Chapitre 3 : Listes, Piles, Files et récursivité

Dans ce chapitre, nous allons décrire trois grandes familles de structures de données. Leur structure même suggère un usage linéaire, arborescent ou sous forme de graphe. L'analyse et la conception d'un algorithme pour un problème sont indépendantes de l'implantation : la représentation des données ne sera donc pas fixée dans un premier temps.

Nous étudierons donc successivement les structures de données classées linéairement, puis les structures arborescentes, pour lesquelles les données sont plus finement hiérarchisées, et enfin les structures relationnelles (graphes).

Structures séquentielles : Listes linéaires, Piles, Files
Structures arborescentes : Arbres binaires, planaires
Structures relationnelles : Graphes

1. Les Listes

Les listes linéaires sont une forme très commune d'organisation des données. Les données doivent être traitées séquentiellement, et on peut généralement ajouter ou retirer des données d'une liste, selon des règles strictes.

Il existe 3 sortes de listes, en fonction, justement des techniques d'ajout et de suppression :

- Piles :
    - les données ne peuvent être ajoutées ou supprimées qu'à une même extrémité (sommet).
    - ce sont des structures très simples implémentées directement par le système et utilisées pour son fonctionnement, avec une occupation en place et un temps d'exécution optimisés.
    - elles sont nommées LIFO (Last In, First Out).

Exemple : une pile d'assiette.

- Files :
    - les données ne peuvent être ajoutées qu'à une extrémité (queue), lues et supprimées à l'autre (tête).
    - cela correspond à la réalité physique du fonctionnement des réseaux et de certaines couches basses.
    - elles sont nommées FIFO (First In, First Out).

Exemple : la file d'attente à la boulangerie.

- Listes linéaires, itératives ou récursives :
    - les données peuvent être ajoutées ou supprimées n'importe où dans la liste, ce qui se fait avec plus ou moins de facilité et à l'aide de primitives *ad hoc*.
    - c'est une structure de haut niveau, implémentées souvent *via* du logiciel. Dans certains langages, il s'agit de briques de base.

Exemple : une bibliothèque où l'on peut prendre/ranger les livres où l'on veut (éventuellement en respectant un classement, mais pas nécessairement).

a. Listes : définition générale

Une liste L = <e_1, …, e_n> est une suite d'éléments repérés *a posteriori* suivant leur place (rang) dans cette liste.

On peut donner également une définition récursive : une liste non vide est formée d'une première place suivie d'une autre liste. Cette définition récursive peut s'écrire :

Liste = Vide
Liste = <tête, Liste>

La première place est dite la tête de L et notée tête(L).
Le repérage part de la tête et n'a aucun lien avec les valeurs des éléments.
L'ensemble des places est ordonné *via* une fonction *successeur* (succ(element))qui, appliquée à toute place (sauf la dernière) fournit la place suivante. On pourrait également numéroter les places pour les repérer, mais ce n'est pas nécessaire : il est suffisant que chaque élément connaisse son successeur.
Le nombre total N d'éléments, et par conséquent de places, est appelé longueur de la liste :

- si N = 0 la liste est vide,
- sinon la fonction successeur est définie de la première à la (N-1)^{ième} place de la liste

Comment trouver/identifier les éléments :

- succ(tête(£)), le successeur de tête(£), est la seconde place,
- succ^2(tête(£)) est la troisième place…
- succ^N(tête(£)) n'est pas défini (après la dernière place).

Exemple :

Ma_Liste = <Térébenthine, Firmin, Gudule, Philéas>

Il y a quatre places.
Les trois premières contiennent les éléments Térébenthine, Firmin, Gudule, et un moyen d'accéder à la place suivante.
La quatrième contient l'élément Philéas et un signal de fin.
La liste est donnée par sa tête, qui contient Térébenthine, et le moyen d'accéder à ce qu'il y a après.

b. Représentation des listes

Suivant les familles de langages, les représentations des listes sont extrêmement variables, il faut toutefois avoir en tête que le type d'implémentation sous-jacent n'influence pas le comportement choisi pour la liste (en clair, si l'on a représenté une pile *via* une liste chaînée ou un tableau, on se contraindra à n'accéder qu'au premier élément).
On peut distinguer deux modes de représentation : contiguë ou chaînée.

- **Représentation contiguë des listes**

Cette représentation est présentée ici « pour mémoire » mais n'est pas utilisée dans ce cours.
La liste est représentée par un tableau où les cases jouent le rôle des places : la k^{ième} case est la k^{ième} place de la liste.
On peut imaginer un bloc de places mémoires physiques contiguës réservées pour le tableau : ce sont des structures de taille fixe, la place est réservée à l'avance, quelle que soit l'occupation ou l'utilisation qui en est faite.

Si T est un tel tableau, T[k] désigne la k^{ième} place de la liste et aussi le contenu de cette place.
L'opération successeur est représentée par l'incrémentation de +1 sur le numéro des places. Avec

une telle représentation, l'accès et le parcours séquentiel sont simples, la suppression et l'insertion le sont moins puisqu'il faut déplacer les cases et leur contenu avant ou après la case considérée pour ne pas créer de « trous ».

Exemple : pour la liste de l'exemple précédant, on obtient donc la représentation suivante.

Ma_Liste de type Liste_contiguë

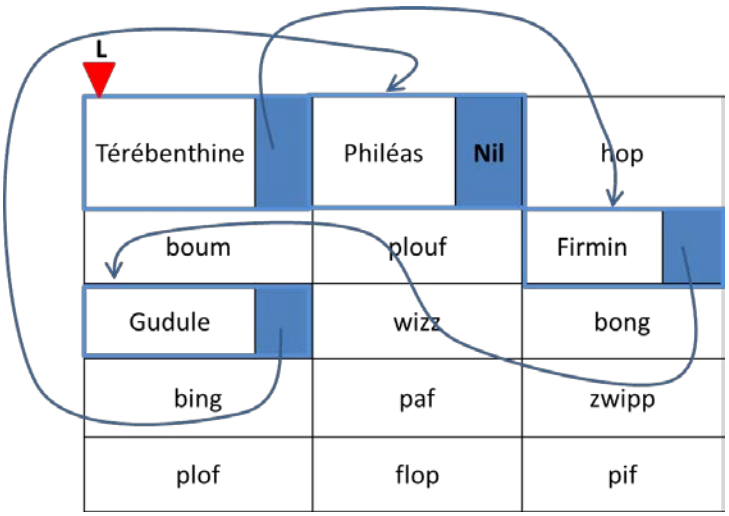
	1	2	3	4
Ma_Liste	Térébenthine	Firmin	Gudule	Philéas

Ma_Liste[1] = Térébenthine, etc…

- **Représentation chaînée des listes**

Il s'agit de la représentation que nous utiliserons la plupart du temps.
La représentation chaînée est une bonne façon de traduire les notions de tête et de successeur.
Le fonctionnement suggère que les places sont localisées à divers endroits de la mémoire repérés par leurs adresses en mémoire.
On connaît seulement une « poignée » donnant accès au premier élément (adresse de la cellule en tête de liste). Chaque cellule connaît ensuite l'adresse de son successeur.
La déclaration de telles listes traduit une structure récursive. Il s'agit de récursivité dite croisée : la définition s'utilise elle-même, de façon explicite

Liste : lien (pointeur) vers une cellule
Cellule : contient une valeur et un autre lien (pointeur)



Si L est une liste non vide, cel(L) permet d'accéder à la cellule référencée par L.
Si c est une cellule, val(c) est son élément, suiv(c) est la Liste vers laquelle elle pointe.
Pour une liste L, val(cel(L)), également notée L.val, est son premier élément, suiv(cel(L)), noté aussi L.suiv, est la liste qui reste après ce premier élément.

NB : certains langages de programmation shuntent la notion de place et de cellule et accèdent directement aux champs.

Exemple : pour la liste mentionnée plus haut, Ma_Liste = <Térébenthine, Firmin, Gudule, Philéas>

cel(Ma_Liste) contient Térébenthine et l'accès à la suite ; val(cel(Ma_liste)) est également Térébenthine, suiv(cel(Ma_liste)) est la liste<Firmin, Gudule, Philéas>.

On écrit aussi Ma_Liste.val = Térébenthine et Ma_Liste.suiv = <Firmin, Gudule, Philéas>.
Pour pouvoir prendre en compte l'ajout et la suppression d'éléments, on ajoute des opérateurs de mutation qui permettent de rendre compte des quatre instructions pour un lien (pointeur) π.

- création : créer un nouvel objet et l'attacher à π
- rattachement : attacher π à un objet existant
- oubli : rendre π vide
- dé-référençage : accéder à la valeur de l'objet attaché à π

Lors de l'usage des listes, nous supposerons donc que le langage utilisé nous donne un moyen de créer et de modifier un lien (pointeur) vers une cellule, c'est-à-dire que sont définies les instructions suivantes :

Créer(L)	définit un nouveau lien (pointeur)
Allouer(Cel(L)) ou L ← new(Liste)	alloue de la place en mémoire (crée un nouveau lien)
L ← Null	définit une liste vide
L' ← L	assigne à L' la même adresse qu'à L (définit 2 listes pointant la même cellule)
Cel (L)	permet d'accéder à la cellule référencée par L

c. Listes : primitives et fonctionnalités

Voici la définition en langage algorithmique d'une liste.

```
type    Liste = lien (pointeur) vers une cellule
        cellule = (enregistrement de deux champs)
                    val : élément
                    suiv : Liste
```

Liste est un lien vers un objet cellule, de la forme ci-dessous, où suiv est lui-même un lien vers une cellule.

Remarque : si L est une variable de type Liste, il faut demander une allocation de cellule pour pouvoir l'utiliser. Par exemple, suivant le langage, on peut déclarer Var Ma_Liste : Liste, puis demander une allocation de place : New(Ma_liste) ou Ma_Liste ← New(Liste).
A ce stade, une place est réservée, Ma_Liste est l'adresse de la zone mémoire, qui ne contient pas encore de valeur.

- **liste vide**
  Pour désigner une liste vide, on utilise Vide ou Nil (Null). Aucune place n'est réservée.
  Pour tester si une liste est vide, on utilise les booléens : L = Vide, ou Vide(L), ou L = Nil

- **Contenu (valeur d'un élément contenu dans une place)**
  Si L n'est pas vide, L.val est la valeur de son premier élément.
  tête désigne la place du premier élément : contenu(tête(L)) = L.val

- **Successeur (élément suivant l'élément courant, s'il existe).**
  La fonctionnalité successeur apparaît à travers fin.

- **Fin (liste suivant la première place)**

Si L n'est pas vide, L.suiv est la liste placée après la première place de L (autrement dit la liste privée de son premier élément), on dit aussi fin(L).
fin(<e_1, …, e_n>) = <e_2 …, e_n>

- **Ajout d'un élément**

La fonction cons permet d'ajouter un élément en tête d'une liste, en créant une nouvelle liste.
cons(e, Vide) = <e> et cons(e, <e_1, …, e_n>) = <e, e_1, …, e_n>.
Nous considérons cons comme une primitive de liste. Voici un algorithme possible, dans notre représentation :

```
Fonction Cons (e : élément ; L : Liste) : Liste
                                        {rend une liste constituée de la concaténation}
                                    {de e, ajouté en tête de la liste L passée en paramètre}
        {variable locale M : Liste}
M ← New(Liste)
M.val ← e
M.suiv ← L
Rendre M
{fin fonction cons}
```

On peut construire nombre d'autres fonctions classiques à partir des briques de base présentées ci-dessus. En voici quelques exemples.

- **Longueur d'une liste**

La longueur d'une liste L correspond au nombre d'éléments qu'elle contient. La fonction peut évidemment être récursive : la longueur de L est égale à 1+la longueur de « L privée de son premier élément ».

```
Fonction longueur (L : Liste) : entier
                                    /*calcule la longueur d'une liste*/
si L = Nil alors
        rendre 0
sinon rendre (1 + longueur (fin(L)))
{fin fonction}
```

La complexité d'une telle fonction est la longueur elle-même (linéaire en la taille des données, donc).
To do : rédiger la version itérative de cet algorithme.

- **Trouver la dernière place (dernier élément) d'une liste**

Le but est d'associer à la liste L un pointeur sur la dernière place : der(<e_1, …, e_n>) sera un pointeur sur la place contenant e_n (ou Nil si la liste est vide).

```
Fonction der
                        {rend la dernière place d'une liste, si elle existe}
        entrées : L : liste,
        sortie : un pointeur sur la dernière place
algo :
si L = Vide alors
        rendre Nil
sinon
        si fin(L) = Nil alors rendre L
        sinon rendre der(fin(L))
{fin fonction}
```

Là encore, la complexité est en la longueur de la liste.
To do : rédiger la version itérative de cet algorithme en faisant attention à la condition d'arrêt.

- **Recherche d'un élément dans une liste : est_dans ?**

Ici, on cherche si un élément donné est contenu dans la liste : on peut retourner soit un booléen, soit une place de la liste, selon les cas.
Il existe de nombreux algorithmes pour ce genre de recherche, nous en avons choisi un particulièrement trivial, toujours sous forme récursive : regarder si l'élément recherché est en tête de liste : si oui, on a gagné, sinon, on continue sur la liste privée de son premier élément.
Ici, on retournera un booléen.

```
Fonction est_dans? (L : Liste ; e : élément) : booléen
                                {rend vrai si l'élément e est dans la liste et faux sinon}
si L = Nil alors
        rendre faux
sinon
        si L.val = e alors rendre vrai
        sinon rendre(est_dans ?(L.suiv, e))
{fin fonction}
```

To do : modifier cette fonction pour renvoyer la place de l'élément dans la liste ou Nil.
To do Bis : donner une version itérative de cet algorithme.

- **Accéder à une place en particulier dans la liste**

L'accès est séquentiel : cela revient donc à avancer de k pas pour trouver le $k^{ième}$ emplacement et renvoyer un lien vers celui-ci… on renverra Nil (Null) dans le cas où cette $k^{ième}$ place n'existe pas.
Remarque : la $k^{ième}$ place de la liste L correspond à la $k-1^{ième}$ place de L privée de son premier élément.

```
Fonction accès( L : Liste ; k : entier) : Liste
                                {renvoie le lien vers la k^ième place de L, si elle existe}
Si ((L = vide) ou (k < 1)) alors
        rendre Nil
sinon
        si (k = 1) alors rendre L            {on est au bon emplacement}
        sinon rendre accès(L.suiv, k-1)      {il faut encore avancer}
fin_fonction
```

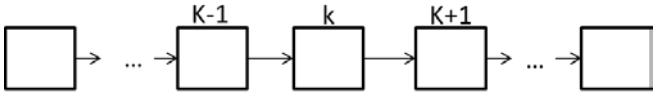
Complexité :
Opération fondamentale : affectation d'un pointeur.
N : longueur de la liste.

- accès à la place k : linéaire, donc en k
- complexité au mieux, Min(N,k) : 0 si la liste est vide (peu probable) ou 1 si accès au premier élément.
  on parle de complexité constante.
- complexité au pire, Max(N,k) : N, pour un accès en fin de liste.
  on parle de complexité linéaire.
- complexité moyenne : on suppose que les places sont équiprobables… ce qui nous donne le calcul suivant :
  $Moy(N) = \sum_{k=1..N} proba(k)*coût(k)$
  les places sont équiprobables, donc proba(k) = 1/N, donc

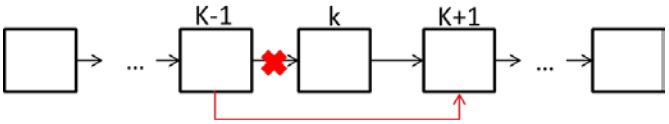
  $Moy(N) = 1/N \sum_{k=1..N} coût(k)$
  l'accès à la case k est linéaire en k, donc coût(k) = k, donc
  $Moy(N) = 1/N \sum_{k=1..N} k$
  ce qui, d'après les tableaux donnés dans le chapitre 1 donne :
  $Moy(N) = N*(N+1)/2N = (N+1)/2$ C'est une complexité linéaire.

- **Supprimer un élément de la liste**

On souhaite supprimer la case k (si elle existe) :



Il faut aller en case k-1, et la relier à la case k+1. On perd ainsi la trace de la place k.



L'algorithme peut prendre la forme suivante…

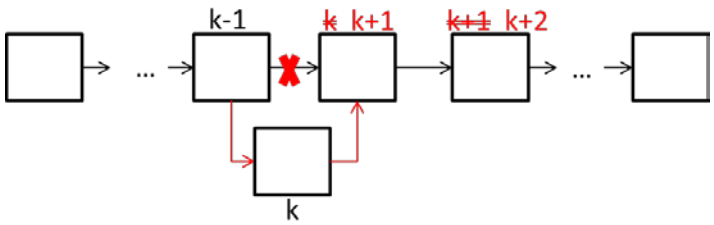
```
Fonction Supprimer (var L : Liste ; k : entier ;)
                                    /* utilise la fonction accès*/
                                    /*supprime la place n° k si 0<k ≤ longueur (L)*/
                                    /* modifie L durablement */
P ← accès(L, k-1)                   /* P est une variable locale à la fonction */
si (P = Vide) ou (P.suiv = Vide) alors                  /* la place n'existe pas */
        rendre Nil
sinon
        P.suiv ← P.suiv.suiv
        rendre L
{fin Supprimer}
```

Nb : dans cette fonction, on ne libère pas l'espace mémoire mobilisé pour k : il se peut que cette place soit utilisée ailleurs.

Complexité : il s'agit de la complexité de l'accès à l'élément k de la liste L.

- **Insérer un élément dans une liste**

Insérer un élément en place k est possible si k >0 et si la longueur de la liste est au moins k-1.
Pour ce faire, on crée une nouvelle place contenant l'élément, on se positionne en place k-1, on attache à la nouvelle place le suivant de la place de k-1 (actuelle place k) puis on attache à la place de k-1 la nouvelle place.



Bilan des opérations : on a modifié un pointeur et ajouté un autre pointeur.
Un algorithme possible serait celui-ci :

```
Fonction insérer (var L : Liste ; k : entier ; e : élément )
                                    /*insère e à la place k, avec 0< k ≤ 1+longueur(L)*/
si (k=1) alors
        rendre cons(e, L)
sinon   P ← accès(L, k-1)                       /* P est une variable locale*/
        si (P = Vide) alors rendre Nil
        sinon
                P.suiv ← cons(e, P.suiv)  /* on insère e comme 1er élément de fin de liste*/
                rendre L
```

La liste résultat de la fonction occupe les mêmes zones mémoire que la liste L d'origine, plus une place (celle de e).

Complexité :
Opération fondamentale : affectation d'un pointeur.
N : longueur de la liste.

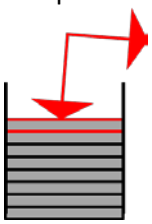
- Coût(Insertion en place k) = coût(accès en place k-1) + 2 affectations de pointeur, soit une complexité k+1.
- Complexité au mieux, Min(N, k) : 2, si l'insertion se fait en première place. C'est une complexité constante.
- Complexité au pire, Max(N, k) : de N+2 pour une insertion à la fin de la liste, en place N+1. C'est une complexité linéaire.
- Complexité moyenne : on suppose que toutes les places sont équiprobables
  Moy(N) = $\sum_{k=1..N}$ proba(k)*coût(k)
  = 1/N$\sum_{k=1..N+1}$ (k+1)
  = 1/(N+1) * ((N+2)*(N+3)/2-1)
  = 2 + N/2
  Complexité linéaire.

2. Les piles

Les Piles et les Files sont des structures séquentielles plus pauvres que les listes, qui correspondent à des implantations systèmes dans les couches basses : elles sont donc très importantes.
Les piles sont un cas particulier des listes, dans lequel les données ne peuvent être ajoutées ou retirées qu'à une même extrémité : le sommet. Elles sont de type LIFO (Last In, First Out). On peut simuler une pile avec une file, mais pas l'inverse.

Les piles correspondent à des structures matérielles très fréquemment utilisées en informatique, comme, par exemple, les empilements successifs de zones mémoires lors des appels récursifs de méthodes, ou encore la compilation (*e.g.* les piles permettent de gérer l'emboîtement des blocs d'instructions matérialisés par des accolades)

Une représentation physique intuitive pourrait être la pile d'assiettes sur laquelle on peut ajouter/retirer une assiette, mais au milieu de laquelle il serait périlleux de tenter d'accéder…



- **Une définition possible de pile**

  Type abstrait : Pile utilise : booléen, Élément
  Fonctionnalités de base, données d'emblée :
  Vide : on peut savoir si la pile est vide
  Sommet (ou tête) : on peut accéder au sommet de la pile
  Empiler : on peut empiler un élément en haut de la pile
  Dépiler : on peut retirer l'élément du haut de la pile.

L'interprétation sémantique est celle suggérée plus haut :

- Vide est une pile… vide (*i.e.* elle existe mais ne contient aucun élément) !
- vide?(P) ou p = Vide est vrai si et seulement si P = Vide.
  On peut donc tester une pile pour savoir si elle est vide.
- sommet(P) = e compare le contenu du haut de la pile est e.
  On peut « consulter » la dernière donnée ajoutée, et seulement celle-là.
- empiler(P, e) renvoie la pile P sur le sommet de laquelle on a déposé l'élément e.
  On peut empiler à tout moment (tant que la pile n'est pas pleine).
- dépiler(P) est ce qui reste de la pile P quand on lui a enlevé l'élément du haut.
  Tant que la pile n'est pas vide, on peut dépiler.
  Attention : dépiler ne fournit pas la valeur.

On ne peut rien faire d'autre que consulter, ajouter et retirer au sommet.
Exercice d'école : nous allons implanter une pile avec nos outils habituels, mais on n'utilisera qu'une partie des fonctionnalités de la structure de données, pour satisfaire les contraintes d'usage liées aux piles.

- **Implantation du type Pile avec des listes chaînées**

On veut représenter une liste d'élément du type Ma_Pile = <Térébenthine, Firmin, Gudule, Philéas> avec une pile, en utilisant des listes chaînées.
L'élément Philéas sera au fond de la pile et Térébenthine sera au sommet.

```
Type        Pile = pointeur sur Case
            Case = enregistrement
                    val : élément
                    suiv : Pile
```

En raccourci, on dit d'une variable P de type Pile qu'elle a deux champs : val et suiv.

Pile vide
Elle vaut Nil ou Null. Aucune place ne lui est réservée. Le test du vide est celui de l'égalité à Nil :

```
vide ?(P) = (P = Nil).
```

Sommet
C'est le contenu de la première variable pointée (si elle existe) : sommet(P) = P.val

Empiler
On ajoute au début (donc sur le dessus), comme avec Cons pour les listes.
Une nouvelle case a été créée par le Cons : la nouvelle pile Q occupe en mémoire les cases d'origine ainsi que cette nouvelle case.

```
Procédure empiler (e : élément ; var P : Pile)
                    /*ajoute l'élément e au sommet de la pile*/
P ← cons(e, P)
fin-procédure
```

Dépiler
Pour dépiler, on oublie la première case (si elle existe).

```
Procédure dépiler (var P : Pile)
                    /*non définie si la pile est vide*/
        P ← P.suiv /*enlève la première case*/
fin-procédure
```

On peut aussi créer une primitive qui dépile et récupère dans une variable l'élément du haut de pile.

```
Procédure enlever_haut (var P : Pile ; var v : élément)
                                                    /*pile non vide*/
        v ← P.val
        P ← P.suiv
fin-procédure
```

La difficulté est de n'utiliser que les fonctionnalités définies ci-dessus pour implémenter une Pile. En effet, ce type n'autorise que ces primitives de base et celles construites avec ces dernières. La suite présente quelques exemples.

- **Autres fonctions associées à une Pile**

Transvaser

On peut vouloir transvaser une pile dans une autre. Dans ce cas, la pile d'arrivée sera l'inverse de la première. Cette opération peut détruire la pile d'origine (cf. ci-dessous) et créer de nouvelles cases.

```
Procédure vider (var P, Q : Pile)
                                    /*met dans Q l'inverse de P, qui est détruite*/
Initialiser une pile Q
Tant que non vide?(P) faire
        Empiler(Q, sommet(P))
        Dépiler(P)
Fin tant que
```

Remarque : nous utiliserons souvent ce bloc, que nous appellerons transfert_tête(P,Q) :

```
Empiler(Q,sommet(P)) ; Dépiler (P)
```

Complexité : La complexité en temps est de l'ordre d'un parcours, c'est-à-dire, si N est la taille de la pile, N actions d'empilements et N actions de dépilements. En espace, cela nécessite une pile supplémentaire et la donnée est détruite.

Si l'on veut récupérer P non modifiée après le transfert dans Q, on peut construire deux piles inversées de P et transvaser à nouveau l'une des deux dans P.

```
Procédure inverser (P : Pile ; var Q : Pile)
                                    /*met dans Q l'inverse de P, qui reste intacte*/
Initialiser deux piles Q et R
Tant que non (vide?(P)) faire
        Empiler(Q, sommet(P))
        Empiler(R, sommet(P))
        Dépiler(P)
Tant que non (vide?(R)) faire
        Empiler(P, sommet(R))
        Dépiler(R)
```

Complexité : La complexité en temps est, si N est la taille de la pile, 3*N actions d'empilements et 2*N actions de dépilements. En espace, cela nécessite deux piles supplémentaires, mais les données initiales ne sont pas détruites.

Copier une pile

On peut faire une copie de P dans Q sans pour autant détruire P.

```
Procédure copier (P : Pile ; var Q : Pile)
                        /*met dans Q une copie de P, qui reste intacte*/
Initialiser deux piles Q et R
Tant que non (vide?(P)) faire
        Empiler(R, sommet(P)) ; Dépiler(P)
Tant que non (vide?(R)) faire
        Empiler(P, sommet(R))
        Empiler(Q, sommet(R))
        Dépiler(R)
```

Complexité : La complexité en temps est, si N est la taille de la pile, 3*N actions d'empilements et 2*N actions de dépilements. En espace, cela nécessite deux piles supplémentaires et la donnée n'est pas détruite.

Egalité

On peut tester l'égalité entre deux piles, en les détruisant (on pourrait faire des copies non destructives avant).

```
Fonction égale?(P, Q : Pile) : booléen
                        /*rend vrai si et seulement si les piles sont égales*/
b : variable locale booléenne initialisée à vrai
Tant que (non (vide?(P))) et (non(vide?(Q))) et (b) faire
        Si sommet(P) ≠ sommet(Q) alors
                b ← faux
        sinon
                dépiler(P)
                dépiler(Q)
rendre (b et vide?(P) et vide?(Q))
```

Complexité : la complexité en temps est au pire, si N est la taille de la pile, 2*N actions de dépilements et N comparaisons d'éléments.
En espace, cela nécessiterait deux piles supplémentaires pour ne pas détruire les données.

Hauteur

On peut calculer la hauteur d'une pile, sans la détruire.

```
Fonction hauteur(P : Pile) : entier
                        /*rend le nombre d'éléments de la pile sans la détruire*/
h : variable locale entière initialisée à 0
Q : pile locale initialisée à vide
Tant que (non vide?(P) ) faire
        h ← h+1
        Transfert_tête(P, Q)
Tant que (non vide?(Q) ) faire
        Transfert_tête(Q, P)
Rendre h
```

Complexité : à chaque passage dans la première boucle on fait un test, une incrémentation et un transfert de sommet, puis à nouveau une boucle avec N tests et N transferts de sommets.
La complexité est donc de l'ordre de la taille des données, autrement dit la hauteur.

3. Les Files

Les files sont un cas particulier des listes où les données ne peuvent être ajoutées qu'à une extrémité, la queue, et enlevées à l'autre, la tête. Elles sont de type FIFO (First In, First Out).
Les files ne se distinguent des piles que par la technique de suppression d'un élément, mais cette distinction modifie considérablement le pouvoir d'expression.
Les files correspondent aux structures matérielles de transmission du signal sur un fil, de transmission d'information sur un canal ou d'une file d'attente (à la boulangerie ou ailleurs).

Une représentation simple de la vie courante : un tuyau dans lequel transitent des balles de tennis ; elles entrent par un bout et sortent dans le même ordre, à l'autre bout.

Cette structure FIFO, physique ou logicielle, est essentielle dans les couches basses, les fils, mais aussi dans les grands réseaux de transmission.
Comme pour les piles, l'élément de base qu'était la place disparaît : on ne peut plus accéder qu'à une seule place et son contenu, la première à droite.

- **Une définition possible de File**

Type abstrait : File utilise: Booléen, Élément
Fonctionnalités de base :
Vide : on peut savoir si la file est vide
Premier : on peut accéder au premier élément de la file (à droite)
Ajouter : on peut ajouter un élément en queue de file (à gauche)
Décapiter : on peut couper la tête de la file (enlever son premier élément)

Interprétation sémantique :

- file_vide est une file… vide !
- premier(f) = e signifie que le contenu du premier élément de la file est e.
  On peut donc consulter la donnée qui est en début, mais seulement celle-là.
- vide?(f) est vrai si et seulement si f = file_vide.
- ajouter(f, e) retourne la file f à la queue de laquelle on a ajouté e.
- décapiter(f) retourne ce qu'il reste de la file f quand on lui a enlevé son premier élément.
  Attention, cette primitive ne fournit pas le premier élément.

Les opérations possibles sur une file sont : ajouter en queue et retirer ou consulter en tête (rien d'autre de façon immédiate).

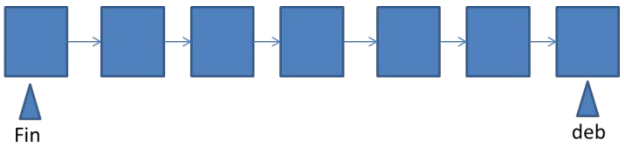
- **Implantation d'une File avec des listes chaînées**

On veut représenter une liste d'élément du type Ma_File = <Térébenthine, Firmin, Gudule, Philéas> avec une file, en utilisant des listes chaînées.
On peut utiliser des chaînages simples ou doubles. Nous avons choisi le chaînage simple, plus facile à définir mais plus compliqué à manipuler.
Une file est donnée par une succession chaînée de cases contenant les noms, sur laquelle pointent deux « poignées », l'une au début (là où l'on enlève, la tête), l'autre à la fin (là où l'on ajoute, la queue).
De fait, le champ fin pointe sur la dernière case de l'élément, le champ deb pointe sur première case de l'élément.

```
Type    File = enregistrement
                    deb : Liste
                    fin :Liste_chaînée

        Liste_chaînée = ^Case

        Case = enregistrement
                    val : élément
                    suiv : Liste
```



File Vide

Par convention, on décide qu'une file vide est une file pour laquelle F.deb = F.fin = Nil.
Le test du vide est celui de l'égalité à Nil : vide?(F) = (F.deb = Nil) et (F.fin = Nil).
NB : une file ne sera jamais déclarée pleine puisque l'allocation de place est dynamique.

Pour créer une file, on demande une allocation de place : New(F.deb) puis on dit que F.fin **est égale** à F.deb : F.fin:= F.deb (à ce stade, il n'y a rien dans la file).

NB : Pour travailler avec une file, il ne suffit pas de la déclarer, il faut aussi la créer.

Premier élément

Il s'agit du contenu de la première variable pointée : premier(F) = F.deb.val

Ajouter (en queue)

Pour ajouter, on crée une case après le champ repéré par fin. Si la file initiale était vide, les deux champs (deb et fin) pointent sur cette nouvelle case.

```
Procédure ajouter (mot : élément ; var F : File)
                                /*ajoute le mot à gauche de la file*/
New(Q) ;
Q.val ← mot ;
Q.suiv ← Nil
si (F.fin ≠ Nil) alors
        Q ← F.fin.suiv
        F.fin ← Q
sinon
        F.fin ← Q
        F.deb ← Q
```

Enlever (décapiter)

Pour décapiter la file, il faut lui couper la tête, *i.e.* retirer la première case, si elle existe : il suffit de la sauter. S'il n'y a qu'un seul élément dans la file d'origine, on doit également modifier le champ fin.

```
Procédure décapiter(var F : File)
                                /*non définie si la file est vide*/
Si F.deb ≠ Nil alors
        F.deb ←  fin(F.deb)     /*enlève la case la plus à droite*/
Si F.deb = Nil alors
        F.fin ←  Nil
```

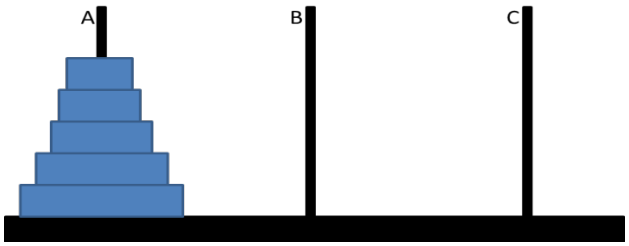
Décapiter ET récupérer la valeur
Variante de la primitive précédente : on décapite la file tout en récupérant le premier élément dans une variable v de la procédure appelante (passage par adresse).

```
Procédure enlever(var F : File ; var v : élément)
                                        /*non définie si la file est vide*/
v ← F.deb.val
F.deb ← F.deb.suiv      /*enlève la case la plus à droite*/
Si F.deb = Nil alors
        F.fin ← Nil
```

Là encore, on ne peut utiliser que ces fonctions de base pour implémenter d'autres fonctionnalités utiles : longueur, recherche d'un élément, accès à la $k^{ième}$ place, insertion, etc… à vous de jouer !

4. Récursivité : les tours de Hanoï

Le but est de manipuler la récursivité sur les piles en traitant des « tours de Hanoï ».
Encore le folklore : il s'agit d'un jeu d'enfants, à savoir des disques de bois de diamètres connus et distincts, que l'on empile sur des bâtons pour faire des tours.
Les disques ne pourront être placés et déplacés qu'en suivant certaines règles. Le jeu donne naissance à des algorithmes éminemment simples en récursif et généralement moins en itératif, de complexité souvent exponentielle.



Définition : une tour (de Hanoï) est une pile de disques telle qu'un disque ne puisse reposer que sur un disque de diamètre supérieur.
Le but du jeu est de déplacer la pile complète d'un pic à un autre tout en respectant la règle des diamètres.
La manipulation nécessaire peut-être rédigée sous forme récursive, en utilisant une Pile comme structure de données.

Nous représenterons les disques par un type quelconque pour lequel on sait définir un champ (ou une fonction) diamètre et les tours par des piles de disques.

```
Type Disque =  enregistrement
                        diamètre : flottant
                fin-enregistrement
Tour = Pile de Disques
```

Vous verrez en TD diverses fonctions sur les tours et en calculerez la complexité.