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.

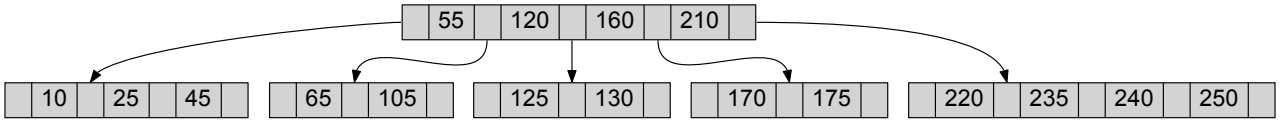
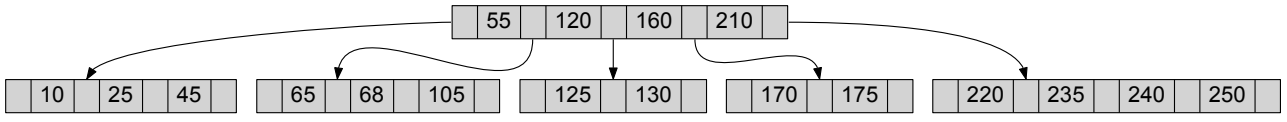


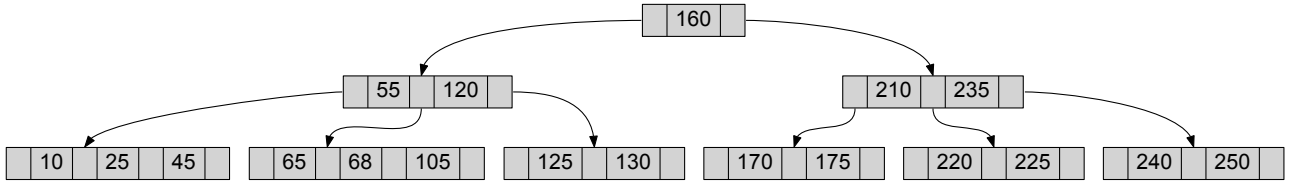
FIGURE 1 – Un arbre-b

Solution proposée :

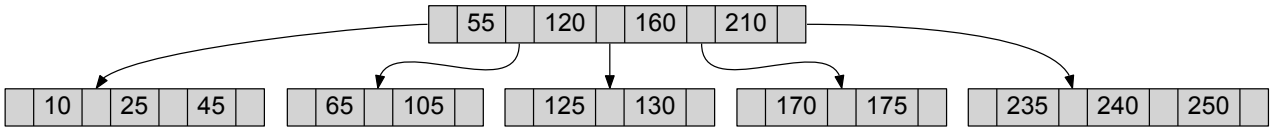
— Après insertion de 68 (0.5 point)



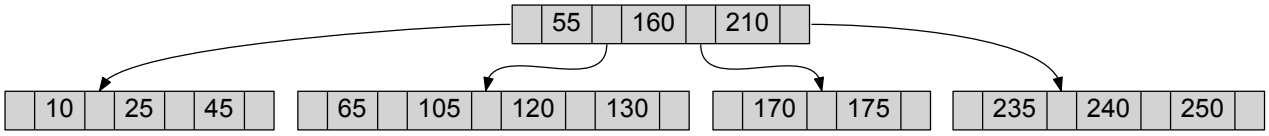
— Après insertion de 225 (1.5 points)



— Après suppression de 220 (0.5 point)



— Après suppression de 125 (1.5 points)



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.

Solution proposée :

1. 1 point

```
procédure ajouter (E/S uneListe : ListeChainee,E element : Element)
   Déclaration  temp : ListeChainee
debut
   allouer(temp)
   temp→element ← element
   temp→listeSuivante ← uneListe
   uneListe ← temp
fin
```

2. 1 point

```
fonction nbOccurences (uneListe : ListeChainee, element : Element) : Naturel
debut
   si estVide(uneListe) alors
      retourner 0
   sinon
      si obtenirElement(uneListe) = element alors
         retourner 1 + nbOccurences(obtenirListeSuivante(uneListe),element)
      sinon
         retourner nbOccurences(obtenirListeSuivante(uneListe),element)
      finsi
   finsi
fin
```

3. 2 points

```
procédure ajouterEnQueue (E/S uneListe : ListeChainee,E element : Element)
   Déclaration  temp : ListeChainee
debut
   si estVide(uneListe) alors
      ajouter(uneListe,element)
   sinon
      temp ← obtenirListeSuivante(uneListe)
      ajouterEnQueue(temp,element)
      si estVide(obtenirListeSuivante(uneListe)) alors
         fixerListeSuivante(uneListe, temp)
      finsi
   finsi
fin
```

4. 2 points

```
procédure insererApres (E/S uneListe : ListeChainee,E elementAInserer, element : Element)

   Déclaration  temp : ListeChainee
debut
   si non estVide(uneListe) alors
      temp ← obtenirListeSuivante(uneListe)
      si obtenirElement(uneListe) = element alors
         ajouter(temp, elementAInserer)
      sinon
```

```
            insererApres(temp, elementAInserer, element)
        finsi
        fixerListeSuivante(uneListe, temp)
    finsi
fin
```

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.

Solution proposée :

Nom :	Mot	
Utilise :	**NaturelNonNul**, **Caractere**	
Opérations :	mot :	Chaine $\nrightarrow$ Mot
	longueur :	Mot $\rightarrow$ **NaturelNonNul**
	iemeCaractere :	Mot $\times$ **NaturelNonNul** $\nrightarrow$ **Caractere**
	sousMot :	Mot $\times$ **NaturelNonNul** $\times$ **NaturelNonNul** $\nrightarrow$ Mot
Préconditions :	mot(ch) :	ch$\neq$"" et $\forall$ i $\in$ 1..longueur(ch), iemeCaractere(ch,i) $\in \{'a', .., 'z', 'A', .. 'Z$
	iemeCaractere(m,i) :	i$\leq$longueur(m)
	sousMot(m,d,f) :	d$\leq$longueur(m) et f$\leq$longueur(m) et d<f

Nom :	Dico	
Utilise :	Mot, **Booleen**	
Opérations :	dictionnaire :	$\nrightarrow$ Dico
	estPresent :	Dico $\times$ Mot $\rightarrow$ **Booleen**
	ajouter :	Dico $\times$ Mot $\nrightarrow$ Dico
	supprimer :	Dico $\times$ Mot $\nrightarrow$ Dico
Préconditions :	ajouter(d,m) :	non estPresent(d,m)
	supprimer(d,m) :	estPresent(d,m)

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 permettra d'accéder aux lettres composant les suffixes possibles,
— le fils droit d'un nœud permettra d'accéder aux différents mots qui ont le même préfixe.

L'arbre binaire représentant un dictionnaire au départ vide auquel on ajouterait successivement les mots *aussi*, *barda*, *auto*, *autos*, *barde*, *bardeau* et *aube* est présenté par la figure 2 (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 avons grisé certains nœuds. 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 nœuds (donc dans leurs fils gauches) ont le même préfixe, ici « au ».

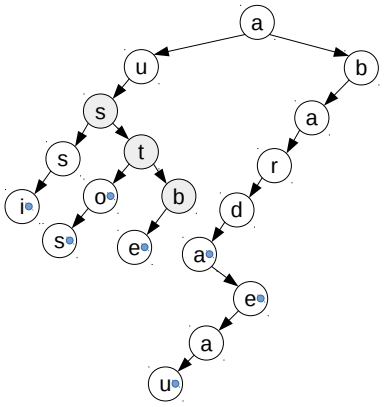


FIGURE 2 – Un arbre binaire pour stocker des mots

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**)**

Solution proposée :

— 1 point

```
Type NoeudLettre = Structure
    lettre : Caractere
    finDeMot : Booleen
finstructure
Type Dico = Structure
    arbre : ArbreBinaire<NoeudLettre>
```

```
    finstructure
```

— 2,5 points

```
fonction estPresent (d : Dico, m : Mot) : Booleen
debut
    retourner estPresentR(d.arbre,m)
fin
fonction estPresentR (a : ArbreBinaire<NoeudLettre>, m : Mot) : Booleen
debut
    si estVide(a) alors
        retourner FAUX
    sinon
        si longueur(m)=1 alors
            retourner obtenirElement(a).lettre = iemeCaractere(m,1) et obtenirElement(a).finDeMot
        sinon
            si obtenirElement(a).lettre = iemeCaractere(m,1) alors
                retourner estPresentR(obtenirFilsGauche(a),sousMot(m,2,longueur(m)))
            sinon
                retourner estPresentR(obtenirFilsDroit(a),m)
            finsi
        finsi
    finsi
fin
```

— 3,5 points

```
procédure ajouter (E/S d : Dico, E m : Mot)
debut
    ajouterR(d.arbre, m)
fin
procédure ajouterR (E/S a : ArbreBinaire<NoeudLettre>, E m : Mot)
    Déclaration  n : Noeud
                 temp : ArbreBinaire<NoeudLettre>
debut
    si estVide(a) alors
        n.lettre ← iemeCaractere(m,1)
        n.finDeMot ← longueur(m)=1
        a ← creerRacine(n, arbreBinaire(), arbrebinaire())
        si longueur(m)>1 alors
            temp ← obtenirFilsGauche(a)
            ajouterR(temp,sousMot(2,longueur(m)))
            fixerFilsGauche(a,temp)
        finsi
    sinon
        si obtenirElement(a).lettre = iemeCaractere(m,1) alors
            si longueur(m) = 1 alors
                n ← obtenirElement(a)
                n.finDeMot ← VRAI
                fixerElement(a,n)
            sinon
                temp ← obtenirFilsGauche(a)
                ajouterR(temp,sousMot(2,longueur(m)))
                fixerFilsGauche(a,temp)
```

```
        finsi
      sinon
        temp ← obtenirFilsDroit(a)
        ajouterR(temp,m)
        fixerFilsGauche(a,temp)
      finsi
    finsi
fin
```