RÉSUMÉ 11 - TYPE ABSTRAIT LISTE

Lien vers l'activité : Type abstrait Liste
www.infoforall.fr - Dernière modif. : 17 10 2020
Ceci est un résumé global des activités 11 à 14 sur les LISTES.

11.1 - IMPLÉMENTATION : DE L'ABSTRAIT VERS LE CONCRET

Les **algorithmes** manipulent des **types abstraits** de données. L'un et l'autre ne sont que des idées qui vont nécessiter une **implémentation** en machine pour pouvoir fonctionner.
L'**implémentation** est le nom qu'on donne au passage de l'abstrait vers le concret.

- un **algorithme** est implémenté en tant que **programme** dans un langage réel
- un **type abstrait** de données est implémenté en tant que **structure** de données dans un langage réel

11.2 - TYPE ABSTRAIT DE DONNÉES

- un **principe général d'organisation** des données
- un ensemble minimum de **fonctions d'interface** qui permettent à un algorithme d'interagir avec les données abstraites.

Toute organisation compatible avec les deux points pourra porter le nom du type abstrait.

11.3 - STRUCTURE DE DONNÉES

La **structure de données** est une structure codée dans un langage particulier.
Si elle possède des fonctions réelles qui correspondent aux **fonctions d'interface** imposées par le type abstrait, on peut dire qu'une structure de données implémente un type abstrait.
On notera que :

- il existe toujours **plusieurs façons** d'implémenter un type abstrait
- La **structure de données** réelle peut faire plus que l'interface de base
- Deux structures implémentant un même type ont certainement des performances différentes sur certaines actions
- L'utilisateur n'a pas à utiliser et manipuler le code interne de la structure : il n'a besoin que de connaître les fonction d'interface. Si le code interne change, l'utilisateur n'aura donc pas besoin de réecrire ses programmes.

11.4 - TYPE ABSTRAIT LISTE

ORGANISATION GÉNÉRALE

La LISTE est vue comme une **séquence ordonnée et finie** d'éléments. On considère qu'on peut lire, modifier, insérer et supprimer à n'importe quelle place d'une **LISTE**.
VOCABULAIRE : On peut donc voir la **LISTE** comme une succession d'éléments, dont le premier élément se nomme la **TETE**. Le reste des éléments se nomme la **QUEUE**.
On peut donc considérer qu'une LISTE est :

- soit l'ensemble vide
- soit une TETE (un élément de la liste) suivie d'une QUEUE (une autre liste)

EXEMPLE

On peut donc représenter la suite 5, 15, 25 comme on le désire :
`Tete --> 5 --- 15 --- 25 --- None` ou encore `(5, (15, 25, () ) )`
Sur ces exemples, la tête est `5` et la queue est `15, 25`.

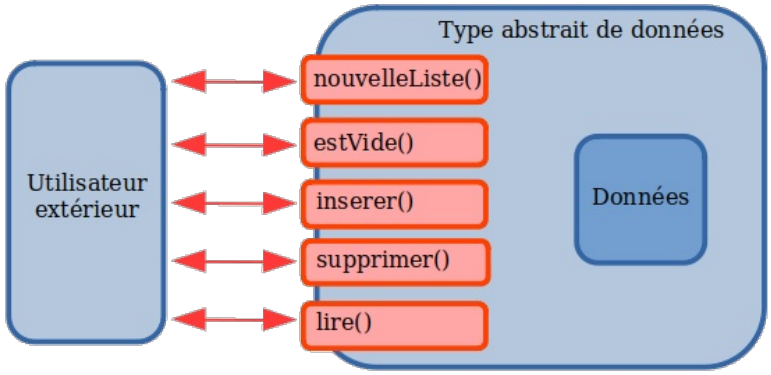
FONCTIONS D'INTERFACE

Une LISTE doit posséder au moins les fonctions ci-dessous :

- créer une nouvelle liste et stocker son identifiant-mémoire avec une fonction
  - `nouvelleListe() ->Liste`
- tester si une Liste est vide (ne possède pas de tête) avec une fonction
  - `estVide(lst:Liste) -> bool`
- insérer un nouvel élément **x** dans la liste **lst** avec une fonction d'insertion dont il existe de nombreuses variantes possibles :
  - `insererPosition(x:Elt lst:Liste, index:int) -> Liste` pour une version non-mutable
  - `insererPosition(x:Elt lst:Liste, index:int) -> None` pour une version mutable
  - `insererTete(x:Elt lst:Liste) -> Liste` pour une version non mutable n'agissant que sur la Tête
  - ou une autre insertion (à la fin, au milieu...)
- supprimer un élément, soit à une position précise, soit celui qui a une valeur précise... Il existe de nombreuses variantes. Exemple :
  - `supprimerTete(lst:Liste) -> Liste` pour une Liste non-mutable
  - `supprimerTete(lst:Liste) -> None` pour une Liste mutable ou même
  - `supprimerTete(lst:Liste) -> Elt` pour une Liste mutable : cette version renvoie l'ancienne Tête par exemple.
- lire le contenu d'un élément **x** dans la liste **lst** avec une fonction de lecture dont il existe de nombreuses variantes possibles :
  - `lirePosition(lst:Liste, index:int) -> Elt`
  - `lireTete(lst:Liste) -> Elt`

- `lirePositionCourante(lst:Liste) -> Elt`

Il existe bien d'autres fonctions d'interface mais elles ne sont pas nécessaires pour définir le type abstrait LISTE.

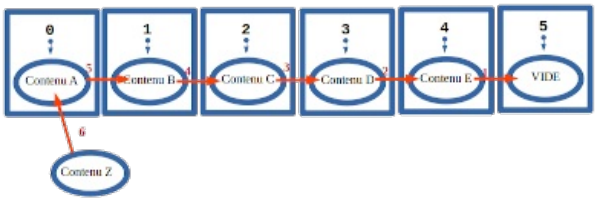


On notera que même l'interface la plus basique (action sur la tête) permet en réalité de construire les fonctions plus élaborées.

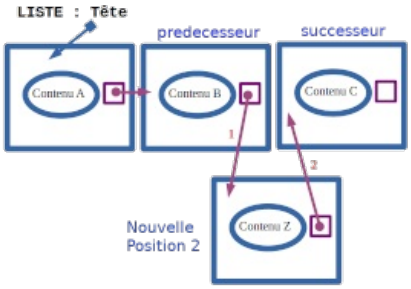
11.5 4 IMPLÉMENTATIONS COURANTES

- **tuple** de la forme **(tete, queue)** (voir activité 12)
  - Avantage : simplicité et très grande performance si on se limite à des actions sur la tête.
  - Désavantage : coût linéaire pour la plupart des opérations dans le pire des cas

- **tableau** : les différents éléments sont donc disposés les uns à côté des autres (voir activité 13)
  - Avantages :
    - coût en lecture / modification constant
    - place mémoire réduite : sa "case" de la liste ne contient finalement que la valeur de la case
  - Désavantage :
    - coût linéaire en insertion / suppression dans le pire des cas
    - concaténer deux listes-tableaux est une tâche coûteuse.



- **liste chaînée** : les différents éléments sont reliés entre eux par des relations de type prédécesseur - successeur (voir activité 14)
  - Avantage :
    - coût constant en insertion / suppression pour de nombreux cas (à voir en fonction de l'implémentation)
    - possibilité de créer une concaténation de deux listes chaînées à coût constant.
  - Désavantage :
    - coût linéaire en lecture / modification dans le pire des cas.
    - place mémoire plus importante qu'un **tableau** : chaque case de la liste contient la valeur de la case mais également des indications sur le successeur et parfois le prédécesseur.
  - Implémentation en Python de la liste chaînée : comme la liste chaînée n'existe pas nativement en Python, il faut créer cette structure. Nous avons fait un TP en utilisant un objet Liste et des objets Cellule. Dans une autre activité, nous avons également vu qu'on pouvait utiliser de simples tableaux.



- Type natif `list` de Python : Python ne possède pas d'implémentation de la liste chaînée : son type natif `list` est en réalité un tableau dynamique sur lequel on a rajouté de nombreuses fonctionnalités lui permettant de ressembler à une liste :
- la méthode `append()` permet de rajouter à la fin, sur place. La méthode ne renvoie rien.
- la méthode `pop(x)` permet de supprimer l'élément en position x. La méthode renvoie d'ailleurs l'élément supprimé.

CHOIX D'UNE IMPLÉMENTATION DE LISTE

Vous devez savoir que le choix se fait en fonction des besoins de l'algorithme.