Algorithmique et Base de la programmation

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.

1 Un autre algorithme pour le tri par insertion (4 points)

Pour rappel l'algorithme de tri par insertion (en ordre croissant d'un tableau d'entiers) que nous avons vu en cours est le suivant :

```
procédure triParInsertion (E/S t :Tableau[1..MAX] d'Entier,E nb :NaturelNonNul)
   Déclaration  i,j : NaturelNonNul
                temp : Entier
debut
   pour i ←2 à nb faire
      j ← obtenirIndiceDInsertion(t,i,t[i])
      temp ← t[i]
      decaler(t,j,i)
      t[j] ← temp
   finpour
fin
```

Il existe une variante de cet algorithme qui effectue *en même temps* la recherche de l'indice d'insertion et le décalage (en effectuant cette recherche et ce décalage depuis l'indice i et non pas l'indice 1). L'algorithme devient donc :

```
procédure triParInsertion (E/S t :Tableau[1..MAX] d'Entier,E nb :NaturelNonNul)
   Déclaration  i : Naturel
debut
   pour i ←2 à nb faire
      inserer(t,i)
   finpour
fin
```

Proposez l'algorithme de la procédure `inserer`. Montrez que la complexité de cet algorithme du tri par insertion est alors en $O(n^2)$ et en $\Omega(n)$.

Solution proposée :

```
procédure inserer (E/S t :Tableau[1..MAX] d'Entier,E i :NaturelNonNul)
```

Déclaration temp : **Entier**
debut
temp ← t[i]
tant que i>1 et t[i-1]>temp **faire**
t[i] ← t[i-1]
i ← i-1
fintantque
t[i] ← temp
fin

Dans le pire des cas, la procédure `inserer` fera i itérations. Dans ce cas la complexité du tri sera égal à $1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2}$ soit $O(n^2)$.

Dans le meilleur des cas, la procédure `inserer` ne fera aucune itération, et donc une complexité en $\Omega(1)$. Dès lors la complexité du tri sera en $\Omega(n)$.

2 Calculatrice (9 points)

L'objectif ici est de concevoir une calculatrice avec des possibilités étendues.

2.1 Première version

Les contraintes d'utilisation de la première version de cette calculatrice sont :
- la chaîne doit correspondre à une opération arithmétique simple : opérande opérateur opérande ;
- une opérande est un nombre (entier ou réel) positif ou une opération arithmétique entre parenthèses ;
- le caractère représentant la virgule des nombres réels peut être '.' ou ',' ;
- un nombre réel est composé d'une partie entière (avant la virgule) et d'une partie réelle (après la virgule). La partie entière ou (exclusif) la partie réelle est optionnelle ;
- les caractères représentant les parenthèses sont uniquement '(' pour la parenthèse ouvrante et ')' pour la parenthèse fermante ;
- les opérations sont uniquement l'addition (caractère '+'), la soustraction (caractère '-'), la multiplication (caractère '*') et la division (caractère '/') ;

Nous avons précédemment étudié une analyse descendante permettant de calculer une expression arithmétique (contenant uniquement des nombres positifs) représentée par une chaîne de caractères. La figure 1 présente cette analyse.

Pour rappel :
- la chaîne de caractères en entrée de chaque opération représente la chaîne dans sa globalité ;
- le naturel non nul en entrée de chaque opération représente l'indice de la chaîne où débute l'analyse effectuée par l'opération ;
- le booléen en sortie de chaque opération permet de savoir si l'analyse effectuée est un succès (VRAI si tout s'est bien passé, FAUX lorsqu'il y a eu un problème) ;
- le naturel non nul en sortie de chaque opération représente l'indice de la chaîne ou se termine l'analyse effectuée par l'opération dans le cas ou le booléen précédent vaut VRAI. Si ce dernier vaut FAUX, ce naturel non nul vaut la même valeur que celle fournie en entrée.

Questions (3 points)

Proposez l'algorithme des fonctions ou procédures (rappelez si besoin les signatures de fonctions usuelles sur les chaînes de caractères que vous utiliserez) :

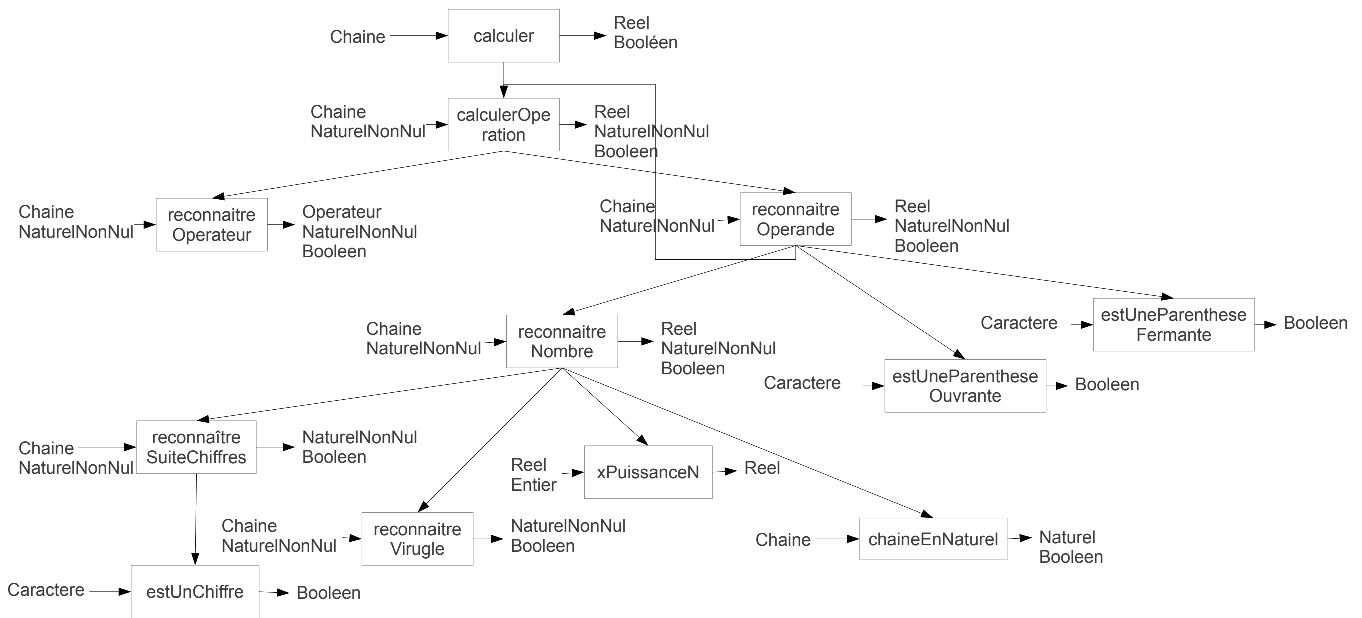


FIGURE 1 – Analyse descendante de l'opération `calculer`

1. de l'opération `reconnaitreNombre` ;

   Solution proposée :

   procédure reconnaitreNombre **(E** leTexte : **Chaine de caracteres** ; debut : **NaturelNonNul** ; **S** leReel : **Reel** ; fin : **NaturelNonNul** ; ok : **Booleen)**

   ⌊**précondition(s)** debut $\leq$ longueur(leTexte)

   Déclaration partieEntiere, partieReelle : **Naturel**

```
debut
   reconnaitreSuiteChiffres(leTexte,debut,fin,ok)
   si ok alors
      partieEntiere chaineEnNaturel(sousChaine(leTexte,debut,fin-1))
      reconnaitreVirgule(leTexte,debut,fin,ok)
      si ok alors
         reconnaitreSuiteChiffres(leTexte,debut,fin,ok)
         si ok alors
            partieReelle chaineEnNaturel(sousChaine(leTexte,debut,fin-1))
            leReel ← partieEntiere + partieReelle / xPuissanceN(10,fin-debut)
         sinon
            leReel ← partieEntiere
            ok ← VRAI
         finsi
      sinon
         leReel ← partieEntiere
         ok ← VRAI
      finsi
   sinon
      reconnaitreVirgule(leTexte,debut,fin,ok)
      si ok alors
         reconnaitreSuiteChiffres(leTexte,debut,fin,ok)
         si ok alors
```

```
        partieReelle chaineEnNaturel(sousChaine(leTexte,debut,fin-1))
        leReel ← partieReelle / xPuissanceN(10,fin-debut)
      finsi
    finsi
  finsi
fin
```

2. de l'opération `reconnaitreOperande`.
   Solution proposée :

   procédure reconnaitreOperande **(E** leTexte : **Chaine de caracteres** ; debut : **NaturelNonNul** ; **S** leReel : **Reel** ; fin : **NaturelNonNul** ; ok : **Booleen)**

   ⌊**précondition(s)** debut $\leq$ longueur(leTexte)

```
debut
  si estUneParentheseOuvrante(iemeCaractere(leTexte,debut)) alors
    reconnaitreOperation(leTexte, debut+1, leReel, fin, ok)
    si ok alors
      si estUneParentheseFermante(iemeCaractere(leTexte, fin)) alors
        fin ← fin+1
      sinon
        ok ← FAUX
      finsi
    finsi
  sinon
    reconnaitreNombre(leTexte,debut,leReel,fin,ok)
  finsi
fin
```

2.2 Deuxième version

On veut maintenant que notre calculatrice soit capable de calculer la valeur d'une expression qui pourrait contenir des variables (leurs identifiants seraient une chaîne de caractères contenant uniquement des lettres majuscules ou minuscules). Les valeurs de ces variables seraient définies dans une mémoire, qui serait une nouvelle entrée de l'opération `calculer`.

2.2.1 TAD Mémoire (4 points)

Comme nous venons de le voir, une mémoire associe des valeurs (de type réel) à des identifiants de variables. Les opérations disponibles pour une mémoire sont :
- création d'une mémoire vide ;
- savoir si une variable est présente dans une mémoire ;
- ajouter une variable (identifiant et valeur) dans une mémoire (il faut dans ce cas que la variable ne soit pas présente) ;
- modifier la valeur d'une variable dans une mémoire (il faut dans ce cas que la variable soit présente) ;
- obtenir la valeur d'une variable dans une mémoire(il faut dans ce cas que la variable soit présente) ;
- retirer une variable d'une mémoire (il faut dans ce cas que la variable soit présente).

1. Analyse : proposez le TAD `Memoire` ;
   Solution proposée :

Nom: Memoire

Utilise: **Reel**,**Chaine de caracteres**,**Booleen**

Opérations:

memoire:	$\rightarrow$ Memoire
estPresente:	Memoire $\times$ **Chaine de caracteres** $\rightarrow$ **Booleen**
ajouter:	Memoire $\times$ **Chaine de caracteres** $\times$ **Reel** $\nrightarrow$ Memoire
modifier:	Memoire $\times$ **Chaine de caracteres** $\times$ **Reel** $\nrightarrow$ Memoire
obtenir:	Memoire $\times$ **Chaine de caracteres** $\nrightarrow$ **Reel**
retirer:	Memoire $\times$ **Chaine de caracteres** $\nrightarrow$ Memoire

Préconditions:

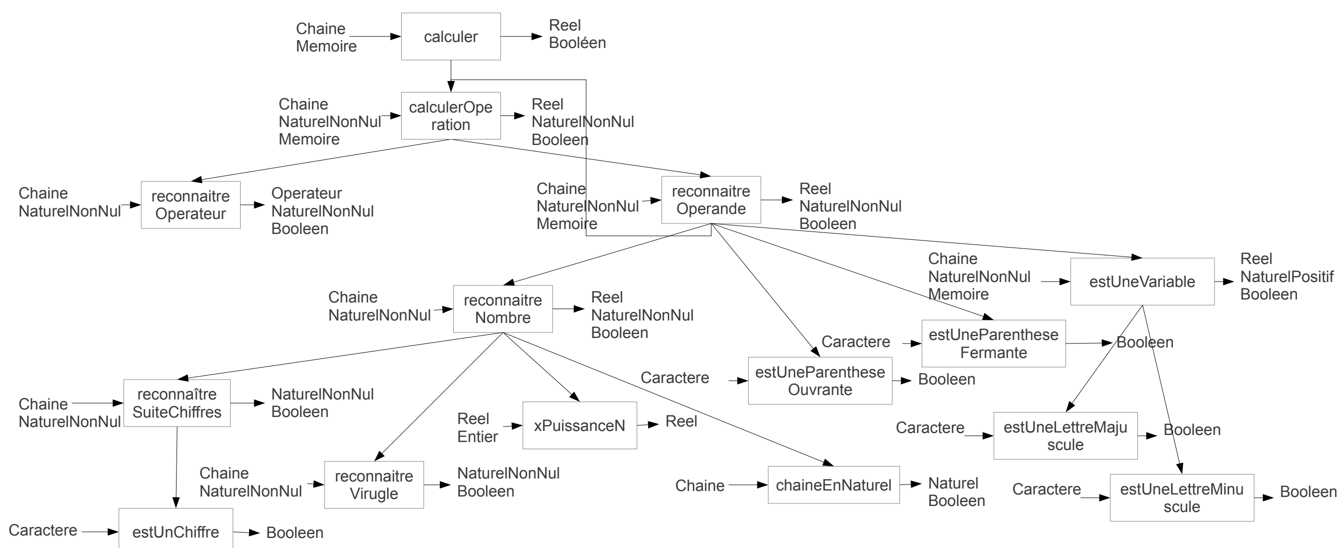
ajouter(m,i,v):	non estPresente(m,i)
modifier(m,i,v):	estPresente(m,i)
obtenir(m,i):	estPresente(m,i)
retirer(m,i):	estPresente(m,i)

2. Conception préliminaire : donnez les signatures des fonctions et procédures de ce TAD ;
   Solution proposée :
   - **fonction** memoire **() :** Memoire
   - **fonction** estPresente **(**m : Memoire, id : **Chaine de caracteres) : Booleen**
   - **procédure** ajouter **(E/S** m : Memoire, **E** id : **Chaine de caracteres**, v : **Reel)**
     ⌊**précondition(s)** non estPresente(m,id)
   - **procédure** modifier **(E/S** m : Memoire, **E** id : **Chaine de caracteres**, v : **Reel)**
     ⌊**précondition(s)** estPresente(m,id)
   - **fonction** obtenir **(**m : Memoire, id : **Chaine de caracteres) : Reel**
     ⌊**précondition(s)** estPresente(m,id)
3. **procédure** retirer **(E/S** m : Memoire, **E** id : **Chaine de caracteres)**
   ⌊**précondition(s)** estPresente(m,id)
4. Conception détaillée : proposez une implantation de ce TAD (uniquement le type, vous ne donnerez pas les algorithmes des fonctions et procédures). Justifiez votre choix.
   Solution proposée :
   Puisque le but de mémoire est d'associer une valeur à un identifiant de variable, le mieux est de représenter la type `Memoire` à l'aide d'un dictionnaire (c'est le concepteur du TAD dictionnaire qui sera en charge de trouver la conception la plus performante) :
   - **Type** Memoire = Dictionnaire<**Chaine de caracteres**,**Reel**>

2.2.2 Modification de l'analyse descendante (2 points)

Proposez une nouvelle analyse descendante pour l'opération `calculer` permettant aux opérandes de pouvoir être des nombres, des opérations ou des variables (vous ne redessinerez sur votre copie que les parties de l'analyse que vous modifierez).

Solution proposée :



3 Arbre de codage (7 points)

Nous avons vu en projet que l'on peut obtenir un code à longueur variable grâce à un arbre binaire particulier (que l'on a appelé `ArbreDHuffman` dans le projet et que l'on va appeler `ArbreCodage` ici).

Soit le TAD `ArbreCodage` suivant :

Nom:	ArbreCodage	
Paramètre:	Element	
Utilise:	**Booleen**	
Opérations:	creerFeuille:	Element $\times$ **Naturel** $\to$ ArbreCodage
	creerArbre:	ArbreCodage $\times$ ArbreCodage $\to$ ArbreCodage
	estUneFeuille:	ArbreCodage $\to$ **Booleen**
	element:	ArbreCodage $\nrightarrow$ Element
	cout:	ArbreCodage $\to$ **Naturel**
	filsGauche:	ArbreCodage $\nrightarrow$ ArbreCodage
	filsDroit:	ArbreCodage $\nrightarrow$ ArbreCodage
Préconditions:	element(a):	estUneFeuille(a)
	filsGauche(a):	non estUneFeuille(a)
	filsDroit(a):	non estUneFeuille(a)

Nous avons vu en projet une méthode (utilisant une file de priorité) permettant de créer un arbre d'Huffman à partir de statistiques. Il existe un autre algorithme (Shannon-Fano) permettant d'obtenir un résultat pratiquement équivalent. Cet algorithme prend en entrée un tableau d'éléments (triés en ordre croissant de leurs occurrences) et aura en sortie l'arbre de codage correspondant.

Le principe est le suivant : à partir d'une partie du tableau (identifiée par deux indices d et f), un algorithme va diviser cette partie en deux parties (de d à m et $m+1$ à f) telles que la somme des occurrences de la partie gauche (de d à m) soit égale à la somme des occurrences de la partie de droite (de $m+1$ à f) à une valeur près **minimale**. Dans ce cas, la partie du tableau (de d à f) correspond à un arbre de codage dont :

- le fils gauche est un arbre de codage construit à partir des éléments du tableau de d à m ;
- le fils droit est un arbre de codage construit à partir des éléments du tableau de $m+1$ à f ;

Bien entendu, si $d = f$, alors l'arbre de codage créé sera une feuille.

On possède les types `Occurrence` et `Statistiques` :

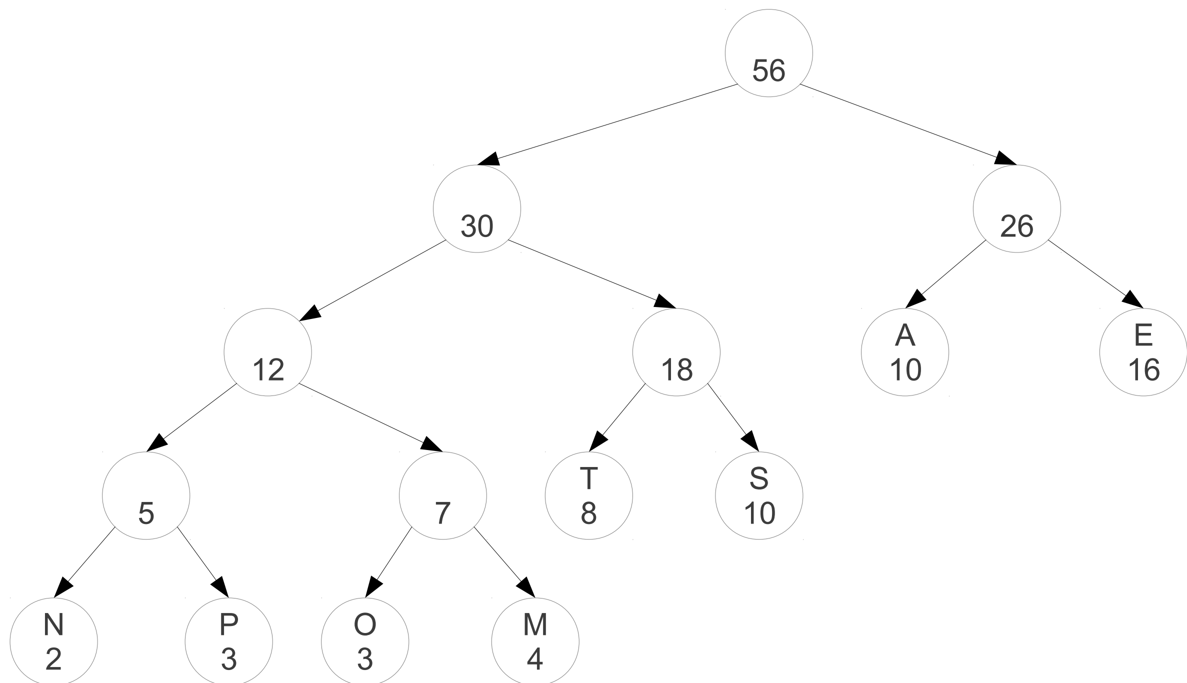
Type Occurence = **Structure**
e : Element
nb : **Naturel**
finstructure

Type Statistiques = **Structure**
s : **Tableau**[1..MAX] **d'**Occurence
nb : **Naturel**
finstructure

1. Donnez l'arbre de codage créé par cet algorithme pour les occurrences suivantes :

N	P	O	M	T	S	A	E
2	3	3	4	8	10	10	16

Solution proposée :



2. Pourquoi dit-on que l'algorithme d'Huffman vu en projet est *bottom-up* alors que celui de cet examen est *top-down* ?
Solution proposée :
Par ce que d'un l'algorithme vu en projet l'arbre est construit en partant des feuilles, alors qu'ici il est construit en partant de la racine de l'arbre.
3. Proposez l'algorithme de la fonction suivante qui est capable de calculer m (on sait $t.s$ trié en ordre croissant au regard des occurences) :
   - **fonction** diviser **(**t : Statistiques; d,f : **NaturelNonNul) : NaturelNonNul**
     ⌊**précondition(s)** d<f

   Solution proposée :
   Le principe est d'incrémenter d et décrementer f jusqu'à ce qu'ils aient le même valeur. La difficulté réside dans le fait qu'il faut savoir s'il faut retourner d ou $d-1$.
   Soit sd la somme de la partie gauche et sf la somme de la partie droite à la fin de l'itération. sd et sf ont en commun $t.s[d].nb$ (noté x). Il faut donc savoir si cette valeur doit appartenir à

la partie gauche ou la partie droite. Il faut donc comparer $sd - x$ et $sf - x$. Si $sd - x < sf - x$ c'est que x doit appartenir à la partie gauche, sinon à la partie droite.

```
fonction diviser (t : Statistiques ; d,f : NaturelNonNul) : NaturelNonNul
   ⌊précondition(s)  d<f
   Déclaration  sd, sf : Naturel
debut
   sd ← t.s[d].nb
   sf ← t.s[f].nb
   tant que d<f faire
      si sd<sf alors
         d ← d+1
         sd ← sd+t.s[d].nb
      sinon
         f ← f-1
         sf ← sf+t.s[f].nb
      finsi
   fintantque
   si sd-t.s[d].nb<sf-t.s[d] alors
      retourner d
   sinon
      retourner d-1
   finsi
fin
```

4. Proposez l'algorithme de la fonction suivante qui créé un arbre de codage à partir de statistiques (on sait $t.s$ trié en ordre croissant au regard des occurences) :
   – **fonction** creerArbreCodage (t : Statistiques) **:** ArbreCodage

   Solution proposée :

```
fonction creerArbreCodage (t : Statistiques) : ArbreCodage
debut
   retourner creerArbreCodageR(t,1,t.nb
fin
fonction creerArbreCodageR (t : Statistiques, d,f : Naturel) : ArbreCodage
   Déclaration  m : Naturel
debut
   si d=f alors
      retourner creerFeuille(t.s[d].e,t.s[d].nb)
   sinon
      m ← diviser(t,d,f)
      retourner creerArbre(creerArbreCodageR(t,d,m-1),creerArbreCodageR(t,m,f))
   finsi
fin
```