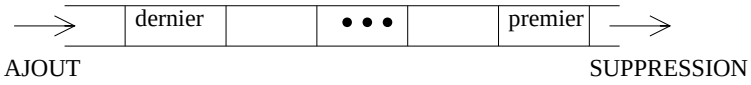
COURS : Files / Listes

1 Files

Correspond à la notion bien connue de file d'attente, par exemple les tickets d'attente SNCF ou sécu ; on parle souvent de structure FIFO (First In First Out). Cette structure est très souvent utilisée par les systèmes d'exploitation pour gérer l'accès à une ressource partageable comme une imprimante ou un disque : liste des fichiers en attente d'impression, etc... Dans une telle structure, les ajouts se font à une extrémité (fin de liste) et les suppressions à une autre (tête de liste) :



On déclarera :

```
type TFile = File de TInfo
var F : TFile
```

1.1 primitives

```
Action créerFile(S F : TFile)
Fonction fileVide(F: TFile) : booléen;
Fonction valeurPremier(F : TFile) : TInfo;
Action enfiler(ES F : TFile ; E Elem : TInfo);
Action défiler(ES F : TFile)
```

CréerFile initialise une file à vide. Doit être appelée avant toute utilisation. **FileVide** permet de savoir si une file est vide ou non. **ValeurPremier** retourne la valeur de l'élément présent en tête de file, cet élément restant dans la file. **Enfiler** rajoute un élément qui se retrouve en dernière position dans la file. **Défiler** supprime de la file l'élément se trouvant en première position. **FileVide**, **ValeurPremier**, **Enfiler** et **Défiler** ne sont pas définies sur une file dont la valeur est indéterminée. **ValeurPremier** et **Défiler** ne sont pas définies sur une file vide.

1.2 Exercices

Exercice 1 : Ecrire une action qui inverse une file passée en entrée/sortie.

Solution :

```
Action inverserFile(ES  F : TFile)

var P : Pile de TInfo

Début
 créerPile(P)
 TantQue non fileVide(F)
 Faire début
         empiler(P, valeurPremier(F))
         défiler(F)
         fin
 TantQue non pileVide(P)
 Faire début
         enfiler(F, valeurSommet(P))
         dépiler(P)
         fin
Fin
```

Exercice 2 : Même chose, sauf qu'il s'agit maintenant d'inverser une pile.

Remarque : ceci est une 2 ème version, on a vu une 1ère version dans le cours sur les piles, il fallait passer par 2 piles intermédiaires...

Solution :

```
Action inverserPile(ES  P : TPile)

var F : File de TInfo

début
 créerFile(F)
 TantQue non pileVide(P)
 Faire début
         enfiler(F, valeurSommet(P))
         dépiler(P)
         fin
 TantQue non fileVide(F)
 Faire début
         empiler(P, valeurPremier(F))
         défiler(F)
         fin
Fin
```

2 Listes

2.1 Présentation

Présentation intuitive

Une *liste* est une suite d'éléments. Les places des éléments sont ordonnées, pas nécessairement les éléments eux-mêmes ; autrement dit, $(1, 7, 2, 2, 9)$, $(1, 2, 7, 9, 2)$ et $(1, 2, 2, 7, 9)$ sont trois listes d'entiers différentes. On peut **ajouter** ou **supprimer** un élément en **n'importe quelle position** dans la liste. On peut accéder (consulter, modifier) au premier élément ; quand on a accédé à un élément, on peut accéder au suivant s'il existe. On peut donc parcourir la liste du premier au dernier élément, en passant chaque fois d'un élément à l'élément suivant ; c'est pourquoi on parle parfois de liste séquentielle.

Description plus précise

la liste $(1, 7, 2, 2, 9)$ par exemple peut être vue comme une liste dont le premier élément est l'entier 1, et dont la suite est la liste $(7, 2, 2, 9)$. En itérant cette description, on obtient la liste

$(1, (7, (2, (2, (9, ())))))$

La dernière sous-liste est vide : $()$

Les seules primitives utiles sont donc : savoir si une (sous-)liste est vide, consulter le premier élément d'une (sous-)liste, passer à la sous-liste suivante, insérer/supprimer un élément en tête d'une (sous-)liste. Considérons par exemple la liste suivante, qui offre la particularité d'être ordonnée :

$(1, (1, (2, (7, (9()))))))$

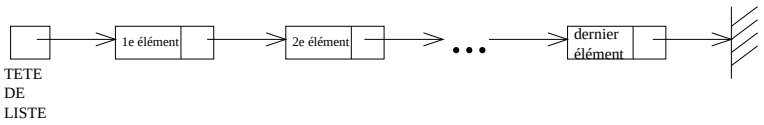
et supposons que l'on veuille insérer 4 en respectant l'ordre des éléments ; il suffit de voir que le premier élément est trop petit, de même le premier de la sous-liste suivante, etc... et on arrive sur le 2 ; on voit alors que le premier de la sous-liste suivante est plus grand que 4, on va donc insérer 4 en tête de la sous-liste suivante, pour obtenir :

$(1, (1, (2, (4, (7, (9()))))))$

Le même principe permet de supprimer le premier élément de n'importe quelle sous-liste non vide, donc de supprimer n'importe quel élément d'une liste non vide.

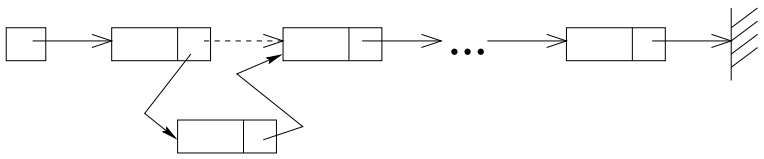
Présentation "algorithmique"

En pratique, on va distinguer le premier élément ; on dira que l'on insère/supprime un élément en tête de liste, ou après tel ou tel élément :

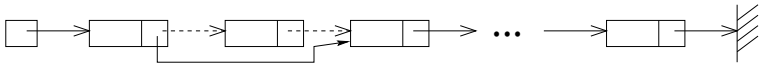


Montrer la notion de *tête de liste*, de *flèche*. Tout élément, sauf le dernier, a un suivant. Il n'y pas encore de notion de précédent. Finalement, une liste est un ensemble de couples (élément, flèche) où la flèche désigne l'élément suivant.

Voici par exemple comment on peut insérer un élément (ici entre le 1e et le 2e élément) :



Et pour supprimer :



La position physique des éléments est absolument indépendante de leur ordre logique au sein de la liste ; ce dernier est uniquement lié au mécanisme des flèches. Il suffit donc de modifier les flèches pour changer l'ordre des éléments dans la liste.

Nous introduisons un type `TAdresse` qui matérialise la notion de flèche ; intuitivement, la valeur d'un objet de type `TAdresse` correspond à l'adresse physique d'un objet de la liste. Une liste de `TInfo` se déclarera :

```
type TListe = Liste de TInfo;
var L : TListe;
```

Le type `TListe` va utiliser le type `TAdresse`, dont une valeur particulière : `NULL`, indiquera la flèche qui ne débouche sur rien (flèche du dernier élément, ou liste vide).

2.2 Primitives

```
Action créerListe( S L : TListe );
Fonction adressePremier( E L : TListe ) : TAdresse;
// Retourne NULL si la liste est vide.
Fonction adresseSuivant( E L : TListe, E Adr : TAdresse ) : TAdresse;
Fonction valeurElement( E L : TListe, E Adr : TAdresse ) : TInfo;
Action modifieValeurElement( E L : TListe, E Adr : TAdresse, E Elem : TInfo );
Action insérerEnTête( ES L : TListe, E Elem : TInfo );
Action insérerApres( ES L : TListe, E Elem : TInfo, E Adr : TAdresse );
Action supprimerEnTête( ES L : TListe );
Action supprimerAprès( ES L : TListe, E Adr : TAdresse );
```

Toutes les primitives sont indéfinies si la liste n'est pas créée. Seules les primitives `adressePremier` et `insérerEnTête` ont un sens dans le cas où la liste est vide. Pour les primitives dont un paramètre est une adresse, si l'adresse donnée en argument ne correspond pas à une adresse d'un élément de la liste, alors le comportement de la primitive est indéterminé.

2.3 Algorithme de parcours

On illustre l'algorithme de parcours à travers une action qui affiche le contenu de la liste L :

```
Action afficheContenu(E L : Tliste)
var Adr : TAdresse
début
Adr <- adressePremier(L)
Tant Que Adr <> NULL
Faire début
      écrire (valeurElément(L, Adr))
      Adr <- adresseSuivant(L, Adr)
      fin
fin
```

2.4 Un exemple illustrant presque toutes les primitives

Faire sous forme d'exemple mystère que les étudiants font tourner sur un exemple pour décrouvrir que cette action inverse une liste.

Truc : insertion en tête de l'élément suivant.

Remarque : il serait possible mais déraisonnable (trop coûteux) d'utiliser une file.

```
Action inverserListe (ES L : TListe)
var Adr, Suiv : TAdresse

début
Adr <- adressePremier (L)
Si Adr <> NULL
Alors début
      Suiv <- adresseSuivant (L, Adr)
      Tant Que Suiv <> NULL
      faire début
            insérerEnTête (L, valeurElément (L, Suiv))
            supprimerAprès(L, Adr)
            Suiv <- adresseSuivant (L, Adr)
            fin
      fin
fin
```

2.5 Algorithmes usuels

2.5.1 Algorithmes de recherche

- Recherche globale

```
<Initialisations>
Tant Que <pas trouvé> et <pas a la fin>
Faire Si <élément cherché> = <élément courant>
      Alors <on a trouvé>
      Sinon <on passe au suivant>
```

- Recherche partielle (parce que trié par exemple)

```
<Initialisations>
Tant Que <pas trouvé> et <pas a la fin> et <cherche encore>
Faire Si <élément cherché> = <élément courant>
      Alors <on a trouvé>
      Sinon Si <inutile de chercher plus loin>
            Alors <on ne cherche plus>
            Sinon <on passe au suivant>
```

Exercice 3 : Fonction de recherche dans une liste non triée :

```
Fonction recherche (L : TListe ; Elém : TInfo) : TAdresse
// retourne l'adresse de Elém dans la liste L, NULL si Elém n'est pas présent.

var Adr : TAdresse ; Trouvé : booléen

début
Adr <-adressePremier(L)
Trouvé <- Faux
Tant Que non Trouvé et Adr <> NULL
Faire Si valeurElément(L, Adr) = Elem   // = supposé défini pour TInfo
```

```
      Alors Trouvé <- Vrai
      Sinon Adr <- adresseSuivant(L, Adr)
Retourner Adr
fin
```

Exercice 4 : Modifier la fonction précédente pour l'adapter à une liste triée :

```
Fonction rechercheTriee (L : TListe ; Elém : TInfo) : TAdresse
// retourne l'adresse de Elém dans la liste L, NULL si Elém n'est pas présent.

var Adr : TAdresse ; Trouvé : booléen
    Fini : booléen

début
Adr <-adressePremier(L)
Trouvé <- Faux
Fini <- Faux
Tant Que non Trouvé et non Fini et Adr <> NULL
Faire Si valeurElément(L, Adr) = Elem   // = supposé défini pour TInfo
      Alors Trouvé <- Vrai
      Sinon Si valeurElément(L, Adr) > Elem
            Alors Fini <- Vrai
            Sinon Adr <- adresseSuivant(L, Adr)
Si non Trouvé
Alors Adr <- NULL
Retourner Adr
fin
```

2.5.2 Algorithmes de mise à jour

Donner juste les grands principes avec des schémas, les algos seront écrits en TD.

- Ajout non trié (trivial, en tête) et trié
- Suppression non trié et trié : s'appuie sur la recherche

2.6 Insertion d'un élélement en fin de liste

permet de voir la notion d'adresse précédente.

```
Action InsèreElémentFin (ES L : TListe ; E Elém : TInfo)
var Adr, AdrPréc : TAdresse

début
Adr <- adressePremier(L)
AdrPréc <- NULL
Tant que Adr <> NULL
Faire début
        AdrPréc <- Adr
        Adr <- adresseSuivant(L, Adr)
      fin
Si AdrPréc = NULL Alors insérerEnTête(L, Elém)
                  Sinon insérerAprès(L, Elém, AdrPréc)
fin
```

2.7 Concaténation de 2 listes dans une 3ème

permet de voir le "truc" de l'élément fictif ajouté pour éviter de nombreux tests sur la liste

dire attention, ne pas utiliser ce truc systématiquement, voir si d'autres solutions plus élégantes.

```
Action concat (E L1, L2 : TListe ; S L3 : TListe)
var Fictif : TInfo
    Adr, Suiv : TAdresse

début
créerListe(L3)
insérerEnTête(L3, Fictif)
Suiv <- adressePremier(L3)
Adr <- adressePremier(L1)
Tant Que Adr <> NULL
Faire début
      insérerAprès(L3, valeurElément(L1,Adr), Suiv)
      Suiv <- adresseSuivant(L3, Suiv)
      Adr <- adresseSuivant(L1, Adr)
      fin
Adr <- adressePremier(L2)
Tant Que Adr<> NULL
Faire début
      insérerAprès(L3, valeurElément(L2,Adr), Suiv)
      Suiv <- adresseSuivant(L3, Suiv)
      Adr <- adresseSuivant(L2, Adr)
      fin
supprimerEnTête(L3) // suppression de l'élément fictif
fin
```