Algorithmique avancée et Programmation C

Durée : *3h00*
Documents autorisés : **AUCUN**

Remarques :
- Veuillez lire attentivement les questions avant de répondre.
- Le barème donné est un barème indicatif qui pourra évoluer lors de la correction.
- Rendez une copie propre.
- À chaque question est associée, à titre informatif, la difficulté estimée : ☺ facile, 😐 moyenne, ☹ difficile

1 Question de cours (4 points)

Soit l'arbre-b d'ordre 2 donné par la figure 1.

1. ☺ Donnez, en justifiant, les arbres-b issus des insertions successives des valeurs 68 puis 225.
2. ☺ Toujours à partir de l'arbre-b de la figure 1 ; donnez, en justifiant, les arbres-b issus des suppressions successives des valeurs 220 puis 125.

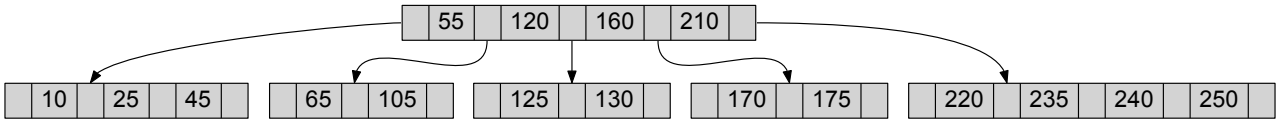


FIGURE 1 – Un arbre-b

2 Exercice de TD (6 points)

Pour rappel le SDD ListeChainee est défini de la façon suivante :

Type ListeChainee = ^Noeud
Type Noeud = **Structure**
  element : Element
  listeSuivante : ListeChainee
finstructure

On peut l'utiliser à l'aide des fonctions et procédures suivantes (principe d'encapsulation) :

- **fonction** listeVide **() :** ListeChainee
- **fonction** estVide (uneListe : ListeChainee) **: Booleen**
- **procédure** ajouter (**E/S** uneListe : ListeChainee,**E** element : Element)
- **fonction** obtenirElement (uneListe : ListeChainee) **:** Element

⌊**précondition(s)** $non(estVide(uneListe))$

— **fonction** obtenirListeSuivante (uneListe : ListeChainee) **:** ListeChainee

⌊**précondition(s)** $non(estVide(uneListe))$

— **procédure** fixerListeSuivante (**E** uneListe : ListeChainee, nelleSuite : ListeChainee)

⌊**précondition(s)** $non(estVide(uneListe))$

— **procédure** supprimerTete (**E/S** uneListe : ListeChainee)

⌊**précondition(s)** $non(estVide(uneListe))$

— **procédure** supprimer (**E/S** uneListe : ListeChainee)

1. ☺ Donnez l'algorithme de la procédure ajouter.
2. ☺ Donnez l'algorithme d'une fonction récursive qui permet de compter le nombre d'occurrences d'un élément.
3. ☺ Donnez l'algorithme d'une procédure permettant d'ajouter un élément en queue de liste.
4. ☺ Donnez l'algorithme d'une procédure permettant d'ajouter un élément après un élément donné. Si ce dernier n'est pas présent, il n'y a pas d'insertion.

3 Problème (10 points)

Aujourd'hui la plupart des traitements de texte possèdent un correcteur orthographique. La composante essentielle d'un correcteur orthographique est son dictionnaire, c'est-à-dire une entité qui contient un ensemble de mots et un ensemble de fonctionnalités permettant de « gérer » ces mots.

On peut donc considérer qu'un dictionnaire est un type abstrait, qui :
— permet de stocker des mots, c'est-à-dire qui permet d'ajouter ou de supprimer des mots,
— permet de savoir si un mot est correctement orthographié, c'est-à-dire s'il appartient au dictionnaire.

Un mot quant à lui est un type abstrait qui ressemble à une chaîne de caractères mais il ne peut pas être vide et il contient uniquement des lettres sans tenir compte de leur casse.

3.1 Spécification (2 points)

☺ Proposez les TAD Mot et Dictionnaire.

3.2 Conception préliminaire (1 point)

☺ Donnez les signatures des opérations des deux types abstraits précédents.

3.3 Conception détaillée (7 points)

On se propose de stocker les différents mots du dictionnaire en :
— optimisant l'opération qui permet de savoir si un mot est bien ou mal orthographié,
— optimisant la place mémoire qui sera utilisée pour stocker tous ces mots.

Pour ce faire, nous allons utiliser un arbre binaire tel que :
— chaque nœud de cet arbre contiendra deux informations :
  — un caractère,
  — un indicateur booléen permettant de savoir si le nœud courant marque la fin d'un mot,

— le fils gauche d'un nœud n permettra d'accéder aux suffixes des mots du dictionnaire qui ont comme préfixe les lettres contenu dans les nœuds du chemin de la racine au nœud n,
— le fils droit d'un nœud n permettra d'accéder aux différents suffixes d'un même préfixe. Les nœuds accessibles par les fils droits depuis la racine représentent le premier caractère des mots du dictionnaire (le préfixe est vide).

La figure 2 représente l'arbre binaire issu des insertions des mots *aussi*, *barda*, *auto*, *autos*, *barde*, *bardeau* et *aube* à un arbre initialement vide (la présence d'un rond à coté d'une lettre signifie que l'indicateur booléen de fin de mot est à vrai).

Pour mieux comprendre, nous allons étudier certains nœuds, que nous avons grisé. Celui contenant un 't' et celui contenant un 'b' sont à la droite de celui contenant un 's' car les quatre mots issus de ces trois nœuds, donc dans leurs fils gauches (*aussi*, *auto*, *autos* et *aube*) ont le même préfixe, ici « au ».

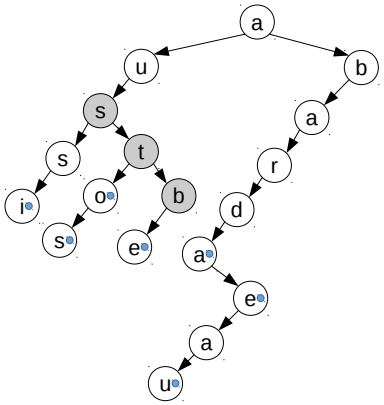
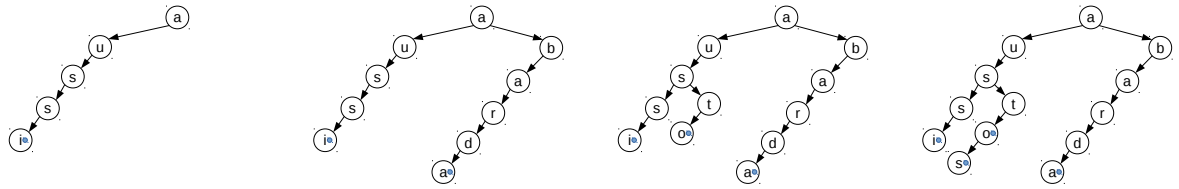


FIGURE 2 – Un arbre binaire pour stocker des mots

La figure 3 représente les arbres après les insertions successives des mots *aussi*, *barda*, *auto* et *autos* depuis un arbre initialement vide.



(a) aussi (b) barda (c) auto (d) autos

FIGURE 3 – Arbres après les insertions successives de mots depuis un arbre vide

Pour simplifier les algorithmes suivants, nous ne prenons pas en compte l'ordre lexicographique entre les caractères contenus dans les fils droits (comme c'est le cas pour les trois nœuds grisés).

1. ☺ Proposez une conception pour le type Dictionnaire.
2. 😐 Donnez l'algorithme de la fonction suivante qui permet de savoir si un mot est présent dans le dictionnaire :
   — **fonction** estPresent **(**d : Dictionnaire, m : Mot**) : Booleen**
3. ☹ Donnez l'algorithme de la procédure suivante qui permet d'ajouter un mot dans un dictionnaire :
   — **procédure** ajouter **(E/S** d : Dictionnaire, **E** m : Mot**)**