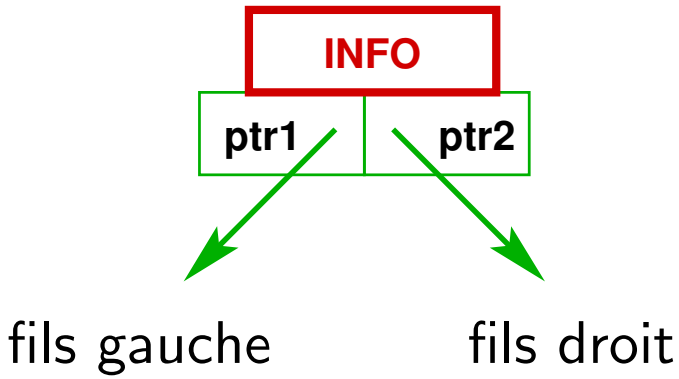
Les cellules qui constituent l'arbre s'appellent des nœuds.
En haut (!) de l'arbre, il y a un nœud unique : la racine.
Le nœud racine a plusieurs fils, chacun ayant lui-même plusieurs fils, etc. . . ou pas de fils du tout : il s'agit alors d'une feuille, tout en bas (!) de l'arbre.

La **taille** de l'arbre est le nombre des nœuds qui le constituent.
La **profondeur** de l'arbre est la longueur d'un chemin allant de la racine vers la feuille la plus basse.

Arbre binaire vs arbre quelconque

A priori, les nœuds d'un arbre quelconque n'ont pas tous les mêmes nombres de fils. Pour programmer un arbre quelconque, on peut implémenter chaque nœud comme une cellule constituée : d'une **information** et d'une **liste de pointeurs** (liste chaînée, par exemple).

Un cas particulier, très souvent utile, est celui de l'arbre binaire : chaque nœud a **exactement deux fils**, un *fils droit* et un *fils gauche*.



Parcourir un arbre binaire

On peut parcourir un arbre de deux manières : en largeur ou en profondeur.

En largeur : on commence par la racine (de profondeur 0), puis on inspecte tous ses nœuds fils, qui sont à la profondeur 1, puis les fils des fils (profondeur 2), et ainsi de suite . . .

En profondeur : on descend de la racine vers une feuille, puis on remonte à la racine, plusieurs fois, afin de passer par tous les nœuds de l'arbre. On distingue trois manières de faire cela :

- parcours préfixe : inspection, fils gauche, fils droit
- parcours infixe ou symétrique : fils gauche, inspection, fils droit
- parcours suffixe : fils gauche, fils droit, inspection

N.B. On programme cela de manière récursive.

Arbre binaire de recherche

On peut insérer un nouvel élément dans un arbre en cherchant la place qui lui convient pour respecter un ordre donné (ex. ranger des nombres par ordre croissant).

On dit alors que l'on construit un arbre binaire de recherche, parfois appelé ABR.

Question :
que donne alors le parcours infixe ?

```
Arbre_Insérer(Arbre T, noeud z)
        y:=NIL
        x:=racine[T]

        TantQue x distinct de NIL faire
                y:=x
                Si clef[z]<clef[x]
                alors x:=fils_gauche[x]
                sinon x:=fils_droit[x]
                FinSi
        FinTantQue

        père[z]:=y
        Si y=NIL
        alors racine[T]:=z
        sinon
                Si clef[z]<clef[y]
                alors fils_gauche[y]:=z
                sinon fils_droit[y]:=z
                FinSi
         FinSi
```