T NSI **Structures de données linéaires**

1. Généralités

Les structures de données abstraites

Lorsqu'on crée un algorithme, on s'aperçoit assez vite que, pour manipuler des données, l'utilisation de simples variables de type "nombres" (entiers ou flottants) ou "booléens" n'est pas suffisant. Par exemple, on a vu que le regroupement de données dans un tableau permet d'accéder à ces données simplement à l'aide d'un index.
Au stade de l'algorithme, et indépendamment d'une quelconque implémentation, on peut envisager des structures de données plus complexes, selon les problèmes à traiter.
On se donne une notation pour décrire ces structures ainsi que l'ensemble des opérations que l'on peut appliquer et les propriétés de ces opérations. On parle alors de **type abstrait de données**.
La notion de type abstrait permet de définir des types de données "non primitifs", c'est-à-dire non disponibles dans les langages de programmation courants.
Les types de données "primitifs" sont par exemple les entiers, les flottants, les booléens.

Nous étudierons au cours de l'année 4 types de structures de données abstraites :
- Les structures **linéaires** : les listes, les piles et les files.
- Les structures à **accès par clés** : les dictionnaires.
- Les structures **hiérarchiques** : les arbres.
- Les structures **relationnelles** : les graphes.

Remarque : Ces structures de données sont parfois "prévues" nativement dans certains langages de programmation comme type de données, mais ce n'est pas toujours le cas !

Une structure de données possède un ensemble de routines (procédures ou fonctions) permettant d'ajouter, d'effacer, d'accéder aux données. Cet ensemble de routines est appelé **interface**.

Les opérations élémentaires

L'interface est généralement constituée des 4 routines élémentaires dites CRUD :
Create : ajout d'une donnée.
Read : lecture d'une donnée.
Update : modification d'une donnée.
Delete : suppression d'une donnée.

Derrière les opérations de lecture, de modification ou de suppression d'une donnée se cache une autre routine tout aussi importante : la **recherche** d'une donnée. Car il faut d'abord trouver la donnée dans la structure avant de pouvoir la lire, la modifier ou la supprimer.

2. Les listes

Définition : Une **liste** est une structure de données permettant de regrouper des données et dont l'accès est séquentiel. Elle correspond à une suite finie d'éléments repérés par leur rang (index). Les éléments sont ordonnés et leur place a une grande importance.
Une liste est évolutive, on peut ajouter ou supprimer n'importe quel élément.

Voici deux exemples d'opérations qui peuvent être effectuées sur une liste créée :

INSERER(L, x, i) : repère le donnée de rang i-1 puis décale de 1 rang à droite tous les éléments situés derrière elle et insère x au rang i.
SUPPRIMER(L, i) : qui repère la donnée de rang i (en commençant par le premier élément), puis décale de 1 rang à gauche tous les éléments situés derrière elle.

Type abstrait Liste :

Type abstrait : Liste.
Données : éléments de type T.
Opérations :

CREER_LISTE_VIDE() qui retourne un objet de type Liste.
La liste existe et elle est vide.
INSERER(L, e, i)
L'élément e est inséré à la position i dans la liste L.
SUPPRIMER(L, i)
L'élément situé à la position i est supprimé de la liste L.
RECHERCHER(L, e) qui retourne un objet de type Entier.
L'élément e est recherché dans la liste L et on retourne son index (sa position).
LIRE(L, i) qui retourne un objet de type T.
L'élément situé à la position i dans la liste L est retourné.
MODIFIER(L, i, e)
L'élément situé à la position i dans la liste L est écrasé par le nouvel élément e.
LONGUEUR(L) qui retourne un objet de type Entier.
Le nombre d'éléments présents dans la liste L est retourné.

Conditions :

LIRE(L, i), SUPPRIMER(L, i), MODIFIER(L, i, e) sont définis si, et seulement si :
$1 \leq i \leq$ LONGUEUR(L).

INSERER(L, e, i) est défini si, et seulement si, $1 \leq i \leq$ LONGUEUR(L) + 1.

Exemple d'application de ce type abstrait :

Ecrire l'évolution de la liste L lors de l'exécution de la suite d'instructions suivante :

L = CREER_LISTE_VIDE()
INSERER(L,'A', 1)
INSERER(L,'O', 2)
INSERER(L,'B', 1)
INSERER(L,'V', 3)
INSERER(L,'R', 2)

Représentation d'une liste avec un tableau
Tous les langages de programmation permettent de réaliser des tableaux.
On utilise un tableau de taille fixe n + 1 dont les indices vont de 0 à n.
●La première case du tableau (indice 0) contient le nombre d'éléments présents dans la liste.

●Les cases suivantes du tableau (indices 1 à n) contiennent les éléments de la liste ou sont vides.
Dans cette représentation, une liste a donc une taille maximale : n. Lorsque la liste atteint sa taille maximale, on dit qu'elle est pleine.
Il est utile de définir deux opérations en plus de celles vues précédemment :
EST_VIDE(L) qui retourne un objet de type Booléen.
Retourne Vrai si la liste L est vide c'est-à-dire si LONGUEUR(L) == 0.
EST_PLEINE(L) qui retourne un objet de type Booléen.
Retourne Vrai si la liste L est pleine c'est-à-dire si LONGUEUR(L) == n.

Exemple :
On représente une liste à l'aide d'un tableau de taille fixe 6.
Lors de la création de la liste vide, toutes les cases du tableau sont initialisées à 0.
Ecrire le contenu du tableau à chaque étape :
L = CREER_LISTE_VIDE()
INSERER(L, 3, 1)

INSERER(L, 5, 2)

INSERER(L, 8, 1)

SUPPRIMER(L, 2)

Exercice 1 :

On donne la séquence d'instructions suivante : (on supposera que les listes ne sont jamais pleines)

```
L1 = CREER_LISTE_VIDE()
L2 = CREER_LISTE_VIDE()
INSERER(L1, 1, 1)
INSERER(L1, 2, 2)
INSERER(L1, 3, 3)
INSERER(L1, 4, 4)
INSERER(L2, LIRE(L1, 1), 1)
INSERER(L2, LIRE(L1, 2), 1)
INSERER(L2, LIRE(L1, 3), 1)
INSERER(L2, LIRE(L1, 4), 1)
```

1. Illustrer le résultat de chaque étape de cette séquence.
2. Quelle est l'opération effectuée sur L1 permettant d'obtenir L2 ?

Exercice 2 :

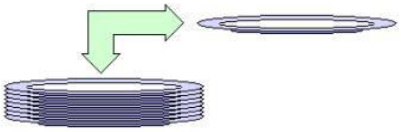
Implémenter en Python chaque fonction du type abstrait Liste.

Parmi les listes, il existe 2 structures de données particulières dont les accès mémoire sont optimisés : les piles et les files.

3. Les piles

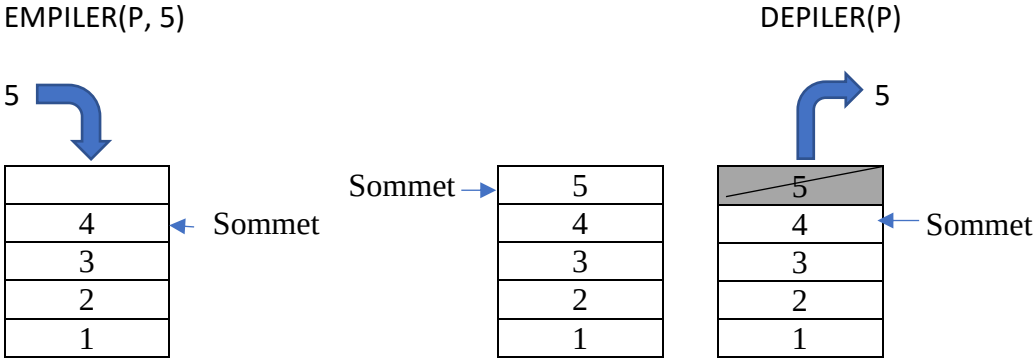
Définition : il s'agit d'une structure des données qui donne accès en priorité aux dernières données ajoutées. La dernière donnée ajoutée sera la première à en sortir. Autrement dit, **on ne peut accéder qu'à l'objet situé au sommet de la pile**. On résume cela par l'expression : **"dernier entré, premier sorti"** c'est-à-dire, en anglais, **"Last In, First Out" (LIFO)**.

Analogie : Pour "visualiser" une pile (informatique), penser à une pile d'assiettes : l'ordre dans lequel les assiettes sont dépilées est l'inverse de celui dans lequel elles ont été empilées, puisque seule l'assiette supérieure est accessible.



Les deux opérations élémentaires dont on a besoin avec cette structure sont :

EMPILER(P, x) qui correspond à l'insertion de la donnée x au sommet de la pile P si la pile n'est pas pleine.
DEPILER(P) qui retire la dernière donnée de la pile Pet la retourne si la pile n'est pas vide.



On peut aussi définir d'autres opérations comme :
PILE_VIDE(P) qui indique si la pile P est vide ou non.
PILE_PLEINE(P) qui indique si la pile P est pleine ou non.

Applications : par exemple, dans un navigateur web, une pile peut servir à mémoriser les pages visitées. L'adresse de chaque nouvelle page visitée est empilée et en cliquant sur un bouton "Afficher la page précédente", on dépile l'adresse de la page précédente.
Autre exemple : dans une console Python, les entrées successives sont stockées dans une pile.

Type abstrait Pile

Type abstrait : Pile
Données : éléments de type T

Opérations
CREER_PILE_VIDE() qui retourne un objet de type Pile.
La pile existe et elle est vide.

EMPILER(P, e)
L'élément est inséré au sommet de la pile P.

DEPILER(P) qui retourne un objet de type T
L'élément situé au sommet de la pile P est enlevé de la pile et est retourné.
EST_VIDE(P) qui retourne un objet de type Booléen.
Retourne Vrai si la pile P est vide et retourne Faux sinon.
EST_PLEINE(P) qui retourne un objet de type Booléen.
Retourne Vrai si la pile P est pleine et retourne Faux sinon.

Conditions :
EMPILER(P, e) est définie si, et seulement si, EST_PLEINE(P) = Faux.
DEPILER(P) est définie si, et seulement si, EST_VIDE(P) = Faux.

Exemple d'application de ce type abstrait :

On considère la suite d'instructions suivante :
P = CREER_PILE_VIDE()
EMPILER(P, 3)
EMPILER(P, 2)
N = DEPILER(P)
EMPILER(P, 5)
EMPILER(P, 7)
EMPILER(P, 9)

Donner une représentation de la pile P à chaque étape et le contenu de la variable N.

Représentation d'une pile avec un tableau :

On peut réaliser une pile capable de contenir n éléments avec un tableau P[0..n] pouvant contenir (n + 1) éléments :
●La première case du tableau (d'indice 0) contient l'indice de la prochaine case vide (c'est l'indice qui correspondra au prochain élément à insérer dans la pile).
●Les cases suivantes du tableau (d'indices 1 à n), contiennent les éléments de la pile ou sont vides. La dernière case non vide du tableau est le sommet de la pile.

Remarques :
●Si P[0] == 1 alors la pile est vide.
●A chaque fois qu'on empile un élément, on augmente P[0] d'une unité.
●A chaque fois que l'on dépile un élément, on diminue P[0] d'une unité, et il n'est pas nécessaire de toucher, dans le tableau, à la donnée qui vient d'être dépilée. Elle est toujours présente "physiquement" dans le tableau, mais on n'y a plus accès par l'intermédiaire des fonctions sur la pile.
●Lorsque P[0] == n + 1 alors la pile est pleine.

Exemple :
On représente une pile P à l'aide d'un tableau de taille fixe 6 : P[0..5]
Lors de la création de la pile vide, toutes les cases du tableau sont initialisées à 0, sauf P[0] qui contient 1.
1) Ecrire le contenu du tableau si l'on empile successivement dans P les éléments 8, 3, 5 :
2) Ecrire le contenu du tableau si on exécute la fonction DEPILER(P).

Exercices sur les piles :

Exercice 1 :
On suppose que l'on a placé dans une pile P la chaine de caractères "BONJOUR". On suppose que l'on dispose des opérations de base sur les piles. Ecrire un algorithme permettant de créer une pile N contenant les éléments de P mais en sens inverse, c'est-à-dire représentant la chaine de caractères "RUOJNOB".

Exercice 2 :

Implémenter en Python les opérations classiques sur les piles à l'aide d'un tableau que l'on "simulera" à l'aide d'une liste. On supposera que le tableau a une taille fixe, par exemple 10.

Exercice 3 :

Proposer une implémentation des opérations classiques de la pile à l'aide des méthodes .pop() et .append() du type liste de Python.

Exercice 4 : Bon parenthèsages ?

Une expression (algébrique ou arithmétique) est correctement parenthésée si d'une part, le nombre de parenthèses et crochets, ouvrants et fermants, est le même et si d'autre part les correspondances ne se croisent pas.
Par exemple, l'expression "[3 + (5 – 7] * 3)" n'est pas correctement parenthésée car la parenthèse fermante ne peut pas venir après le crochet fermant.
1. Ecrire un algorithme qui s'aide d'une pile pour contrôler le bon parenthésage d'une expression.
2. Ecrire en Python une fonction Est_bien_Parenthesee(E) qui renvoie True si l'expression E est correctement parenthésée, et False dans le cas contraire.

Exercice 5 : Calcul d'une expression arithmétique postfixée

On ne considère ici que les expressions numériques contenant les quatre opérations élémentaires (+, −, *, /).
La notation postfixée, appelée aussi Notation Polonaise Inversée (NPI), consiste à mettre les opérations arithmétiques **après** leurs arguments.
Par exemple, l'expression 5 + 3 s'écrit : 5 3 + en NPI.
L'avantage, c'est que la NPI se passe complètement de parenthèses. Par exemple, l'expression 3*((4 + 5) − 6) peut s'écrire en NPI : 3 4 5 + 6 − *.
On se propose de se servir d'une **pile** pour calculer une expression arithmétique exprimée en NPI.
Les données sont lues comme une suite de caractères. Chaque donnée est séparée des autres par un espace. La méthode consiste à utiliser une **pile** pour conserver les **valeurs numériques** au fur et à mesure de leur lecture (de la gauche vers la droite), et à effectuer un traitement quand un **opérateur** est lu. Ce traitement doit dépiler les deux dernières valeurs présentes dans la pile, effectuer l'opération, puis empiler le résultat dans la pile.
Il faut être attentif ici à l'ordre des opérandes dans le cas des opérateurs non commutatifs (– et /).
Le résultat final est contenu dans la pile.

1. Ecrire un algorithme utilisant une pile pour effectuer un calcul en NPI.
2. Implémenter cet algorithme en Python en définissant une fonction calcul(E) où E est l'expression en NPI, écrite sous forme d'une chaine de caractères dans laquelle chaque nombre ou symbole est séparé par un espace.

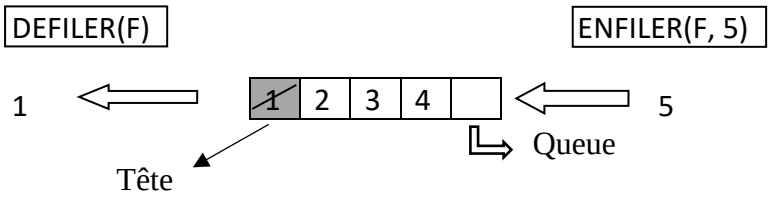
4. Les files

Définition : Une file est une structure de données dans laquelle on accède aux éléments suivant la règle du "premier arrivé, premier sorti". Autrement dit, on ne peut accéder qu'à **l'objet situé au début de la file**. On décrit souvent ce comportement par l'expression **"premier arrivé, premier sorti"**, c'est-à-dire, en anglais, **"First In, First Out" (FIFO)**. La file comporte une **tête** et une **queue**.

Analogie : la **file d'attente** des clients à un guichet ou à une caisse convient à cette description. En effet, le client qui passe en premier est celui qui est arrivé le premier.

Exemple d'application : Lorsqu'on veut imprimer plusieurs documents sur une imprimante, ceux-ci sont stockés dans une file : celui qui aura été lancé en premier sera le premier imprimé.

Les deux opérations élémentaires dont on a besoin pour cette structure sont :
ENFILER(F, x) qui correspond à l'insertion de la donnée x à la queue de la file F si la file n'est pas pleine.
DEFILER(F) qui retire la donnée de tête de la file F et la retourne si la file n'est pas vide.



Type abstrait File

Type abstrait : File
Données : éléments de type T

Opérations
CREER_FILE_VIDE() qui retourne un objet de type File.
La file existe et elle est vide.

ENFILER(F, e)
L'élément est inséré en queue de la file F.
DEFILER(F) qui retourne un objet de type T
L'élément situé en tête de la file F est enlevé de la file et est retourné.
EST_VIDE(F) qui retourne un objet de type Booléen.
Retourne Vrai si la file F est vide et retourne Faux sinon.
EST_PLEINE(F) qui retourne un objet de type Booléen.
Retourne Vrai si la file F est pleine et retourne Faux sinon.

Conditions :
ENFILER(F, e) est définie si, et seulement si, EST_PLEINE(F) = Faux.
DEFILER(F) est définie si, et seulement si, EST_VIDE(F) = Faux.

Exemple d'application de ce type abstrait :

On considère la suite d'instructions suivante :
F = CREER_FILE_VIDE()
ENFILER(F, 21)
ENFILER(F, 22)

ENFILER(F, 23)
N = DEFILER(F)
ENFILER(F, 24)
ENFILER(F, 25)
N = DEFILER(F)

Donner une représentation de la file F à chaque étape et le contenu de la variable N.

Représentation d'une file avec un tableau

On peut par exemple réaliser une file capable de contenir *n* éléments avec un tableau F[0 .. (*n* + 2)] pouvant contenir (*n* + 3) éléments :
●La première case du tableau (d'indice 0) contient l'indice de la tête de la file.
●La deuxième case du tableau (d'indice 1) contient l'indice de la queue de la file, c'est-à-dire la prochaine case disponible pour la queue.
●La troisième case du tableau (d'indice 2) contient le nombre d'éléments présents dans la file, c'est-à-dire la taille de la file.
●Les cases suivantes du tableau (d'indices 3 à *n* + 2) contiennent les éléments de la file ou sont vides.

Remarques :
Si (taille == 0) la file est vide et si (taille == *n*) la file est pleine.
A chaque fois qu'on enfile un élément, on augmente la taille d'une unité ainsi que la queue.
A chaque fois qu'on défile un élément, on diminue la taille d'une unité et on augmente la tête d'une unité.
Dès que les indices de tête ou de queue dépassent la longueur du tableau, ils repartent au début du tableau (on appelle cela un tableau avec une gestion circulaire).

Exemple de représentation pour *n* = 5 (5 sera la taille maximale de la file) :

Après enfilage de 8, 3 et 5 (la file contient (8, 3 , 5)) :

Tête	**Queue**	**Taille**					
3	**6**	**3**	8	3	5		
0	1	2	3	4	5	6	7

Après un défilage (la file contient (3 , 5))

Tête	**Queue**	**Taille**					
4	**6**	**2**	8	3	5		
0	1	2	3	4	5	6	7

Après un défilage supplémentaire (la file contient (5)) :

Tête	**Queue**	**Taille**					
5	**6**	**1**	8	3	5		
0	1	2	3	4	5	6	7

Après enfilage de 10, puis 12 (la file contient (5, 10, 12)) :

Tête	**Queue**	**Taille**					
5	**3**	**3**	8	3	5	10	12
0	1	2	3	4	5	6	7

L'indice de Queue devrait être 8, mais comme on dépasserait le dernier indice possible du tableau, on revient au début du tableau à la première case "vide". On sait que c'est possible car la taille de la liste (3) est inférieure à 5 (taille maximale de la file).

Après trois défilages successifs , la file est vide :

Tête	**Queue**	**Taille**					
3	**3**	**0**	8	3	5	10	12
0	1	2	3	4	5	6	7

Là aussi, l'indice de Tête devrait être 8, mais comme on dépasserait le dernier indice possible du tableau, on revient au début du tableau : 8 – 5 = 3.
Maintenant, tout se passe comme si la file venait d'être créée vide.

Remarque : on a toujours (**Queue** – **Tête**) = **Taille** **(modulo *n*)**.

Exercices

Exercice 1
On donne la séquence d'instructions suivantes :
F = CREER_FILE_VIDE()
ENFILER(F, 4)
ENFILER(F, 1)
ENFILER(F, 3)
N = DEFILER(F)
ENFILER(F, 8)
ENFILER(F, 9)
N = DEFILER(F)

1) Donner le contenu de la file F à chaque étape, en la représentant sous forme d'un tuple, et le contenu de la variable N.
2) On décide de représenter la file F par un tableau de 7 éléments, comme vu plus haut dans le cours.
a) Quelle est la taille maximale de la file ?
b) Représenter le contenu du tableau à chaque étape.

Exercice 2

On suppose que l'on a déjà une file F1 qui contient les éléments suivants saisis dans l'ordre alphabétique : F1 = ('A', 'B', 'C', 'D', 'E').
1) Quel est l'élément issu d'un défilage de F1 ?
2) Proposer une séquence d'instructions utilisant deux piles P1 et P2, permettant la saisie d'affilée des 5 éléments 'A', 'B', 'C', 'D' et 'E', puis de sortir ces éléments comme s'ils sortaient d'une file.

Exercice 3

Implémenter en Python les opérations classiques sur les files à l'aide d'un tableau que l'on "simulera" à l'aide d'une liste. On supposera que le tableau a une taille fixe, par exemple 10. On gèrera ce tableau de façon circulaire : dès que les indices de tête ou de queue dépassent la longueur du tableau, ils repartent au début du tableau.

Exercice 4

Proposer une implémentation des opérations classiques de la file à l'aide des méthodes .pop() et .append() du type liste de Python. Une file vide sera représentée par la liste vide. On pourra se donner arbitrairement une taille maximale pour la file, initialisée dans une variable globale, par exemple MAX = 10.
On rappelle que, si F est une liste Python, L.pop(0) retourne l'élément d'indice 0 de F et le supprime de la liste.

Exercice 5 : Conversion d'une expression infixée en postfixée

On ne considère ici que les expressions numériques contenant les quatre opérations élémentaires (+, −, *, /).
On souhaite ici convertir une expression arithmétique infixée (avec des parenthèses) en une expression postfixée (notation polonaise inversée (NPI), sans parenthèses).
Par exemple, l'expression infixée :

7 * (((13 + 22) – 15) / 5)

sera traduite en :

7 13 22 + 15 – 5 / *

Dans l'expression infixée, chaque opération entre deux opérandes sera nécessairement mise entre parenthèses, car on ne tiendra pas compte ici de la priorité des opérations les unes sur les autres.
Ainsi, une expression telle que :

3 * 2 + 4

sera écrite : (3 * 2) + 4.

Le programme lit l'expression infixée comme une suite de caractères, et range le résultat de la conversion dans une file qui s'affichera.

Ce programme utilise une pile (**pile_operateurs**) pour gérer les opérateurs, et une file (**file_expression**) pour conserver l'expression postfixée.

Les **valeurs numériques** lues sont directement rangées dans la file, alors que les **opérateurs** sont traités afin d'apparaitre dans la file *après* leurs opérandes. Ils sont d'abord empilés (mis de côtés) dans **pile_operateur**, puis dépilés pour être rangés dans la file, après que les deux opérandes sont enfilées. Cela se produit quand la parenthèse fermante est rencontrée. La file contient les différentes valeurs et les opérateurs.

Quand la fin de ligne de l'expression infixée est atteinte, tous les opérateurs qui restent encore dans la pile sont dépilés et rangés dans la file.

Pour afficher l'expression postfixée, on défile un par un tous les éléments de la file et on les concatène en les séparant par un espace.