ITI3 mercredi 7 janvier 2026

Algorithmique avancée et Programmation C

Durée : *2h00*
Documents autorisés : **AUCUN**

Remarques :

— Veuillez lire attentivement les questions avant de répondre ;

— Pour chaque section comportant des questions, il est indiqué, entre parenthèses, le nombre d'attendus d'apprentissages disciplinaires (AAD) évalués par ces questions. Attention ce n'est pas parce qu'il y a peu d'AAD dans une section que la part de cette section dans le calcul de la note finale est faible.

1 QCM (3 AAD)

Répondez au qcm ci joint (à rendre avec votre copie). La note de chaque question peut varier de -1 à $+1$. Il n'y a qu'une seule bonne réponse par question : la note d'une mauvaise réponse est de -1 et d'une bonne réponse de $+1$.

2 Une nouvelle collection : DictionnaireOrdonne (16 AAD)

Un dictionnaire ordonné est un dictionnaire dans lequel l'ordre d'insertion des couples (clé, valeur) est préservé au regard des clés. Ainsi l'opération d'obtention des clés, ne retourne pas un ensemble de clés, mais une liste de clés (l'ordre de cette liste est celui de l'insertion des couples (clé,valeur)). Aux opérations classiques du TAD Dictionnaire, nous ajoutons une nouvelle opération qui a pour objectif de mettre à la fin une clé préalablement présente dans le dictionnaire.

2.1 Question de cours (3 AAD)

1. Analyse : rappelez le TAD Dictionnaire vu en cours sans la partie axiome ;
2. Conception préliminaire : donnez les signatures des fonctions ou procédures correspondantes.

2.2 Analyse (5 AAD)

Donnez le TAD `DictionnaireOrdonnee` (DO) sans la partie axiome.

2.3 Conception préliminaire

Donnez les signatures des fonctions ou procédures correspondantes.

2.4 Conception détaillée (7 AAD)

L'excellent article « How Does Python's OrderedDict Maintain Order ? » (`https://www.piglei.com/articles/en-why-is-python-ordereddict-ordered/`) explique comment ce type est conçu en python afin de garantir les opérations d'ajout et de suppression en temps, identique à celui du dictionnaire classique (utilisant des tables de hachage). Cette conception repose sur l'utilisation conjointe :

1. d'un dictionnaire classique (non ordonné) pour stocker les couples (clé, valeur) ;
2. d'une liste doublement chaînée de clés pour stocker dans l'ordre d'insertion les clés insérées ;
3. d'un dictionnaire classique (toujours non ordonné) qui associe à une clé le pointeur du nœud de la liste doublement chaînée la stockant.

La figure 1 issue de cet article illustre ce principe.

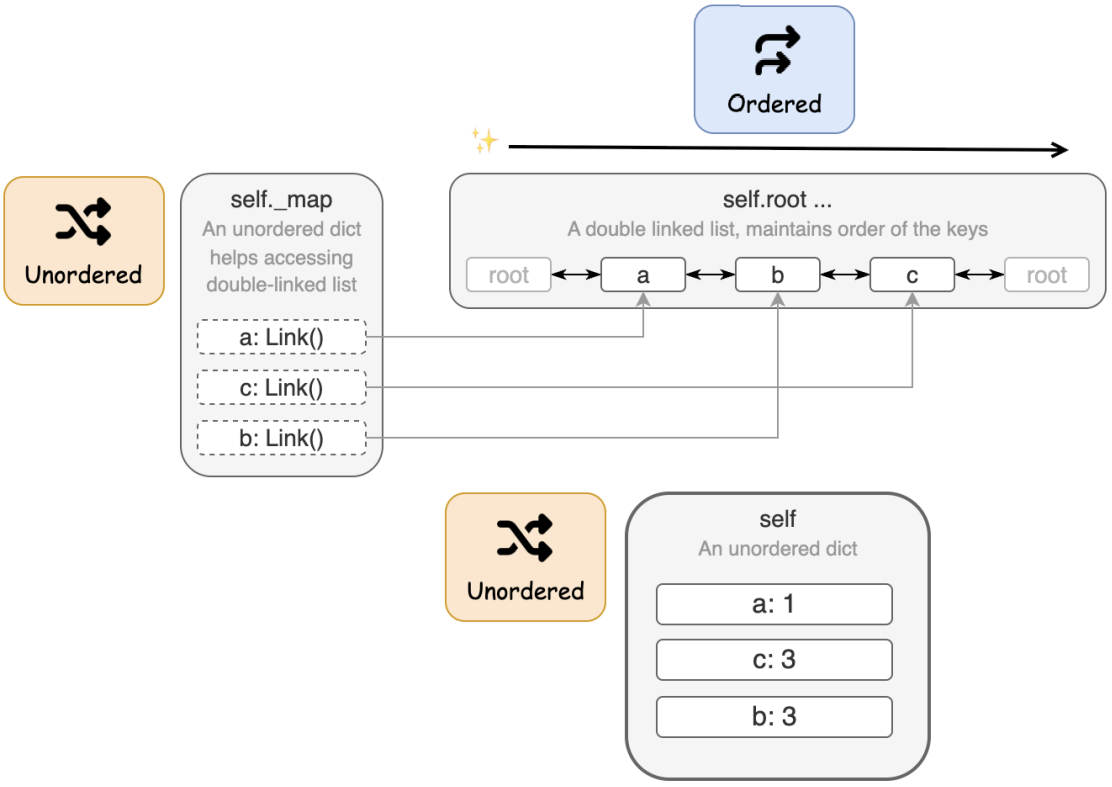

FIGURE 1 – Conception d'un dictionnaire ordonné

1. Expliquez pourquoi avec cette conception, en utilisant en plus une petite astuce, les opérations d'ajout et de suppression ont des complexités en temps identiques à celles du type dictionnaire non ordonné.
2. Proposez une conception pour le type `DictionnaireOrdonne` ;
3. Donnez les algorithmes des fonctions/procédures d'ajout et de suppression dans un dictionnaire ordonné.

Annexe

La SSD `ListeDoublementChainee` (`LDC`) est conçu de la façon suivante :

Type LDC = ^ Noeud
Type Noeud = **Structure**

element : Element
listeSuivante : LDC
listePrecedente : LDC
finstructure

Pour simplifier son utilisation de ce type, nous avons les signatures de fonctions et procédures suivantes :

- **fonction** listeDoublementChainee **() :** LDC
- **fonction** estVide (l : LDC) **: Booleen**
- **procédure** inserer **(E/S** l : LDC,**E** element : Entier**)**
- **fonction** obtenirElement (l : LDC) **:** Entier
  ⌊**précondition(s)** $non(estVide(l))$
- **fonction** obtenirListeSuivante (l : LDC) **:** LDC
  ⌊**précondition(s)** $non(estVide(l))$
- **fonction** obtenirListePrecedente (l : LDC) **:** LDC
  ⌊**précondition(s)** $non(estVide(l))$
- **procédure** fixerElement **(E** l : LDC, e : Element**)** $non(estVide(l))$
- **procédure** fixerListeSuivante **(E** l : LDC, l' : LDC**)**
  ⌊**précondition(s)** $non(estVide(l))$
- **procédure** fixerListePrecedente **(E/S** l : LDC, l' : LDC**)**
  ⌊**précondition(s)** $non(estVide(l))$
- **procédure** supprimerNoeud **(E/S** l : LDC, **S** avant, apres : LDC**)**
  ⌊**précondition(s)** $non\ estVide(l)$
- **procédure** supprimer **(E/S** l : LDC**)**