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 ;
   Solution proposée:

Savoir et Savoir-faire évaluées
— AN302 : Formaliser sous forme de TAD une collection

Nom: Dictionnaire

Paramètre: Cle,Valeur

Utilise: **Booleen**,Ensemble

Opérations:

dictionnaire:	$\rightarrow$ Dictionnaire
ajouter:	Dictionnaire $\times$ Cle $\times$ Valeur $\rightarrow$ Dictionnaire
retirer:	Dictionnaire $\times$ Cle $\rightarrow$ Dictionnaire
estPresent:	Dictionnaire $\times$ Cle $\rightarrow$ **Booleen**
obtenirValeur:	Dictionnaire $\times$ Cle $\nrightarrow$ Valeur
obtenirCles:	Dictionnaire $\rightarrow$ Ensemble<Cle>

Préconditions: obtenirValeur(d,c): $estPresent(d, c)$

2. Conception préliminaire : donnez les signatures des fonctions ou procédures correspondantes.
Solution proposée:

Savoir et Savoir-faire évaluées

— CP003 : Choisir entre une fonction et une procédure
— CP004 : Concevoir une signature (préconditions incluses)

— **fonction** dictionnaire **() :** Dictionnaire
— **procédure** ajouter **(E/S** d : Dictionnaire, **E** cle : Cle, element : Valeur**)**
— **procédure** retirer **(E/S** d : Dictionnaire, **E** cle : Cle**)**
— **fonction** estPresent **(**d : Dictionnaire, cle : Cle**) : Booleen**
— **fonction** obtenirValeur **(**d : Dictionnaire, cle : Cle**) :** Valeur estPresent(d, cle)
— **fonction** obtenirCles **(**d : Dictionnaire**) :** Ensemble<Cle>

2.2 Analyse (5 AAD)

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

Savoir et Savoir-faire évaluées

— AN201 : Identifier les dépendances d'un TAD
— AN202 : Définir des TAD génériques
— AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
— AN204 : Formaliser des opérations d'un TAD
— AN205 : Formaliser les préconditions d'une opération d'un TAD

Nom: DictionnaireOrdonnee (DO)
Paramètre: Cle,Valeur
Utilise: **Booleen**,Liste
Opérations:

dictionnaire:	$\rightarrow$ DO
ajouter:	DO $\times$ Cle $\times$ Valeur $\rightarrow$ DO
retirer:	DO $\times$ Cle $\rightarrow$ DO
estPresent:	DO $\times$ Cle $\rightarrow$ **Booleen**
obtenirValeur:	DO $\times$ Cle $\nrightarrow$ Valeur
obtenirCles:	DO $\rightarrow$ Liste<Cle>
mettreALaFin:	DO $\times$ Cle $\nrightarrow$ DO

Préconditions: obtenirValeur(d,c): $estPresent(d, c)$
mettreALaFin(d,c): $estPresent(d, c)$

2.3 Conception préliminaire

Donnez les signatures des fonctions ou procédures correspondantes.
Solution proposée:

Savoir et Savoir-faire évaluées

- CP003 : Choisir entre une fonction et une procédure
- CP004 : Concevoir une signature (préconditions incluses)

- **fonction** dictionnaireOrdonne **() :** DO
- **procédure** ajouter **(E/S** do : DO, **E** cle : Cle, element : Valeur**)**
- **procédure** retirer **(E/S** do : DO, **E** cle : Cle**)**
- **fonction** estPresent **(**do : DO, cle : Cle**) : Booleen**
- **fonction** obtenirValeur **(**do : DO, cle : Cle**) :** Valeur
  ⌊**précondition(s)** estPresent(do,cle)
  estPresent(do, cle)
- **fonction** obtenirCles **(**do : DO**) :** Liste<Cle>
- **procédure** mettreALaFin **(E/S** do : DO, **E** cle : Cle**)**
  ⌊**précondition(s)** estPresent(do,cle)

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.

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.
   Solution proposée:

   Savoir et Savoir-faire évaluées

   - AN004 : Comprendre et appliquer des consignes algorithmiques sur un exemple

   À l'image de la conception d'une file à l'aide d'une liste chaînée vue en cours, l'astuce consiste à référencer à l'aide de deux variables le début et la fin de la liste doublement chaînée. Ainsi l'ajout d'un élément implique :
   - la création d'un nœud n à la fin de la liste doublement chainée cles, en complexité $O(1)$ grâce au référencement son dernier élément ;
   - l'ajout du couple (cle, n) au dictionnaire positions, donc avec la même complexité
   - l'ajout du couple (clé, valeur) au dictionnaire valeurs, donc avec la même complexité.

   De plus grâce au dictionnaire qui stocke les couples (cle, LDC), la suppression de la LDC de la clé a pour complexité celle d'accès à un élément d'un dictionnaire non ordonné (car la suppression du noeud de la LDC est en O(1)). L'opération de suppression d'une clé dans un dictionnaire ordonné est donc identique à la suppression d'une clé d'un dictionnaire non ordonné.
2. Proposez une conception pour le type `DictionnaireOrdonne` ;
   Solution proposée:

   Savoir et Savoir-faire évaluées

   - CD901 : Concevoir un type de données adapté à la situation en terme d'espace mémoire et d'efficacité

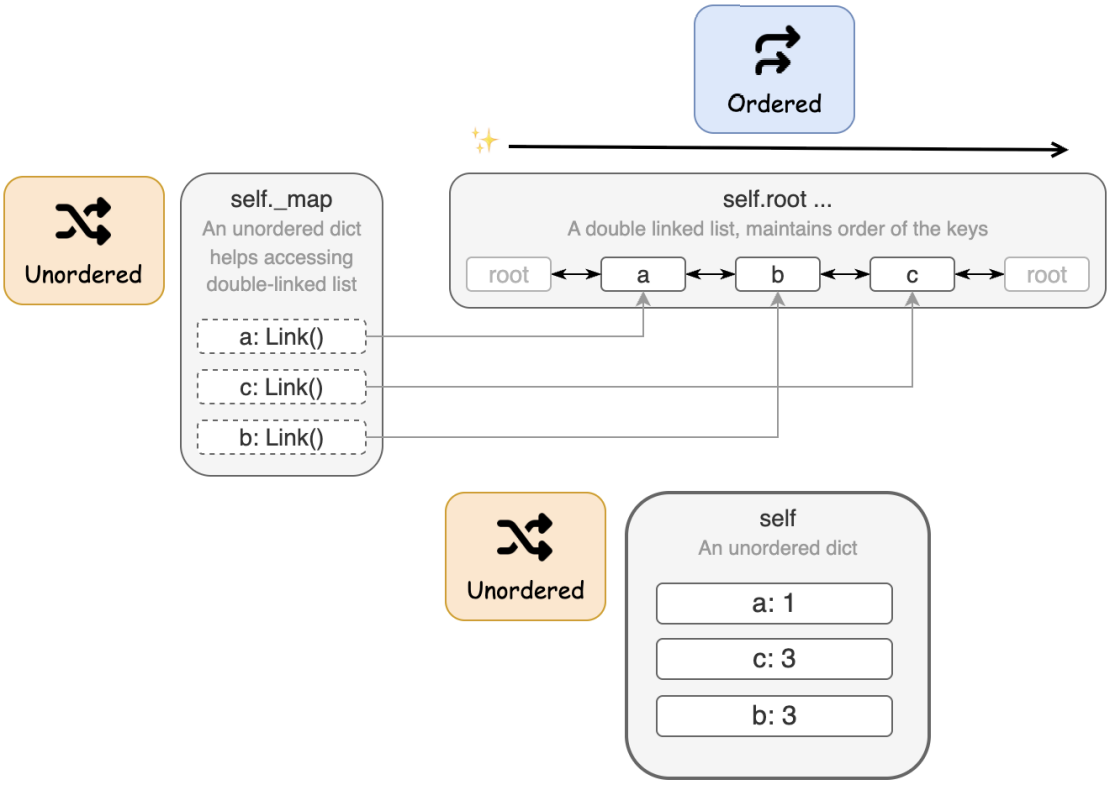

FIGURE 1 – Conception d'un dictionnaire ordonné

Type DictionnaireOrdonne = **Structure**
valeurs : Dictionnaire<Cle,Valeur>
clesDebut : ListeDoublementChainee<Cle>
clesFin : ListeDoublementChainee<Cle>
positions : Dictionnaire<Cle,ListeDoublementChainee>
finstructure

3. Donnez les algorithmes des fonctions/procédures d'ajout et de suppression dans un dictionnaire ordonné.
Solution proposée:

Savoir et Savoir-faire évaluées

- CD002 : En tant qu'utilisateur, respecter une signature
- CD003 : Utiliser le principe d'encapsulation
- CD004 : Écrire des algos avec le pseudo code utilisé à l'INSA
- CD009 : Écrire un algorithme qui résout le problème
- CD402 : Concevoir et utiliser des listes doublement chaînées

procédure ajouter **(E/S** d : DO,**E** c : Cle, v : Valeur**)**
Déclaration nouveauNoeud : ListeDoublementChainee
debut
si non Dictionnaire.estPresent(d.valeurs,c) **alors**
nouveauNoeud ← listeDoublementChainee()
inserer(nouveauNoeud,c)
si estVide(d.clesDebut) **alors**
d.clesDebut ← nouveauNoeud
d.clesFin ← d.clesDebut

```
        sinon
            fixerListeSuivante(d.clesFin,nouveauNoeud)
            fixerListePrecedente(nouveauNoeud,d.clesFin)
            d.clesFin ← nouveauNoeud
        finsi
        Dictionnaire.ajouter(d.positions,c,nouveauNoeud)
    finsi
    Dictionnaire.ajouter(d.valeurs,c,v)
fin
```

Savoir et Savoir-faire évaluées

- CD002 : En tant qu'utilisateur, respecter une signature
- CD003 : Utiliser le principe d'encapsulation
- CD004 : Écrire des algos avec le pseudo code utilisé à l'INSA
- CD009 : Écrire un algorithme qui résout le problème
- CD402 : Concevoir et utiliser des listes doublement chaînées

```
procédure retirer (E/S d : DO,E c : Cle)
    Déclaration  noeudASupprimer, avant, apres : ListeDoublementChainee
debut
    si Dictionnaire.estPresent(d.valeurs,c) alors
        noeudASupprimer ← Dictionnaire.obtenirValeur(d.positions,c)
        supprimerNoeud(noeudASupprimer,avant,apres)
        si estVide(apres) alors
            d.clesFin ← avant
            si estVide(avant) alors
                d.clesDebut ← avant
            sinon
                fixerListeSuivante(avant,apres)
            finsi
        sinon
            fixerListePrecedente(apres,avant)
        finsi
        Dictionnaire.retirer(d.positions,c)
        Dictionnaire.retirer(d.valeurs,c)
    finsi
fin
```

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**)**