UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Faculté des Sciences et d'Ingénierie

Récursivité sur les listes

CTD 6 : Listes chaînées et Récursivité

Algorithmique

Semestre 2 - 2019-2020

Introduction - Motivation et rappels

Motivation : pour la récursivité, on va raisonner sur la structure liste chaînée, qui est une structure de données **récursive**

Abstraction de données

De manière plus générale on appelle **abstraction de données** la définition d'un type de données avec la description des **opérations qu'il est possible de lui appliquer**, indépendamment de la manière dont cela sera programmé.

La **description de chaque opération** précise :

- Ses entrées.
- Les préconditions qui doivent être vérifiées pour que l'opération s'exécute correctement.
- Ses sorties.
- Les post conditions vraies après exécution de l'opération.

Structure de données

Une **structure de données** est décrite en présentant :

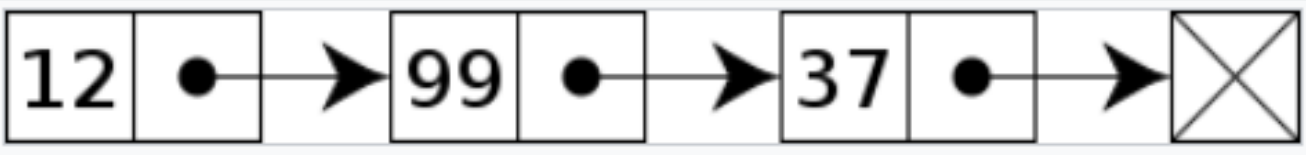
- L'**abstraction de données** qui la définit.
- Son **implémentation**, c'est-à-dire sa mise en oeuvre matérielle.

Rappel/Exemple : Tableau

- description du type
  - Le **nombre d'éléments** du tableau est **fixe**.
  - Tous les éléments du tableau sont de **même type**.
  - Chaque élément est repéré par sa position dans le tableau, son indice.
- opérations
  - création : précision du type des éléments et de la taille du tableau
  - accès à un élément `tableau[i]` : élément d'indice `i` dans `tableau`
- implémentation en python avec les listes
  - interdiction d'utiliser les opérations qui modifient la taille (del, append, insert...)
  - interdiction d'utiliser les opérations qui ne maintiennent pas le type unique

Liste chaînée - Description du type

- nombre d'élément **variable** (évolution dynamique)
- séquence d'élements de même type
- structure **récursive** : on distingue l'élément **en tête** et le reste de la liste (**la queue**)



Une liste simplement chaînée, composée de trois éléments ayant respectivement la valeur : 12, 99 et 37.

Liste chaînée - Abstraction : Opérations

- opérations de modification
  - création (d'une liste vide)
  - insertion en tête
  - suppression en tête
- opérations d'accès
  - accès au premier élément
  - accès à la queue de la liste
  - longueur de la liste
  - liste vide

les insertions et suppressions en milieu, en fin ne sont pas autorisées

Liste chaînée - Structure de données (implémentation python)

- à l'aide du type `list`
- accès autorisé à l'élément 0 (`liste[0]`)
- interdiction à l'accès direct à l'aide d'un indice (ex : `liste[3]` est interdit)
- accès à la queue de liste par "tranche" (ex : `liste[1:]`)
- liste.insert(i,elem) autorisé seulement pour i qui vaut 0
- seulement del liste[0] autorisé (ou `liste.remove(liste[0])`)

Attention : l'utilisation des tranches en python réalise une copie de liste... Il serait plus avisé de travailler sur les indices plutôt que sur les tranches, mais alors on ne respecterait plus l'abstraction "liste chaînée".

Liste chaînée - Structure de données (autres implémentations)

- struct en C
- `deque` en python
- enregistrements
- en caml (h : :t)
- à l'aide d'un tableau (taille très supérieure pour décaler les éléments ajoutés...)
- listes doublement chaînées (pour accès rapide au dernier élément) ...

Récursivité - Introduction

Une solution récursive est particulièrement adaptée quand on doit traiter des structures de données récursives.

Les listes (et d'autres structures de données que nous verrons plus tard) sont des structures récursives. En effet, on peut dire d'une liste qu'elle est :

- soit vide ;
- soit construite à partir d'un élément et d'une autre liste (ajout d'un élément dans la liste).

Récursivité - Introduction

On peut donc en déduire que le traitement d'une liste suivra (le plus souvent) le schéma récursif suivant :

- La liste vide correspondra au cas d'arrêt (cas trivial, de base) de la récursivité.
- La liste résultant de l'ajout de l'élément element à la liste reste_liste correspondra au cas de calcul récursif avec la taille du problème diminuée si on fait le traitement sur reste_liste.

Bien entendu, il peut y avoir des traitements où nous aurons besoin d'autres cas d'arrêt (liste à un seul élément, par exemple).

Récursivité - Introduction

Jusqu'à présent, quand nous avons traité des listes, nous avons le plus souvent parcouru toute la liste, élément par élément pour faire le traitement adéquat. Suivant le modèle de solution choisi, le parcours des listes pouvait se faire :

- de la gauche vers la droite,
- de la droite vers la gauche,
- à partir du milieu,
- ...

Pour la construction et le traitement récursif des listes, il est généralement admis de faire le parcours de gauche à droite, c'est-à-dire que l'ajout d'un élément dans une liste se fait en tête (mais on aurait pu choisir un autre mode de solution).

Récursivité - Avertissement

Dans cette partie du cours, nous voulons montrer comment exprimer une solution sans se préoccuper de problème d'efficacité.

Pour ne pas "ajouter" de difficultés, nous avons pris le parti de ne pas modifier les listes données en argument des fonctions. Il est bien évidemment qu'on pourrait donner d'autres(s) version(s) des fonctions en modifiant les listes données en argument pour économiser de la mémoire.

Démarche pour écrire une fonction récursive sur les listes

- Chercher le(s) cas simple(s), c'est-à-dire celui(ceux) qui ne nécessite(nt) pas de rappeler récursivement la fonction :
  - Le plus souvent liste vide.
  - Eventuellement liste à un élément.
- Chercher le sous-problème récursif (sous-problème de taille réduite par rapport au problème) pour rappeler la fonction récursive pour ce sous-problème :
  - la liste privée d'un élément,
  - le plus souvent privée de son 1^{er} élément.

Récursivité - Démarche

"Vérifier/Constater" que la fonction termine, c'est-à-dire qu'on atteint au moins un cas d'arrêt :

- le plus souvent, on a diminué la taille de la liste en enlevant un élément de celle-ci.

Récursivité - Exemples

Ecrire la fonction récursive longueur_liste qui donne le nombre d'éléments d'une liste.

2 cas à traiter :

- Cas d'arrêt : la liste est vide
  - → On retourne 0.
- Cas récursif : la liste n'est pas vide
  - → On retourne la longueur de la liste privée d'un élément (par exemple, le 1^{er}) + 1.

Fonction récursive longueur_liste

```
1 def longueur_liste(liste):
2   if (liste == []):
3     return 0
4   else:
5     return (1 + longueur_liste(liste[1:]))
```

Exercices - Produit des élements

Exercice 1

Ecrire la fonction récursive produit_elements_de_la_liste qui, étant donnée une liste d'entiers, calcule le produit des éléments de la liste.

Par convention, le produit des éléments d'une liste vide sera 1.

Exercices - Dernier élement

Exercice 2

Ecrire la fonction récursive dernier_element_de_la_liste qui, étant donnée une liste, retourne le dernier élément de la liste s'il existe.

Exercices - Appartenance

Exercice 3

Ecrire la fonction récursive appartient_element_a_la_liste qui, étant donnés un élément et une liste, retourne True si l'élément appartient à la liste et False sinon.

Exercice 4

Ecrire la fonction récursive substituer_element_par_un_autre qui, étant donnés deux éléments, element_a_substituer et nouvel_element, et une liste, construit une nouvelle liste (c'est à dire que la liste initiale ne sera pas modifiée) où toutes les occurrences de l'élément à substituer auront été remplacées par le nouvel élément.

Exercices - Palindrome

Exercice 5

Ecrire la fonction récursive est_palindrome qui, étant donnée une **liste doublement chaînée**, retourne True si la liste est un palindrome et False sinon.
Pour cet exercice, on autorise donc l'accès au dernier élément (`liste[-1]`) et l'utilisation de tranche sans restriction au classique `liste[1:]` utilisé jusqu'ici.

[] [1] [1, 2, 2, 1] [1, 2, 3, 2, 1] sont des palindromes.

Exercices - Moyenne ET nombres supérieurs à la moyenne

Exercice 6

L'objectif est d'écrire une fonction récursive qui calcule la moyenne d'une liste NON VIDE ainsi que le nombre d'éléments de cette liste qui sont strictement supérieurs à cette moyenne.

Pour cela la fonction va calculer la moyenne lors des appels récursifs et comptera le nombre d'éléments plus grands lors du retour des appels récursifs.

Fonction récursive nb_superieurs

On vous demande donc de créer une fonction appelée nb_superieurs prenant trois arguments :

- la liste restant à parcourir,
- la somme des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel),
- le nombre des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel).

Bilan

La récursivité est une solution puissante et finalement assez simple pour exprimer la résolution d'un problème en isolant les cas simples, où le résultat est directement trouvé, et les cas où le problème est découpé en sous-problèmes plus petits (donc plus simples à résoudre) qui seront traités avec des appels récursifs.

Bien évidemment, il faut que le problème s'y prête bien (structures de données naturellement récursives qui s'expriment en fonction d'elles mêmes), ce qui est souvent le cas.

Bilan

La récursivité est une solution à un problème qui s'exprime facilement et assure souvent une bonne lisibilité. Cependant, on lui reproche son inefficacité à cause du nombre d'appels récursifs qu'elle peut engendrer (exemple de Fibonacci . . .).

Il existe des méthodes qui, pour certaines fonctions récursives, permettent de palier le problème en "dérécursivant" la solution pour avoir une solution dite "récursive terminale" qui sera ensuite transformée en boucle while.