3_Structure_de_données_linéaires

November 9, 2021

1 Terminale - spécialité Numérique et Sciences Informatiques

3. Structure de données linéaire

Objectifs: * Introduire le concept de structure de données abstraite * Découvrir les principales structures de données linéaires * Réaliser différentes implémentations de ces structures

1.1 Structure de données abstraite

En théorie de l'informatique, on étudie des algorithmes définient en langage naturel ou en pseudo-code en dehors de la considération des choix d'implémentation et donc en dehors des spécificités techniques d'un langage de programmation.

Ces algorithmes s'appuient sur des modèles théoriques d'organisation et d'interactions de données, les structures de données abstraites.

Une **structure de données abstraite** définit :

- une **organisation** des données:
- linéaire: collection unidimensionnelle
- accès par clef: dictionnaire
- hiérarchique: arbre
- relationnelle: graphe
- des **interfaces** (primitives) pour la manipuler:
- création
- ajout d'une donnée
- modification d'une donnée
- suppression d'une donnée

On va étudier ici des structures de données abstraites **linéaires** classiques et dans de prochains chapitres les autres.

1.2 Liste

Une liste (list ou sequence) est une structure de données abstraite linéaire: * définie itérativement par une suite finie et ordonnée d'éléments: * $(x_1 ; x_2 ; ... ; x_n)$ * on passe d'un élément à son successeur séquentiellement * Les éléments se suivent les uns les autres sur une dimension, c'est bien une structure linéaire. * ou définie récursivement par une liste est: * Soit une liste vide * Soit un couple $(t ; q)$ où: * t est la tête (head): le premier élément de la liste * q est la queue de la liste (tail): la liste privée de son premier élément * avec des interfaces (primitives) pour la manipuler: * `creeListe()`: crée une liste vide * `insereTeteListe(liste, valeur)`: ajoute la valeur spécifiée en tête de liste * `supprimeTeteListe(liste)`: supprime la tete de la liste et renvoie sa valeur * `teteListe(liste)`: renvoie la tête de liste (élément) * `queueListe(liste)`: renvoie la queue de liste (liste) * `estListeVide(liste)`: renvoie `True` si la liste est vide, `False`sinon * `tailleListe(liste)`: renvoie le nombre d'élément de la liste(surcharge de`__len__(self)` en objet) * et bien d'autres: recherche/modifie/insère/supprime élément en ième position, concatène, inverse, compare, …

1.2.1 Mission 3.1.

> Utiliser les primitives pour créer la liste : $(1 ; 2 ; 3 ; 4)$
> Utiliser les primitives pour modifier la liste pour obtenir: $(2 ; 3 ; 1 ; 4)$
> Représenter l'état de la liste après chaque appel de primitive en précisant sa tête et sa queue.

On peut implémenter simplement une liste avec un **tableau** (array)

Un tableau est une structure de données représentant une zone contigüe en mémoire de taille fixée. Chaque élément du tableau est repéré par son **indice**. L'accès en mémoire est alors quasi-direct (simple calcul d'adresse à partir de l'adresse de base)

Les **listes Python** sont en fait des tableaux dynamiques, c'est à dire des tableaux dont la taille est modifiée dynamiquement (pour s'adapter au contenu).

Attention donc à bien distinguer les listes telles que définies ici en tant que structure de données abstraite et les listes Python qui sont un choix d'implémentation spécifique.

1.2.2 Mission 3.2.

> Proposer une implémentation des listes avec un tableau de taille fixe:
> * en pseudo-code * en Python avec le paradigme fonctionnel * en Python avec le paradigme Objet
>
> Identifier les forces et faiblesses de cette implémentation.
>
> Vous pouvez:
> - utiliser la 1ère case du tableau pour stocker le nombre d'élément
> - identifier les cases sans élément avec une valeur nulle (None)
> - considérer que la tête de liste est le dernier élément du tableau

[ ]:
```
# Votre code ici
```

1.3 Liste chaînée:

Une liste chaînée (linked list) est une structure de données abstraite linéaire: * définie itérativement par une suite finie et ordonnée d'éléments: * dont chaque élément pointe vers le suivant: $x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n$ * chaque élément contient donc deux informations: la valeur et le pointeur (adresse) de l'élément suivant * on passe d'un élément à son successeur séquentiellement * Les éléments se suivent les uns les autres sur une dimension, c'est bien une structure linéaire. * ou définie récursivement par une liste chaînée est: * Soit une liste vide * Soit un couple $(t\,;q)$ où: * t est la tête (head): le premier élément de la liste chaînée * q est la queue de la liste chaînée (tail): la liste chaînée privée de son premier élément * avec des interfaces (primitives) pour la manipuler: * `creeListeChainee()`: crée une liste chaînée vide * `insereTeteListeChainee(liste, valeur)`: ajoute la valeur spécifiée en tête de la liste chaînéee * `supprimeTeteListeChainee(liste)`: supprime la tete de la liste chaînéee et renvoie sa valeur * `teteListeChainee(liste)`: renvoie la tête de liste chaînéee (type valeur) * `queueListeChainee(liste)`: renvoie la queue de liste chaînéee (liste chaînée) * `estListeChaineeVide(liste)`: renvoie `True` si la liste chaînéee est vide, `False`sinon * `tailleListeChainee(liste)`: renvoie le nombre d'élément de la liste chaînéee (surcharge de`__len__(self)` en objet) * `elementPosition(liste, i)`: renvoie l'élément en ième position (surcharge de `__getitem__(self, i)` en objet) * `insereValeurPosition(liste, valeur, i)`: insère la valeur spécifiée en ième position (methode `insert(self, i)`en objet) * et bien d'autres: modifie/supprime élément en ième position, concatène, inverse, compare, …

1.3.1 Mission 3.3.

Utiliser les primitives pour créer la liste chaînée: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
Utiliser les primitives pour modifier la liste chaînée pour obtenir: $2 \rightarrow 3 \rightarrow 1 \rightarrow 4$
Représenter l'état de la liste chaînée après chaque appel de primitive en précisant sa tête.

1.3.2 Mission 3.4.

Proposer une implémentation des listes chaînées:
* en pseudo-code * en Python avec le paradigme fonctionnel * en Python avec le paradigme Objet

Identifier les forces et faiblesses de cette structure.

NB: Pour l'implémentation objet, implémenter au préalable la classe `Cellule` avec les attributs:
> `valeur` : la valeur de la cellule
> `suivant`: adresse de la cellule suivante éventuelle (`None` si dernière cellule de la liste chaînée)

1.4 Pile (LIFO):

Une pile (stack) est une structure de données abstraite linéaire: * définie itérativement par une suite finie et ordonnée d'éléments: * le dernier élément ajouté est le premier à sortir (mode LIFO: last In First Out) * cet élément est appelé le sommet (top) de la pile, on le représente généralement en haut de la pile (comme une pile d'assiette) * on ne peut accéder directement à la pile qu'à partir de son sommet: * sommetPile: pour accéder à la valeur du sommet de la pile (la pile n'est pas modifiée) * dépilement(pop): pour supprimer le sommet actuel de la pile (le nouveau sommet est alors l'élement précédent * empilement(push) pour insérer un nouveau sommet sur la pile * on passe d'un élément à son successeur séquentiellement (par dépilement) * Les éléments se suivent les uns les autres sur une dimension, c'est bien une structure linéaire. * ou définie récursivement par une pile est: * Soit une liste vide * Soit un couple (s ; q) où: * s est le sommet de la pile
* q est la queue de la pile (tail): la pile privée de son sommet * avec des interfaces (primitives) pour la manipuler: * `creePile()`: crée une pile vide * `empile(pile, valeur)`: ajoute la valeur spécifiée au sommet de la pile * `depile(pile)`: supprime le sommet de la liste chaînéee et renvoie sa valeur * `sommetPile(pile)`: renvoie le sommet de la pile (type valeur) * `queuePile(pile)`: renvoie la pile privée de son sommet (pile) * `estPileVide(pile)`: renvoie `True` si la pile est vide, `False`sinon * `taillePile(pile)`: renvoie le nombre d'élément de la pile (surcharge de`__len__(self)` en objet) * `elementPositionPile(liste, i)`: renvoie l'élément en ième position (surcharge de `__getitem__(self, i)` en objet) * `insereValeurPositionPile(liste, valeur, i)`: insère la valeur spécifiée en ième position (methode `insert(self, i)`en objet) * et bien d'autres: modifie/supprime élément en ième position, concatène, inverse, compare, …

1.4.1 Mission 3.5.

Utiliser les primitives pour créer la pile:
> 4
> 3
> 2
> 1

Utiliser les primitives pour modifier la pile pour obtenir: > 4
> 1
> 3
> 2

Représenter l'état de la pile après chaque appel de primitive en précisant son sommet

1.4.2 Mission 3.6.

Proposer une implémentation des piles:
* en pseudo-code * en Python avec le paradigme fonctionnel * en Python avec le paradigme Objet

Identifier les forces et faiblesses de cette structure.

NB: Pour l'implémentation objet, implémenter au préalable la classe `Cellule` avec les attributs:
> `valeur` : la valeur de la cellule
> `suivant`: adresse de la cellule suivante éventuelle (`None` sinon)

1.5 File (FIFO):

Une file (queue) est une structure de données abstraite linéaire: * définie itérativement par une suite finie et ordonnée d'éléments: * le premier élément ajouté est le premier à sortir, comme dans une file d'attente (mode FIFO: First In First Out) * cet élément est la tête file (front), on le représente généralement à gauche * l'élément après le dernier élément de la file est le bout de file (rear), on le représente généralement à droite * on ne peut accéder diaccéderrectement à la file qu'à partir de sa tête et son bout: * teteFile: pour accéder à la valeur en tête de file (la file n'est pas modifiée) * défilement(dequeue): pour supprimer la tête actuelle de la file (la nouvelle tête est alors l'élement entré juste après) * enfilement(enqueue) pour insérer un nouvel élément en bout de file * on passe d'un élément à son successeur séquentiellement (par défilement) * Les éléments se suivent les uns les autres sur une dimension, c'est bien une structure linéaire. * ou définie récursivement par une file est: * Soit une file vide * Soit un couple (t ; q) où: * t est la tête de la file
* q est la queue de la file (tail): la file privée de son sommet * avec des interfaces (primitives) pour la manipuler: * `creeFile()`: crée une file vide * `enfile(file, valeur)`: ajoute la valeur spécifiée en tête de file * `defile(file)`: supprime la tête de la file et renvoie sa valeur * `teteFile(file)`: renvoie la tête de file (type valeur) * `queueFile(file)`: renvoie la file privée de sa tête (file) * `estFileeVide(file)`: renvoie `True` si la file est vide, `False`sinon * `tailleFile(file)`: renvoie le nombre d'élément de la file (surcharge de`__len__(self)` en objet) * `elementPositionFile(file, i)`: renvoie l'élément en ième position (surcharge de `__getitem__(self, i)` en objet) * `insereValeurPositionFile(file, valeur, i)`: insère la valeur spécifiée en ième position (methode `insert(self, i)`en objet) * et bien d'autres: modifie/supprime élément en ième position, concatène, inverse, compare, …

1.5.1 Mission 3.7.

Utiliser les primitives pour créer la file: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
Utiliser les primitives pour modifier la file pour obtenir: $2 \rightarrow 3 \rightarrow 1 \rightarrow 4$
Représenter l'état de la liste après chaque appel de primitive

1.5.2 Mission 3.8.

Proposer une implémentation des files:
* en pseudo-code * en Python avec le paradigme fonctionnel * en Python avec le paradigme Objet

Identifier les forces et faiblesses de cette implémentation.

NB: Pour l'implémentation objet, implémenter au préalable la classe `Cellule` avec les attributs:
> `valeur` : la valeur de la cellule
> `suivant`: adresse de la cellule suivante éventuelle (`None` sinon)

1.5.3 Mission 3.9.

Proposer une implémentation d'une File avec deux Piles (en paradigme objet)

1.6 Récursivité

1.6.1 Mission 3.10.

Proposer une implémentation récursive en paradigme objet des méthodes de calcul de longueur et d'accès aux éléments d'une liste chaînée, d'une pile et d'une file.

NB: à traiter après le cours sur la récursivité

1.7 En option (++)

1.7.1 Mission 3.11.

Implémenter une Pile avec deux Files

1.7.2 Mission 3.12.

Implémenter un analyseur syntaxique html