Lauriol François
Gras Yael

Groupe MN - 1

Raffinages

PROGRAMME COMPRESSION

R0 : Compresser un fichier texte avec ou sans l'option -b.

Nom_Fichier : Unbounded_String

Option_b : Boolean

R1 : Comment "compresser un fichier texte avec ou sans l'option -b" ?

- Créer l'arbre de Huffman et le dictionnaire d'un fichier texte avec ou sans l'option -b Fichier_txt : in File_Type; Texte : in out UNBOUNDED_STRING ; Table : in out T_FREQUENCES ; Dictionnaire : in out T_DICTIONNAIRE ; Arbre_Huff : in out ARBRE
- Créer le fichier .hff Fichier_hff Fichier_hff : out File_Type
- Ouvrir le fichier Fichier_hff Fichier_hff : in out File_Type
- Enregistrer le tableau des caractères et l'arbre de Huffman dans le Texte_hff Texte_hff : in UNBOUNDED_STRING; Arbre_Huff : in out ARBRE ; Dictionnaire : in out T_Dictionnaire
- Encoder les octets du texte selon le dictionnaire créé dans le fichier.hff Fichier_hff : in out Fichier_Type ; Texte_hff : in UNBOUNDED_STRING ; Dictionnaire : in T_DICTONNAIRE

R2 : Comment "créer l'arbre de Huffman et le dictionnaire d'un fichier texte avec ou sans l'option -b" ?

- LIRE(Fichier_txt, Texte)
- CALCULER_FREQ (Table, Texte)
- CONSTRUIRE_ARBRE(Arbre_Huff, Table)
- FABRIQUER_DICT(Dictionnaire, Arbre_Huff)
- Si Option_b faire
    - Afficher l'arbre de Huffman et le Dictionnaire
- Fin Si

Lauriol François
Gras Yael

R2 : Comment “enregistrer le tableau des caractères et l’arbre de Huffman dans le Texte_hff” ?

- Texte_hff <- TABLE_EN_BINAIRE(Dictionnaire)
- Texte_hff <- Texte_hff & ARBRE_EN_BINAIRE(Arbre_Huff)
- VIDER_ARBRE(Arbre_Huff)

R2 : Comment “encoder les octets du texte selon le dictionnaire créé dans le fichier.hff” ?

- Pour chaque Caractere du Texte faire
    - Texte_hff <- Texte_hff & LA_DONNEE_DICT(Dictionnaire, Caractere, False)
- Fin Pour
- Texte_hff <- Texte_hff & LA_DONNEE_DICT(Dictionnaire, ‘a’, True)
- ECRIRE_COMPRESSION(Fichier_hff, Texte_hff)
- VIDER_DICT(Dictionnaire)
- Fermer le fichier .hff Fichier_hff

R3 : Comment “afficher l’arbre de Huffman et le Dictionnaire” ?

- AFFICHER_ARBRE(Arbre_Huff)
- AFFICHER_DICT(Dictionnaire)

Lauriol François
Gras Yael

Groupe MN - 1

PROGRAMME DECOMPRESSION

R0 : Décompresser le fichier texte préalablement compressé en .hff

R1 : Comment décompresser le fichier texte préalablement compressé en .hff ?

- Créer le dictionnaire d'un fichier texte préalablement compressé en .hff Arbre_Huff : in out ARBRE ; Dictionnaire : in out T_DICTIONNAIRE ; Texte : in out UNBOUNDED_STRING ; Fichier_hff : in File_Type
- Créer le fichier .txt Fichier_txt Fichier_txt : out File_type
- Ouvrir le fichier Fichier_txt Fichier_txt : in out File_Type
- Encoder les bits en caractère selon le dictionnaire Texte : in UNBOUNDED_STRING ; Dictionnaire : in out T_DICTONNAIRE ; Fichier_txt : out File_type ; Code : in out UNBOUNDED_STRING ; Fini_txt : in out Booléen
- Fermer le fichier .txt Fichier_txt Fichier_txt : in out File_type

R2 : Comment "créer le dictionnaire d'un fichier texte préalablement compressé en .hff" ?

- LIRE(Fichier_hff, Texte)
- CONVERTIR_BINAIRE(Texte)
- RECUPERER_ARBRE(Arbre_Huff, Texte)
- FABRIQUER_DICT(Dictionnaire, Arbre_Huff)
- VIDER_ARBRE(Arbre_Huff)

R2 : Comment "encoder les bits en caractère selon le dictionnaire"?

- Code <- To_Unbounded_String("")
- Pour chaque Caractère dans Texte faire
    - Code <- Code & To_Unbounded_String("" & Caractère)
    - Ajouter le caractère correspondant au Fichier_txt si Code est un code de Huffman
- Fin Pour
- VIDER_DICT(Dictionnaire)

Lauriol François
Gras Yael

R3 : Comment “ajouter le caractère correspondant au Fichier_txt si Code est un code de Huffman” ?

- Fini_txt <- Fini_txt ou sinon (LA_DONNEE_DICT(Dictionnaire, 0, True)=Code) ---- on vérifie que l’on n’a pas atteint la fin du texte de Huffman
- Si not(Fini_txt) et alors CODE_EST_PRESENT_DICT(Code, Dictionnaire) faire
    - ECRIRE_DECOMPRESSER(Fichier_txt; LE_CARACTERE_DICT(Dictionnaire, Code)) — on remplace le code de Huffman trouvé par son caractère respectif dans le .txt
    - Code = To_Unbounded_String(“”) ---- on réinitialise Code
- Fin si

Lauriol François
Gras Yael

MODULE ARBRE DE HUFFMAN

TYPES NON PRIVATE :

type T_Octet est mod 256

Exception_Donnee_Absente : EXCEPTION ---- cas où le dictionnaire n'a pas le caractère demandé

TYPES LIMITED PRIVATE :

type C_Frequences est ENREGISTREMENT
    Caractere : Caractère
    Frequence : entier {Frequence>=0}
FIN ENREGISTREMENT

type A_Frequences est TABLEAU de (1..256) de C_Frequences

type T_Frequences est ENREGISTREMENT
    Taille : entier {0 <= Taille et Taille <= 256}
    Tableau : A_Frequences
FIN ENREGISTREMENT

type ARBRE est POINTEUR sur T_FEUILLE

type T_FEUILLE est ENREGISTREMENT
    Frequence : entier {Frequence >= 0}
    Caractere : Caractère
    Fils_G : ARBRE
    Fils_D : ARBRE
        {(Fils_Gauche = null and Fils_Droit = null) or (Fils_Gauche /= null and Fils_Droit /= null)}
FIN ENREGISTREMENT

type LCA est POINTEUR SUR T_CELLULE

type T_CELLULE est ENREGISTREMENT
    Caractere : Caractère
    Code_Huff : UNBOUNDED_STRING
    Suivant : LCA
FIN ENREGISTREMENT

type T_DICTIONNAIRE est TABLEAU(1..27) de LCA – Choix arbitraire basé sur le principe d'équi-répartition de l'alphabet dans le dictionnaire entre 2 et 27 (sachant que le premier élément du tableau ne décrit qu'un seul caractère '\$')

Lauriol François
Gras Yael

FONCTIONS DE l'ADS

—-- nom : CALCULER_FREQ
---- sémantique : calculer la fréquence des caractères dans le Texte et le retranscrire dans la
---- Table.
---- paramètres : Table : out T_FREQUENCES ---- Table des caractères adjoint avec leur
---- fréquence correspondante
---- Texte : in UNBOUNDED_STRING ---- Texte à analyser
---- tests : Entrées Rien : "" => Table.Taille=0
---- Entrées 1 type d'élément : "aa" => Table.Taille=2 & Table.Tableau(1)={'a',2}
---- Entrées 2 types d'élément : "aab" => Table.Taille=2 & Table.Tableau(1)={'a',2} &
---- Table.Tableau(2)={'b',1}
procedure CALCULER_FREQ(Table : in out T_FREQUENCES; Texte : in UNBOUNDED_STRING)

—-- nom : CONSTRUIRE_ARBRE
---- sémantique : construire l'arbre de Huffman à partir du Tableau des fréquences des
---- caractères.
---- paramètres : Arbre_Huff : out ARBRE
---- Table : in T_FREQUENCES ---- Tableau des fréquences des caractères
---- tests : Entrées Rien : Table.Taille=0 => Arbre_Huff^.Frequence=0 &
---- Character'pos(Arbre_Huff^.Caractere)=0 &
Arbre_Huff^.Fils_G=null
---- & Arbre_Huff^.Fils_D=null
---- Entrées 1 type d'élément :Table.Taille=1 & Table.Tableau(1)={'a',2} =>
---- Arbre_Huff^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Frequence=0 &
---- Character'pos(Arbre_Huff^.Fils_G^.Caractere)=0 &
---- Arbre_Huff^.Fils_G^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D=null
---- Entrées 2 types d'élément : Table.Taille=2 & Table.Tableau(1)={'a',2} &
---- Table.Tableau(2)={'b',1} => Arbre_Huff^.Frequence=3
---- & Arbre_Huff^.Fils_D^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Caractere='b' &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &

```
----                                    Character'pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)
----                                     =0 & Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
----                                     Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null
procedure CONSTRUIRE_ARBRE(Arbre_Huff : out ARBRE; Table : in T_FREQUENCES)


–-- nom : VIDER_ARBRE
---- sémantique : Vider l'espace mémoire occupé par l'arbre de Huffman
---- paramètres : Arbre_Huff : in out ARBRE
---- post-condition : Arbre_Huff=null
---- tests : Entrées Quelconque : Arbre_Huff => Arbre_Huff=null
procedure VIDER_ARBRE(Arbre_Huff : in out ARBRE)


–-- nom : FABRIQUER_DICT
---- sémantique : Former la table de Huffman Dict grâce à son arbre correspondant
----                  Arbre_Huff.
---- paramètres : Arbre_Huff : in ARBRE
----                  Dict : in out T_DICTIONNAIRE
---- tests : Entrées 0 caractère : Dict & Arbre_Huff^.Frequence=0 &
----                                  Character'pos(Arbre_Huff^.Caractere)=0 &
----                                  Arbre_Huff^.Fils_G=null &
----                                  Arbre_Huff^.Fils_D=null => pour tout i dans 2..27 on a
----                                  Dict(i)=null & Character'pos(Dict(1)^.Caractere)=0 &
----                                  Dict(1)^.Code_Huff=To_Unbounded_String("0")
----          Entrées 2 caractères : Arbre_Huff^.Frequence=3 &
----                                   Arbre_Huff^.Fils_D^.Frequence=2 &
----                                   Arbre_Huff^.Fils_D^.Caractere='a' &
----                                   Arbre_Huff^.Fils_D^.Fils_G=null &
----                                   Arbre_Huff^.Fils_D^.Fils_D=null &
----                                   Arbre_Huff^.Fils_G^.Frequence=1 &
----                                   Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
----                                   Arbre_Huff^.Fils_G^.Fils_D^.Caractere='b' &
----                                   Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
----                                   Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
----                                   Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &
----                                   Character'pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)=0
----                                   & Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
----                                   Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null => pour tout i
----                                  dans 2..27 différent de 21, 22 on a Dict(i)=null &
----                                  Dict(21)^.Caractere='a' &
----                                  Dict(21)^.Code_Huff=To_Unbounded_String("1")&
----                                  Dict(22)^.Caractere='b' &
----                                  Dict(22)^.Code_Huff=To_Unbounded_String("01")&
----                                  Character'pos(Dict(1)^.Caractere)=0 &
----                                  Dict(1)^.Code_Huff=To_Unbounded_String("00")
procedure FABRIQUER_DICT(Dict : in out T_DICTIONNAIRE; Arbre_Huff : in ARBRE)


–-- nom : LA_DONNEE_DICT
---- sémantique : Retourner le codage de Huffman d'un caractère
```

---- paramètres : Dict : in T_DICTIONNAIRE
---- Caractere : in Caractère
---- Fin : in BOOLÉEN -- Si on demande pour le caractère '\$'
---- Type de retour : UNBOUNDED_STRING
---- pré-condition : Fin=True ou alors Caractere est dans le Dictionnaire entre 2 et 27
---- tests : Entrées 0 caractère : pour tout i dans 2..27 on a Dict(i)=null &
---- Character'pos(Dict(1)^.Caractere)=0 & Caractere
---- quelconque & Dict(1)^.Code_Huff=
---- To_Unbounded_String("0") & Fin=True
---- => Résultat=To_Unbounded_String("0")
---- Entrées 2 caractères : pour tout i dans 2..27 différent de 21 et 22 on
---- a Dict(i)=null & Dict(21)^.Caractere='a' &
---- Dict(21)^.Code_Huff=To_Unbounded_String("1")&
---- Dict(22)^.Caractere='b' &
---- Dict(22)^.Code_Huff=To_Unbounded_String("01")&
---- Character'pos(Dict(1)^.Caractere)=0 & Fin = False &
---- Dict(1)^.Code_Huff=To_Unbounded_String("00") &
---- Caractère='b' => Résultat=To_Unbounded_String("01")
fonction LA_DONNEE_DICT(Dict : in T_DICTIONNAIRE; Caractere : in Caractère; Fin : in BOOLÉEN) retourne UNBOUNDED_STRING

—-- nom : LE_CARACTERE_DICT
---- sémantique : Retourner le caractère d'un codage de Huffman
---- paramètres : Dict : in T_DICTIONNAIRE
---- Code : in UNBOUNDED_STRING
---- Type de retour : Caractère
---- pré-condition : CODE_EST_PRESENT_DICT(Code,Dict)
---- tests : Entrées 0 caractère : pour tout i dans 1 ou 3..26 on a Dict(i)=null &
---- Character'pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String("0") &
---- Code=To_Unbounded_String("0") =>
---- Résultat=Character'val(0)
---- Entrées 2 caractères : pour tout i dans 2..27 différent de 21 et 22 on
---- a Dict(i)=null & Dict(21)^.Caractere='a' &
---- Dict(21)^.Code_Huff=To_Unbounded_String("1")&
---- Dict(22)^.Caractere='b' &
---- Dict(22)^.Code_Huff=To_Unbounded_String("01")&
---- Character'pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String("00") &
---- Code=To_Unbounded_String("01") => Résultat='b'
fonction LE_CARACTERE_DICT(Dict : in T_DICTIONNAIRE ; Code : in UNBOUNDED_STRING) retourne Caractère

—-- nom : VIDER_DICT
---- sémantique : Vider l'espace mémoire occupé par le Dictionnaire Dict
---- paramètres : Dict : in out T_DICTIONNAIRE
---- post-condition : pour tout i dans 1..27 Dict(i)=null
---- tests : Entrées Quelconque : Dict => pour tout i dans 1..27 Dict(i)=null

Lauriol François
Gras Yael

procedure VIDER_DICT(Dict : in out T_DICTIONNAIRE)

---- nom : RECUPERER_ARBRE
---- sémantique : Extraire l'arbre de Huffman d'un texte préalablement compressé
---- paramètres : Arbre_Huff : out ARBRE
---- Texte : in out UNBOUNDED_STRING
---- tests : Entrées 0 caractère : Texte= To_Unbounded_String (
---- "00000000"&"1"&"0"&"000000") =>
---- Arbre_Huff^.Frequence=0 &
---- Character'pos(Arbre_Huff^.Caractere)=0 &
---- Arbre_Huff^.Fils_G=null &
---- Arbre_Huff^.Fils_D=null &
---- Texte=To_Unbounded_String("0"&"000000")
---- Entrées 1 caractère : Texte= To_Unbounded_String (
---- "00000000"&"01100001"&"01100001"&"011"&"110"&"00")
=>
---- Arbre_Huff^.Frequence=1 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Character'pos(Arbre_Huff^.Fils_G^.Caractere)=0 &
---- Arbre_Huff^.Fils_G^.Frequence=0 &
---- Arbre_Huff^.Fils_G^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D=null &
---- Texte=To_Unbounded_String("110"&"00")
---- Entrées 2 caractère : Texte= To_Unbounded_String("00000000"&"01100010"&
---- "01100001"&"01100001"&"00111"&"110100"&"00000")=>
---- Arbre_Huff^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Caractere=null &
---- Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Caractere='b' &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &
---- Character'pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)=0
---- & Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null &
---- Texte= To_Unbounded_String("110100"&"00000")
procedure RECUPERER_ARBRE(Arbre_Huff : out ARBRE ; Texte : in out
UNBOUNDED_STRING)

—-- nom : CODE_EST_PRESENT_DICT
---- sémantique : Déterminer si le Code est bien un code de Huffman du Dictionnaire

---- paramètres : Code : in UNBOUNDED_STRING
---- Dict : in T_DICTIONNAIRE
---- Type de retour : BOOLÉEN
---- tests : Entrées 0 caractère : pour tout i dans 2..27 on a Dict(i)=null &
---- Character'pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String("0") &
---- Code=To_Unbounded_String("10") => Résultat=False
---- Entrées 2 caractères : pour tout i dans 2..27 différent de 21 et 22 on
---- a Dict(i)=null & Dict(21)^.Caractere='a' &
---- Dict(21)^.Code_Huff=To_Unbounded_String("1")&
---- Dict(22)^.Caractere='b' &
---- Dict(22)^.Code_Huff=To_Unbounded_String("01")&
---- Character'pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String("00") &
---- Code=To_Unbounded_String("01") => Résultat=True
fonction CODE_EST_PRESENT_DICT(Code : in UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE) retourne BOOLÉEN

—-- nom : ARBRE_EN_BINAIRE
---- sémantique : Renvoyer le codage binaire de la forme de l'arbre de Huffman
---- paramètres : Arbre_Huff : in ARBRE
---- Type de retour : UNBOUNDED_STRING
---- pré-condition : Arbre_Huff /=null
---- tests : Entrées 0 caractère : Arbre_Huff^.Frequence=0 &
---- Character'pos(Arbre_Huff^.Caractere)=0 &
---- Arbre_Huff^.Fils_G=null &
---- Arbre_Huff^.Fils_D=null =>
---- Résultat=To_Unbounded_String("1")
---- Entrées 1 caractère : Arbre_Huff^.Frequence=1 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Character'pos(Arbre_Huff^.Fils_G^.Caractere)=0 &
---- Arbre_Huff^.Fils_G^.Frequence=0 &
---- Arbre_Huff^.Fils_G^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D=null =>
---- Résultat=To_Unbounded_String("011")
---- Entrées 2 caractères : Arbre_Huff^.Frequence=3 &
---- Arbre_Huff^.Fils_D^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Caractere='a' &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Caractere='b' &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &

```
----                                  Character‘pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)=0
----                                  & Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
----                                  Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null =>
----                                 Résultat=To_Unbounded_String(“00111”)
fonction ARBRE_EN_BINAIRE(Arbre_Huff : in ARBRE) retourne UNBOUNDED_STRING

–-- nom : TABLE_EN_BINAIRE
---- sémantique : Renvoyer le codage binaire de la forme de la table de Huffman
---- paramètres : Dict : in T_DICTIONNAIRE
---- Type de retour : UNBOUNDED_STRING
---- tests : Entrées 0 caractère :  pour tout i dans 2..27 on a Dict(i)=null &
----                                 Character‘pos(Dict(1)^.Caractere)=0 &
----                                 Dict(1)^.Code_Huff=To_Unbounded_String(“0”) =>
----                                 Résultat=To_Unbounded_String(“00000000”)
----                                 (on discernera ce cas facilement car le fichier compressé
----                                  sera plus petit que deux octet)
----          Entrées 2 caractères : pour tout i dans 2..27 différent de 21 et 22 on
----                                  a Dict(i)=null & Dict(21)^.Caractere=’a’ &
----                                  Dict(21)^.Code_Huff=To_Unbounded_String(“1”)&
----                                  Dict(22)^.Caractere=’b’ &
----                                  Dict(22)^.Code_Huff=To_Unbounded_String(“01”)&
----                                  Character‘pos(Dict(1)^.Caractere)=0 &
----                                  Dict(1)^.Code_Huff=To_Unbounded_String(“00”) =>
----                                 Résultat=To_Unbounded_String( “00000000”&”01100010”&
----                                  ”01100001”&”01100001”)
fonction TABLE_EN_BINAIRE(Dict : in T_DICTIONNAIRE) retourne UNBOUNDED_STRING


---- nom : AFFICHER_ARBRE
---- sémantique : Afficher l’arbre de Huffman comme la figure 20 du polycopié de projet
---- paramètres : Arbre_Huff : in ARBRE
---- tests : Entrées 0 caractère :  Arbre_Huff^.Frequence=0 &
----                                 Character‘pos(Arbre_Huff^.Caractere)=0 &
----                                 Arbre_Huff^.Fils_G=null &
----                                 Arbre_Huff^.Fils_D=null =>
----                                 - - 0 - - (0) ‘\$’
----          Entrées 1 caractère : Arbre_Huff^.Frequence=1 &
----                                 Arbre_Huff^.Fils_D^.Caractere=’a’ &
----                                 Arbre_Huff^.Fils_D^.Frequence=1 &
----                                 Arbre_Huff^.Fils_D^.Fils_G=null &
----                                 Arbre_Huff^.Fils_D^.Fils_D=null &
----                                 Character‘pos(Arbre_Huff^.Fils_G^.Caractere)=0 &
----                                 Arbre_Huff^.Fils_G^.Frequence=0 &
----                                 Arbre_Huff^.Fils_G^.Fils_G=null &
----                                 Arbre_Huff^.Fils_G^.Fils_D=null =>
----                                 (2)
----                                  \ - - 0 - - (0) ‘\$’
----                                  \ - - 1 - - (2) ‘a’
----          Entrées 2 caractères : Arbre_Huff^.Frequence=3 &
----                                  Arbre_Huff^.Fils_D^.Frequence=2 &
----                                  Arbre_Huff^.Fils_D^.Caractere=’a’ &
```

```
----                                Arbre_Huff^.Fils_D^.Fils_G=null &
----                                Arbre_Huff^.Fils_D^.Fils_D=null &
----                                Arbre_Huff^.Fils_G^.Frequence=1 &
----                                Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
----                                Arbre_Huff^.Fils_G^.Fils_D^.Caractere='b' &
----                                Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
----                                Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
----                                Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &
----                                Character'pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)=0
----                               & Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
----                                Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null =>
----                               (3)
----                                \ - - 0 - - (1)
----                                |          \ - - 0 - - (0) '\$'
----                                |          \ - - 1 - - (1) 'b'
----                                \ - - 1 - - (2) 'a'
procedure AFFICHER_ARBRE(Arbre_Huff : in ARBRE)


–--- nom : AFFICHER_DICT
---- sémantique : Afficher la table de Huffman comme la figure 21 du polycopié de projet
---- paramètres : Dictionnaire : in T_DICTIONNAIRE
---- tests : Entrées 0 caractère :  pour tout i dans 2..27 on a Dict(i)=null &
----                                Character'pos(Dict(1)^.Caractere)=0
----                                Dict(1)^.Code_Huff=To_Unbounded_String("0") =>
----                                '\$' --> 0
----          Entrées 2 caractères : pour tout i dans 2..27 différent de 21 et 22 on
----                                 a Dict(i)=null & Dict(21)^.Caractere='a' &
----                                 Dict(21)^.Code_Huff=To_Unbounded_String("1")&
----                                 Dict(22)^.Caractere='b' &
----                                 Dict(22)^.Code_Huff=To_Unbounded_String("01")&
----                                 Character'pos(Dict(1)^.Caractere)=0 &
----                                 Dict(1)^.Code_Huff=To_Unbounded_String("00") =>
----                                '\$' --> 00
----                                'a' --> 1
----                                'b' --> 01
procedure AFFICHER_DICT(Dictionnaire : in T_DICTIONNAIRE)

---- nom : CONVERTIR_BINAIRE
---- sémantique : Convertir un texte de caractères en binaire avec leur code ASCII
----                  binaire correspondant.
---- paramètres : Texte : in out UNBOUNDED_STRING
procedure CONVERTIR_BINAIRE(Texte : in out UNBOUNDED_STRING)

---- nom : CONVERTIR_CARACTERES
---- sémantique : Convertir un texte en binaire en caractères avec leur code ASCII
----                  binaire correspondant.
---- paramètres : Texte : in out UNBOUNDED_STRING
```

Lauriol François
Gras Yael

procedure CONVERTIR_CARACTERES(Texte : in out UNBOUNDED_STRING)

Lauriol François
Gras Yael

Groupe MN - 1

RAFFINAGES DES FONCTIONS DE l'ADS DU MODULE

R0 : Calculer la fréquence des caractères dans le Texte et le retranscrire dans la Table des fréquences. (CALCULER_FREQ)

Table : in out T_FREQUENCES;
Texte : in UNBOUNDED_STRING

R1 : Comment "calculer les fréquences dans le Texte et le retranscrire dans la Table des fréquences" ?

- Copie_Texte <- Texte — Texte : in UNBOUNDED_STRING ; Copie_Texte : out UNBOUNDED_STRING
- Table.Taille <- 0
- Tant que length(Copie_Texte) > 0 faire — Copie_Texte : in UNBOUNDED_STRING
    - Initialiser les compteurs — Longueur, nb_suppression : out INTEGER
    - Caractere <- Copie_Texte(0) — Caractere : out CHARACTER
    - Compter la fréquence du Caractere — CARACTERE : in CHARACTER; Copie_Texte : in out UNBOUNDED_STRING; compteur, nb_suppression : in out INTEGER
    - Incrémenter la table des fréquences — Table : out T_Frequences
- Fin Tant Que

R2 : Comment "initialiser les compteurs" ?

- nb_suppression <- 0
- Longueur <- length(Copie_Texte)

R2 : Comment "compter la fréquence du Caractere" ?

- Pour i allant de 1 à Longueur faire
    - Augmenter de 1 le compteur Nb_Supprimer si le i-ième caractère du texte est identique à Caractere
- Fin Pour

Lauriol François
Gras Yael

R2 : Comment “Incrémenter la table des fréquences”?

- Table.Taille <- Table.Taille+1
- Table.Tableau(Table.Taille).Caractere <- Caractere
- Table.Tableau(Table.Taille).Frequence <- nb_suppression

R3 : Comment “Augmenter de 1 le compteur Nb_Supprimer si le i-ième caractère du texte est identique à Caractere” ?

- Si Copie_Texte(i - nb_suppresion) = Caractere alors
  - nb_suppression <- nb_suppression +1
  - Copie_Texte <- Delete(Copie_Texte, i-nb_suppression, i-nb_suppression)
- Sinon
  - Rien
- Fin Si

R0 : Construire l’arbre de Huffman à partir du Tableau des fréquences des caractères (CONSTRUIRE_ARBRE)

Arbre_Huff : out ARBRE
Table : in T_FREQUENCES

R1 : Comment “construire l’arbre de Huffman à partir du Tableau des fréquences des caractères” ?

- LISTER_FEUILLES(Table, Liste_Arbres) Table : in T_Frequences; Liste_Arbres : out L_Arbres
- TantQue Liste_Arbres.Taille>1 Faire Liste_Arbres : in L_Arbres
  - Déterminer deux Arbres ayant les deux plus petites fréquences de Liste_Arbres indice_Arbre1, indice_Arbre2 : out Integer ; Liste_Arbres : in L_Arbres; freqmin1, freqmin2 : in out Integer
  - ASSEMBLER_2ARBRES (Liste_Arbres, indice_Arbre1, indice_Arbre2) indice_Arbre1, indice_Arbre2 : in Integer ; Liste_Arbres : in out L_Arbres
  - Placer le dernier Arbre de Liste_Arbre à la place du second arbre. Liste_Arbres : in out L_Arbres
- Fin TQ
- Attribuer à Arbre_Huff sa valeur. Arbre_Huff : out ARBRE ; Liste_Arbres : in out L_Arbres

R2 : Comment “déterminer deux Arbres ayant les deux plus petites fréquences de Liste_Arbres” ?

- Initialiser les fréquences minimales avec freqmin1<freqmin2
- Pour indice allant de 3 à Liste_Arbres.Taille Faire
    - Déterminer si on a un arbre avec une plus faible fréquence
- Fin Pour

R2 : Comment “placer le dernier Arbre de Liste_Arbre à la place du second arbre” ?

- Liste_Arbres.Foret(indice_Arbre2) <- Liste_Arbres.Foret(Liste_Arbre.Taille)
- Liste_Arbres.Foret(Liste_Arbres.Taille) <- null
- Liste_Arbres.Taille <- Liste_Arbres.Taille - 1

R2 : Comment “attribuer à Arbre_Huff sa valeur” ?

- Arbres_Huff <- Liste_Arbres.Foret(1)
- Liste_Arbres.Foret(1) <- null
- Liste_Arbres.Taille <- 0

R3 : Comment “Initialiser les fréquences minimales avec freqmin1<freqmin2” ?

- Si Liste_Arbres.Foret(1)^.Frequence<=Liste_Arbres.Foret(2)^.Frequence Alors
    - Initialiser les fréquences minimales avec les deux premiers éléments dans l’ordre croissant
- Sinon
    - Initialiser les fréquences minimales avec les deux premiers éléments dans l’ordre décroissant
- Fin Si

R3 : Comment “Déterminer si on a un arbre avec une plus faible fréquence” ?

- Si freqmin1 <= Liste_Arbres.Foret(indice)^.Frequence et freqmin2 > Liste_Arbres.Foret(indice)^.Frequence Alors
    - freqmin2 <- Liste_Arbres.Foret(indice)^.Frequence
    - indice_Arbre2 <- indice
- Sinon si freqmin1 > Liste_Arbres.Foret(indice)^.Frequence Alors
    - freqmin2 <- freqmin1
    - indice_Arbre2 <- indice_Arbre1
    - freqmin1 <- Liste_Arbres.Foret(indice)^.Frequence
    - indice_Arbre1 <- indice
- Fin Si

Lauriol François Groupe MN - 1
Gras Yael

R4 : Comment “Initialiser les fréquences minimales avec les deux premiers éléments dans l’ordre croissant” ?

- freqmin1 <- Liste_Arbres.Foret(1)^.Frequence
- freqmin2 <- Liste_Arbres.Foret(2)^.Frequence
- indice_Arbre1 <- 1
- indice_Arbre2 <- 2

R4 : Comment “Initialiser les fréquences minimales avec les deux premiers éléments dans l’ordre décroissant” ?

- freqmin1 <- Liste_Arbres.Foret(2)^.Frequence
- freqmin2 <- Liste_Arbres.Foret(1)^.Frequence
- indice_Arbre1 <- 2
- indice_Arbre2 <- 1

R0 : Vider l’espace mémoire occupé par l’arbre de Huffman (VIDER_ARBRE)

Arbre_Huff : in out ARBRE

R1 : Comment “vider l’espace mémoire occupé par l’arbre de Huffman”?

- Si Arbre_Huff /= null alors
  - Vider(Arbre_Huff^.Fils_Gauche)
  - Vider(Arbre_Huff^.Fils_Droit)
  - Free(Arbre)
- Sinon
  - Rien
- Fin Si

R0 : Former la table de Huffman Dict grâce à son arbre correspondant Arbre_Huff (FABRIQUER_DICT)

Dict : in out T_DICTIONNAIRE
Arbre_Huff : in ARBRE

R1 : Comment “former la table de Huffman Dict grâce à son arbre correspondant Arbre_Huff” ?

- Initialiser la table de Huffman Dict Dict : out T_Dictionnaire; Position : out T_Octet

- Si Arbre_Huff^.Fils_G=null alors ---- Cas du fichier vide
    - Dict(1) <- new T_Cellule
    - Dict(1)^.Caractere <- Caractère'val(0)
    - Dict(1)^.Code_Huff <- To_Unbounded_String("0")
    - Dict(1)^.Suivant <- null
- Sinon
    - FABRIQUER_DICT_RECURSIVE(Dict, Arbre_Huff, To_Unbounded_String(""), Position) Dict : in out T_DICTIONNAIRE ; Arbre_Huff : in ARBRE; Position : in out T_Octet
- Fin Si

R2 : Comment "initialiser la table de Huffman Dict" ?

- Pour k allant de 1 à 27 Faire
    - Dict(k) <- null
- Fin Pour
- Position <- T_Octet(0)

R0 : Retourner le codage de Huffman d'un caractère (LA_DONNEE_DICT)
Dict : in T_DICTIONNAIRE
Caractere : in Caractère
Fin : in BOOLÉEN

R1 : Comment "retourner le codage de Huffman d'un caractère" ?

- Si Fin Alors Fin : in BOOLÉEN
    - Retourne Dict(1)^.Code_Huff Dict : in T_DICTIONNAIRE
- Fin Si
- Retourne Code_LCA(Caractere, Dict(HACHAGE(Caractere, False))) Dict : in T_DICTIONNAIRE; Caractere : in Caractère

R0 : Retourner le caractère d'un codage de Huffman (LE_CARACTERE_DICT)
Dict : in T_DICTIONNAIRE
Code : in UNBOUNDED_STRING

R1 : Retourner le caractère d'un codage de Huffman

- indice <- 1 indice : out Entier
- Répéter

    - Caractere_LCA(Code, Dict(indice), Caractere, Trouve) Code : in UNBOUNDED_STRING ; Dict : in T_DICTIONNAIRE ; indice : in Entier ; Trouve : out Booléen ; Caractere : out Caractère
    - indice <- indice + 1 indice : in out Entier
- Jusqu'à Trouve Trouve : in Booléen
- Retourne Caractere Caractere : in Caractère

R0 : Vider l'espace mémoire occupé par le Dictionnaire Dict (VIDER_DICT)
Dict : in out T_DICTIONNAIRE

R1 : Comment "vider l'espace mémoire occupé par l'arbre de Huffman" ?
- Pour i allant de 1 à 27 Faire
    - VIDER_LCA(Dict(i))
- Fin Pour

R0 : Déterminer si le Code est bien un code de Huffman du Dictionnaire (CODE_EST_PRESENT_DICT)

R1 : Comment "déterminer si le Code est bien un code de Huffman du Dictionnaire" ?
- indice <- 1 indice : out Entier
- Répéter
    - Caractere_LCA(Code, Dict(indice), Caractere, Trouve) Code : in UNBOUNDED_STRING ; Dict : in T_DICTIONNAIRE ; indice : in Entier ; Trouve : out Booléen ; Caractere : out Caractère
    - indice <- indice + 1 indice : in out Entier
- Jusqu'à Trouve ou indice>27 Trouve : in Booléen ; indice : in Entier
- Retourne Trouve Trouve : in Booléen

R0 : Extraire l'arbre de Huffman d'un texte préalablement compressé (RECUPERER_ARBRE)
Arbre_Huff : out ARBRE
Texte : in out UNBOUNDED_STRING

R1 : Comment "extraire l'arbre de Huffman d'un texte préalablement compressé" ?
- Si length(Texte)=16 Alors Texte : in UNBOUNDED_STRING

---- cas où le fichier est vide,

- Extraire l'arbre de Huffman d'un Fichier txt initialement vide Arbre_Huff : out ARBRE ; Texte : out UNBOUNDED_STRING
- Sinon
  - Extraire l'arbre de Huffman d'un Fichier txt lorsqu'il n'est pas initialement vide Liste_Caracteres : in out L_Caracteres; Arbre_Huff : out ARBRE ; Texte : in out UNBOUNDED_STRING; nbr0 : in out Entier; nbr1 : in out Entier; Liste_Complets : in out L_Complets; Liste_Arbre : L_Arbres
- Fin Si

R2 : Comment "extraire l'arbre de Huffman d'un Fichier txt initialement vide" ?

- Arbre_Huff <- NEW T_FEUILLE
- Arbre_Huff^.Caractere <- Character'val(0)
- Arbre_Huff^.Frequence <- 0
- Arbre_Huff^.Fils_G <- null
- Arbre_Huff^.Fils_D <- null
- Texte <- To_Unbounded_String("0000000")

R2 : Comment "extraire l'arbre de Huffman d'un Fichier txt lorsqu'il n'est pas initialement vide" ?

- Obtenir la portion des caractères de la table de Huffman
- Créer l'arbre de Huffman à l'aide de sa forme binaire

R3 : Comment "obtenir la portion des caractères de la table de Huffman" ?

- Mettre le '\$' en première position de la liste des caracteres
- TantQue To_String(Texte)(1..8)/=To_String(Texte)(9..16) Faire
  - Liste_Caracteres.Taille <- Liste_Caracteres.Taille+1
  - Liste_Caracteres.Caracteres(Liste_Caracteres.Taille) <- CONVERTIR_BINAIRE_CARACTERE(To_Unbounded_String(To_String(Texte)(1..8)))
  - Texte <- Delete(Texte, 1, 8)
- Fin TQ
- Mettre le dernier caractère dans la liste

R3 : Comment "créer l'arbre de Huffman à l'aide de sa forme binaire" ?

- Initialiser les compteurs
- TantQue nbr0>=nbr1 Faire
  - Agrandir l'arbre selon le bit lu
  - Delete(Texte, 1, 1)
- Fin TQ
- Arbre_Huff <- Liste_Arbre.Foret(1)

- Liste_Arbre.Foret(1) <- null

R4 : Comment “mettre le ‘\$’ en première position de la liste des caracteres” ?

- Liste_Caracteres.Taille <- 1
- Liste_Caracteres.Caracteres(1)<-CONVERTIR_BINAIRE_CARACTERE(To_Unbounded_String(To_String(Texte)(1..8))) ---- Position dans l’arbre du caractère ‘\$’ sous la forme d’un caractère
- Texte <- Delete(Texte, 1, 8)

R4 : Comment “mettre le dernier caractère dans la liste” ?

- Liste_Caracteres.Taille <- Liste_Caracteres.Taille+1
- Liste_Caracteres.Caracteres(Liste_Caracteres.Taille) <- CONVERTIR_BINAIRE_CARACTERE(To_Unbounded_String(To_String(Texte)(1..8)))
- Texte <- Delete(Texte, 1, 16)

R4 : Comment “initialiser les compteurs” ?

- nbr0 <- 0
- nbr1 <- 0
- Liste_Arbre.Taille <- 0
- Liste_Complets.Taille <- 0

R4 : Comment “agrandir l’arbre selon le bit lu” ?

- Si To_String(Texte)(1)=’0’ Alors
  - Rajouter une racine à l’arbre
- Sinon
  - Rajouter une feuille à l’arbre
- Fin Si

R5 : Comment “rajouter une racine à l’arbre” ?

- nbr0 <- nbr0 + 1
- Liste_Complets.Taille <- Liste_Complets.Taille+1
- Liste_Complets.Complets(Liste_Complets.Taille) <-False

R5 : Comment “rajouter une feuille à l’arbre” ?

- nbr1 <- nbr1+1

- Créer la feuille selon sa position dans la portion des caractères de la table de Huffman
- Ajouter la feuille à l'arbre
- Si Liste_Complets.Taille>0 Alors ---- Liste_Complets.Taille=0 est équivalent à nbr0=nbr1+1
    - Liste_Complets.Complets(Liste_Complets.Taille) <-True ----On indique que la dernière racine a sa branche gauche complète
- Fin Si

R6 : Comment "créer la feuille selon sa position dans la portion des caractères de la table de Huffman" ?

- Liste_Arbre.Taille<-Liste_Arbre.Taille+1
- Liste_Arbre.Foret(Liste_Arbre.Taille) <- NEW T_FEUILLE
- Si nbr1=Character'pos(Liste_Caracteres.Caracteres(1)) Alors
    - Donner les caractéristiques de la feuille '\$'
- Sinon
    - Donner les caractéristiques de la feuille non '\$'
- Fin Si
- Liste_Arbre.Foret(Liste_Arbre.Taille)^.Fils_G <- null
- Liste_Arbre.Foret(Liste_Arbre.Taille)^.Fils_D <- null

R6 : Comment "ajouter la feuille à l'arbre" ?

- TantQue Liste_Complet.Taille>0 et alors Liste_Complets.Complets(Liste_Complets.Taille) Faire ---- On vérifie que la dernière racine a sa branche gauche complète
    - ASSEMBLER_2ARBRES(Liste_Arbres, Liste_Arbre.Taille-1, Liste_Arbre.Taille) ----On assemble les deux dernières racines en 1 seule
    - Liste_Arbre.Foret(Liste_Arbre.Taille) <- null
    - Liste_Arbre.Taille <- Liste_Arbre.Taille - 1
    - Liste_Complets.Taille <- Liste_Complets.Taille-1
- Fin TQ

R7 : Comment "donner les caractéristiques de la feuille '\$'" ?

- Liste_Arbre.Foret(Liste_Arbre.Taille)^.Frequence <- 0
- Liste_Arbre.Foret(Liste_Arbre.Taille)^.Caractere <- Character'val(0)

R7 : Comment "donner les caractéristiques de la feuille non '\$'" ?

- Liste_Arbre.Foret(Liste_Arbre.Taille)^.Frequence <- 1
- Affecter le caractère selon sa position dans la portion des caractères de la table de Huffman

Lauriol François
Gras Yael

R8 : Comment “affecter le caractère selon sa position dans la portion des caractères de la table de Huffman” ?

- Si nbr1<Character’pos(Liste_Caracteres.Caracteres(1))
    - Liste_Arbre.Foret(Liste_Arbre.Taille)^.Caractere <- Liste_Caracteres.Caracteres(nbr1+1)
- Sinon
    - Liste_Arbre.Foret(Liste_Arbre.Taille)^.Caractere <- Liste_Caracteres.Caracteres(nbr1)
- Fin Si

R0 : Renvoyer le codage binaire de la forme de l’arbre de Huffman
(ARBRE_EN_BINAIRE)

Arbre_Huff : in ARBRE

R1 : Comment “renvoyer le codage binaire de la forme de l’arbre de Huffman” ?

- Si Arbre_Huff^.Fils_G=null Alors ---- équivalent à Arbre_Huff^.Fils_D=null par construction
    - Retourne To_Unbounded_String(“1”)
- Fin Si
- Retourne To_Unbounded_String(“0”) & ARBRE_EN_BINAIRE(Arbre_Huff^.Fils_G) & ARBRE_EN_BINAIRE(Arbre_Huff^.Fils_D)

R0 : Renvoyer le codage binaire de la forme de la table de Huffman
(TABLE_EN_BINAIRE)

Dict : in T_DICTIONNAIRE

R1 : Comment “renvoyer le codage binaire de la forme de la table de Huffman” ?

- Texte <- To_Unbounded_String(“” & Dict(1)^.Caractere) Texte : out UNBOUNDED_STRING
- Déterminer si le fichier est non vide Fichier_vide : out Booléen
- Si Non Fichier_vide Alors
    - AJOUTER_CARACTERE_CODE(To_Unbounded_String(“”), Texte, Dict, Dict(1)^.Code) Dict : in T_DICTIONNAIRE; Texte : in out UNBOUNDED_STRING
    - Texte <- Texte & To_String(Texte)(length(Texte)) Texte : in out UNBOUNDED_STRING
- Fin Si

- CONVERTIR_BINAIRE(Texte) Texte : in out UNBOUNDED_STRING
- Retourne Texte Texte : in UNBOUNDED_STRING

R2 : Comment “déterminer si le fichier est non vide” ?

- Fichier_vide <- True
- Pour k allant de 2 à 27 Faire
  - Déterminer si la k-ième file du dictionnaire est vide
- Fin Pour

R3 : Comment “déterminer si la k-ième file du dictionnaire est vide” ?

- Si Dict(k)/=null Alors
  - Fichier_vide <- False
- Fin Si

R0 : Afficher l’arbre de Huffman comme la figure 20 du polycopié de projet
(AFFICHER_ARBRE)

Arbre_Huff : in ARBRE

R1 : Comment “afficher l’arbre de Huffman comme la figure 20 du polycopié de projet” ?

- Si Arbre_Huff^.Fils_G/=null Alors ---- équivalent à Arbre_Huff^.Fils_D/=null
  - Afficher la fréquence totale
  - AFFICHER_ARBRE_RECURSIF(To_Unbounded_String(“0”),Arbre_Huff^.Fils_G)
  - AFFICHER_ARBRE_RECURSIF(To_Unbounded_String(“1”),Arbre_Huff^.Fils_D)
- Sinon
  - Écrire(“- - 0 - - (0) ‘\$’”)
  - Nouvelle Ligne
- Fin Si

R2 : Comment “Afficher la fréquence totale”?

- Écrire(“(” & Entier’Image(Arbre_Huff^.Frequence) & “)”)
- Nouvelle Ligne

Lauriol François
Gras Yael

Groupe MN - 1

R0 : Afficher la table de Huffman comme la figure 21 du polycopié de projet (AFFICHER_DICT)

Dictionnaire : in T_DICTIONNAIRE

R1 : Comment “afficher la table de Huffman comme la figure 21 du polycopié de projet” ?

- Écrire(“'\$' --> ” & To_String(LA_DONNEE_DICT(Dict, ‘a’, True)))
- Pour i allant de 1 à 256
    - DÉBUT
        - Écrire le caractère et son code correspondant
    - EXCEPTION
        - Exception_Donnee_Absente => null;
    - FIN
- Fin Pour

R2 : Comment “écrire le caractère et son code correspondant” ?

- Si i=10 Alors
    - Écrire(“'\n' --> ” & To_String(LA_DONNEE_DICT(Dict, Character’val(10), False)))
- Sinon
    - Écrire(“'” & Character’val(i) & ”' --> ” & To_String(LA_DONNEE_DICT(Dict, Character’val(i), False)))
- Fin Si

R0 : Convertir un texte de caractères en binaire avec leur code ASCII binaire correspondant. (CONVERTIR_BINAIRE)

Texte : in out UNBOUNDED_STRING

R1 : Comment “convertir un texte de caractères en binaire avec leur code ASCII binaire correspondant” ?

- Texte_inter <- To_Unbounded_String(“”): Texte_inter : out Unbounded_String
- Pour k allant de 1 à length(Texte) Texte : in UNBOUNDED_STRING
    - Texte_inter <- Texte_inter & CONVERTIR_CARACTERE_BINAIRE(To_String(Texte)(k)) Texte_inter : in out Unbounded_String; Texte : in UNBOUNDED_STRING
- Fin Pour

- Texte <- Texte_inter Texte : out UNBOUNDED_STRING; Texte_inter : in Unbounded_String

R0 : Convertir un texte en binaire en caractères avec leur code ASCII binaire correspondant. (CONVERTIR_CARACTERES)

Texte : in out UNBOUNDED_STRING

R1 : Comment “convertir un texte de caractères en binaire avec leur code ASCII binaire correspondant” ?

- Texte_inter <- To_Unbounded_String(“”) Texte_inter : out Unbounded_String
- Texte <- Texte & (8 - ((Length(Texte)-1) mod 8 + 1))*”0” Texte : in out UNBOUNDED_STRING
- Pour k allant de 1 à length(Texte)/8 Texte : in UNBOUNDED_STRING
    - Texte_inter <- Texte_inter & CONVERTIR_BINAIRE_CARACTERE(To_Unbounded_String(To_String(Texte)((8*(k-1)+1)..8*k))) Texte_inter : in out Unbounded_String; Texte : in UNBOUNDED_STRING
- Fin Pour
- Texte <- To_Unbounded_String(Texte_inter) Texte_inter : in String; Texte : out UNBOUNDED_STRING

Lauriol François
Gras Yael

TYPES DE L'ADB

type A_Arbres est TABLEAU de (1..256) de Arbre

type L_Arbres est ENREGISTREMENT
    Taille : entier {0 <= Taille et Taille <= 256}
    Foret : A_Arbres
FIN ENREGISTREMENT

type A_Caracteres est TABLEAU de (1..257) de Caractère

type L_Caracteres est ENREGISTREMENT
    Taille : entier {0 <= Taille et Taille <= 257}
    Caracteres : A_Caracteres
FIN ENREGISTREMENT

type A_Complets est Tableau de (1..256) de Booléen ---- la branche de gauche est pleine

type L_Complets est ENREGISTREMENT
    Taille : Entier
    Complets : A_Complets
FIN ENREGISTREMENT

Lauriol François
Gras Yael

FONCTIONS DE L'ADB

—-- nom : HACHAGE
---- sémantique : Retourner l'indice d'un caractère dans un Dictionnaire
---- paramètres : Caractere : in Caractère
---- Fin : in BOOLÉEN -- Si on demande pour le caractère '\$'
---- Type de retour : Entier
---- post-condition : Fin=True et Résultat=1 ou alors 2<=Résultat<=27
---- tests : Entrée : Caractere='a' & Fin=True => Résultat=1
---- Entrée : Caractere='a' & Fin=False => Résultat=21
---- Entrée : Caractere='g' & Fin=False => Résultat=27
---- Entrée : Caractere='h' & Fin=False => Résultat=2
fonction HACHAGE(Caractere : in Caractère; Fin : in BOOLÉEN) retourne Entier

---- nom : ASSEMBLER_2ARBRES
---- sémantique : Construire une racine avec ces deux arbres dans le premier Arbre
---- paramètres : Liste_Arbres : in out L_Arbres
---- indice_Arbre1, indice_Arbre2 : in Integer
---- pré-condition : Liste_Arbres.Foret(indice_Arbre1)^.Frequence <=
---- Liste_Arbres.Foret(indice_Arbre2)^.Frequence
---- post-condition : Liste_Arbres.Foret(indice_Arbre1)^.Fils_G= Old'
---- Liste_Arbres.Foret(indice_Arbre1)
---- & Liste_Arbres.Foret(indice_Arbre1)^.Fils_D = Old'
---- Liste_Arbres.Foret(indice_Arbre2) &
---- Liste_Arbres.Foret(indice_Arbre2)=null &
---- Liste_Arbres.Foret(indice_Arbre1)^.Frequence =
---- Old'Liste_Arbres.Foret(indice_Arbre1)^.Frequence+
---- Old'Liste_Arbres.Foret(indice_Arbre2)^.Frequence
procedure ASSEMBLER_2ARBRES(Liste_Arbres : in out L_Arbres; indice_Arbre1 : in Integer; indice_Arbre2 : in Integer)

---- nom : LISTER_FEUILLES
---- sémantique : Créer à partir d'une Table des Fréquences un ensemble contenant toutes
---- les Feuilles de notre Arbre de Huffman
---- paramètres : Table : in T_Frequences
---- Liste_Arbres : out L_Arbres
---- post-condition : Liste_Arbres.Taille=Table.Taille+1 et pour tout i allant de 1 à Table.taille
---- Table.Tableau(i).Frequence = Liste_Arbres.Foret(i)^.Frequence &
---- Table.Tableau(i).Caractere = Liste_Arbres.Foret(i)^.Caractere &
---- Liste_Arbres.Foret(i)^.Fils_G=null & Liste_Arbres.Foret(i)^.Fils_D=null &
---- Liste_Arbres.Foret(Table.Taille+1)^.Frequence=0 &
---- Liste_Arbres.Foret(Table.Taille+1)^.Caractere=Character'val(0) &
---- Liste_Arbres.Foret(Table.Taille+1)^.Fils_G=null &
---- Liste_Arbres.Foret(Table.Taille+1)^.Fils_D=null
procedure LISTER_FEUILLES(Table : in T_Frequences; Liste_Arbres : out L_Arbres)

Lauriol François
Gras Yael

—-- nom : FABRIQUER_DICT_RECURSIVE
---- sémantique : Former la table de Huffman Dict grâce à son arbre correspondant
---- Arbre_Huff avec Dict déjà initialisé
---- paramètres : Arbre_Huff : in ARBRE
---- Dict : in out T_DICTIONNAIRE
---- Code : in UNBOUNDED_STRING
---- tests : Entrées 0 caractère : Dict & Arbre_Huff^.Frequence=0 &
---- Character‘pos(Arbre_Huff^.Caractere)=0 &
---- Arbre_Huff^.Fils_G=null &
---- Arbre_Huff^.Fils_D=null => pour tout i dans 2..27 on a
---- Dict(i)=null & Character‘pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String(“0”)
---- Entrées 2 caractères : Arbre_Huff^.Frequence=3 &
---- Arbre_Huff^.Fils_D^.Frequence=2 &
---- Arbre_Huff^.Fils_D^.Caractere=’a’ &
---- Arbre_Huff^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Frequence=1 &
---- Arbre_Huff^.Fils_G^.Fils_D^.Caractere=’b’ &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_D^.Fils_D=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Frequence=0 &
---- Character‘pos(Arbre_Huff^.Fils_G^.Fils_G^.Caractere)=0
&
---- Arbre_Huff^.Fils_G^.Fils_G^.Fils_G=null &
---- Arbre_Huff^.Fils_G^.Fils_G^.Fils_D=null => pour tout i
dans
---- 2..27 différent de 21, 22 on a Dict(i)=null &
---- Dict(21)^.Caractere=’a’ &
---- Dict(21)^.Code_Huff=To_Unbounded_String(“1”)&
---- Dict(22)^.Caractere=’b’ &
---- Dict(22)^.Code_Huff=To_Unbounded_String(“01”)&
---- Character‘pos(Dict(1)^.Caractere)=0 &
---- Dict(1)^.Code_Huff=To_Unbounded_String(“00”)
procedure FABRIQUER_DICT_RECURSIVE(Dict : in out T_DICTIONNAIRE; Arbre_Huff : in
ARBRE; Code : in UNBOUNDED_STRING)

—-- nom : CODE_LCA
---- sémantique : Retourner le code correspondant d’un caractère différent de ‘\$’ dans un
---- Dictionnaire
---- paramètres : Caractere : in Caractère
---- Sda : in LCA
---- Type de retour : UNBOUNDED_STRING

fonction CODE_LCA(Caractere : in Caractère; Sda : in LCA) retourne UNBOUNDED_STRING

—-- nom : CARACTERE_LCA
---- sémantique : Chercher le caractere correspondant d'un code de Huffman dans un
---- Dictionnaire
---- paramètres : Code : in UNBOUNDED_STRING
---- Sda : in LCA
---- Caractere : out Caractere
---- Trouve : out Booléen ---- indique si le caractère a été trouvé
procedure CARACTERE_LCA(Code : in UNBOUNDED_STRING ; Sda : in LCA ; Caractere : out Caractere ; Trouve : out Booléen)

—-- nom : VIDER_LCA
---- sémantique : Vider l'espace mémoire occupé par une LCA
---- paramètres : Sda : in out LCA
---- post-condition : Sda=null
---- tests : Entrées Quelconque : Sda => Sda=null
procedure VIDER_DICT(Sda : in out LCA)

---- nom : CONVERTIR_BINAIRE_CARACTERE
---- sémantique : Renvoyer d'un octet (unbounded_string de 8 bits) son caractère
---- correspondant.
---- paramètres : Texte_Binaire : in UNBOUNDED_STRING
---- Type de retour : Caractère
---- pré-condition : length(Texte_Binaire)=8
fonction CONVERTIR_BINAIRE_CARACTERE(Texte_Binaire : in UNBOUNDED_STRING) retourne Caractère

---- nom : AJOUTER_CARACTERE_CODE
---- sémantique : Ajouter tous les caractères de manière récursive dans le Texte
---- paramètres : Code : in UNBOUNDED_STRING
---- Texte : in out UNBOUNDED_STRING
---- Dict : in T_DICTIONNAIRE
---- Code_Fin : in UNBOUNDED_STRING ---- Code de Huffman de '\$'
procedure AJOUTER_CARACTERE_CODE(Code : in UNBOUNDED_STRING; Texte : in out UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE; Code_Fin : in UNBOUNDED_STRING)

---- nom : AFFICHER_ARBRE_RECURSIF
---- sémantique : Afficher l'arbre de Huffman comme la figure 20 du polycopié de projet de
---- manière récursive à partir du code Code
---- paramètres : Code : in UNBOUNDED_STRING
---- Arbre_Huff : in ARBRE

Lauriol François
Gras Yael

procedure AFFICHER_ARBRE_RECURSIF(Code : in UNBOUNDED_STRING; Arbre_Huff : in ARBRE)

---- nom : CONVERTIR_CARACTERE_BINAIRE
---- sémantique : Renvoyer d'un caractère son octet (unbounded_string de 8 bits)
---- correspondant.
---- paramètres : Caractere : in Caractère
---- Type de retour : UNBOUNDED_STRING
---- post-condition : CONVERTIR_BINAIRE_CARACTERE(Résultat)=Caractere
fonction CONVERTIR_CARACTERE_BINAIRE(Caractere : in Caractère) retourne UNBOUNDED_STRING

Lauriol François
Gras Yael

Groupe MN - 1

RAFFINAGES DES FONCTIONS DE l'ADB DU MODULE

R0 : Retourner l'indice d'un caractère dans un Dictionnaire (HACHAGE)

Caractere : in Caractère
Fin : in BOOLÉEN

R1 : Comment "Retourner l'indice d'un caractère dans un Dictionnaire" ?

- Si Fin Alors — Fin : in BOOLÉEN
  - Retourne 1
- Fin Si
- Retourne Character'pos(Caractere) mod 26 +2 — Caractere : in Caractère

R0 : Construire une racine avec ces deux arbres dans le premier Arbre (ASSEMBLER_2ARBRES)

Liste_Arbres : in out L_Arbres
indice_Arbre1 : in Integer
indice_Arbre2 : in Integer

R1 : Comment "construire une racine avec ces deux arbres dans le premier Arbre" ?

- Arbre_inter <- NEW T_FEUILLE
- Arbre_inter^.Frequence <- Liste_Arbres.Foret(indice_Arbre1)^.Frequence + Liste_Arbres.Foret(indice_Arbre2)^.Frequence
- Arbre_inter^.Fils_G <- Liste_Arbres.Foret(indice_Arbre1)
- Arbre_inter^.Fils_D <- Liste_Arbres.Foret(indice_Arbre2)
- Liste_Arbres.Foret(indice_Arbre1) <- Arbre_inter
- Liste_Arbres.Foret(indice_Arbre2) <- null

R0 : Créer à partir d'une Table des Fréquences un ensemble contenant toutes les Feuilles de notre Arbre de Huffman (LISTER_FEUILLES)

Table : in T_Frequences
Liste_Arbres : out L_Arbres

R1 : Comment "créer à partir d'une Table des Fréquences un ensemble contenant toutes les Feuilles de notre Arbre de Huffman" ?

- Listes_Arbres.Taille <- Table.Taille+1 ---- Liste_Arbres : out L_Arbres; Table : in T_Frequences
- Pour k allant de 1 à Table.Taille Faire
    - Ajouter le k-ième case du table à la liste des feuilles ---- Liste_Arbres : out L_Arbres; Table : in T_Frequences
- Fin Pour
- Ajouter le caractère '\$' aux feuilles ---- Liste_Arbres : out L_Arbres

R2 : Comment "ajouter le k-ième case du table à la liste des feuilles" ?

- Listes_Arbres.Foret(k) <- New T_Feuille
- Listes_Arbres.Foret(k)^.Frequence <- Table.Tableau(k).Frequence
- Listes_Arbres.Foret(k)^.Caractere <- Table.Tableau(k).Caractere
- Listes_Arbres.Foret(k)^.Fils_G <- null
- Listes_Arbres.Foret(k)^.Fils_D <- null

R2 : Comment "ajouter le caractère '\$' aux feuilles" ?

- Listes_Arbres.Foret(Listes_Arbres.Taille) <- New T_Feuille
- Listes_Arbres.Foret(Listes_Arbres.Taille)^.Frequence <- 0
- Listes_Arbres.Foret(Listes_Arbres.Taille)^.Caractere <-'a'
- Listes_Arbres.Foret(Listes_Arbres.Taille)^.Fils_G <- null
- Listes_Arbres.Foret(Listes_Arbres.Taille)^.Fils_D <- null

R0 : Former la table de Huffman Dict grâce à son arbre correspondant Arbre_Huff avec Dict déjà initialisé (FABRIQUER_DICT_RECURSIVE)

Dict : in out T_DICTIONNAIRE
Arbre_Huff : in ARBRE
Code : in UNBOUNDED_STRING
Position : in out T_Octet ---- position de la feuille

R1 : Comment "former la table de Huffman Dict grâce à son arbre correspondant Arbre_Huff avec Dict déjà initialisé" ?

- Feuille <- (Arbre_Huff^.Fils_G=null) Feuille : out Booléen; Arbre_Huff : in Arbre ---- La construction de l'arbre ne permettant pas qu'un arbre ait une racine à gauche et pas à droite et vice-versa
- Si Feuille Alors Feuille : in Booléen
    - Ajouter la feuille au Dictionnaire Dict : in out T_DICTIONNAIRE ; Arbre_Huff : in Arbre ; pointeur : in out LCA ; Position : out T_Octet
- Sinon
    - FABRIQUER_DICT_RECURSIVE(Dict, Arbre_Huff^.Fils_G, Code & "0", Position) Dict : in out T_DICTIONNAIRE ; Arbre_Huff : in Arbre ; Position in out T_Octet

    - FABRIQUER_DICT_RECURSIVE(Dict, Arbre_Huff^.Fils_D, Code & “1”, Position) Dict : in out T_DICTIONNAIRE ; Arbre_Huff : in Arbre ; Position in out T_Octet
- Fin Si

R2 : Comment “ajouter la feuille au Dictionnaire” ?

- Si Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))=null alors
    - Mettre le premier caractère de la file du Dictionnaire
- Sinon
    - Déterminer le dernier caractère de la file du Dictionnaire
    - Mettre le caractere à la fin de la file du Dictionnaire
- Fin si
- Position <- Position + 1

R3 : Comment “mettre le premier caractère de la file du Dictionnaire” ?

- Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0)) <- NEW T_Cellule
- Si Arbre_Huff^.Frequence = 0 alors
    - Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))^.Caractere <- Character’val(Position)
- Sinon
    - Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))^.Caractere <- Arbre_Huff^.Caractere
- Fin Si
- Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))^.Code_Huff <- Code
- Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))^.Suivant <- null

R3 : Comment “déterminer le dernier caractère de la file du Dictionnaire” ?

- pointeur <- Dict(HACHAGE(Arbre_Huff^.Caractere, Arbre_Huff^.Frequence = 0))
- TantQue pointeur^.Suivant /= null Faire
    - pointeur<-pointeur^.Suivant
- Fin TQ

R3 : Comment “mettre le caractere à la fin de la file du Dictionnaire” ?

- pointeur^.Suivant <- NEW T_Cellule
- Si Arbre_Huff^.Frequence = 0 alors
    - pointeur^.Suivant^.Caractere <- Character’val(Position)

- Sinon
    - pointeur^.Suivant^.Caractere <- Arbre_Huff^.Caractere
- Fin Si
- pointeur^.Suivant^.Code_Huff <- Code
- pointeur^.Suivant^.Suivant <- null

R0 : Retourner le code correspondant d'un caractère différent de '\$' dans un Dictionnaire (CODE_LCA)

Caractere : in Caractère
Sda : in LCA

R1 : Comment "retourner le code correspondant d'un caractère différent de '\$' dans un Dictionnaire" ?

- Si Sda = null Alors — Sda : in LCA
    - LEVER Exception_Donnee_Absente
- Sinon si Sda^.Caractere = Caractere Alors — Sda : in LCA; Caractere : in Caractère
    - Retourne Sda^.Code_Huff — Sda : in LCA
- Fin Si
- Retourne CODE_LCA(Caractere, Sda^.Suivant) — Sda : in LCA; Caractere : in Caractère

R0 : Chercher le caractere correspondant d'un code de Huffman dans un Dictionnaire (CARACTERE_LCA)

Code : in UNBOUNDED_STRING
Sda : in LCA
Caractere : out Caractere
Trouve : out Booléen

R1 : Comment "chercher le caractere correspondant d'un code de Huffman dans un Dictionnaire" ?

- Si Sda = null Alors — Sda : in LCA
    - Trouve <- False — Trouve : out Booléen
    - Caractere <- 'a' — Caractere : out Caractere
- Sinon si Sda^.Code_Huff = Code Alors — Sda : in LCA; Code : in UNBOUNDED_STRING
    - Trouve <- True — Trouve : out Booléen
    - Caractere <- Sda^.Caractere — Sda : in LCA; Caractere : out Caractere
- Sinon

    - CARACTERE_LCA(Code, Sda^.Suivant, Caractere, Trouve) Code : in UNBOUNDED_STRING; Sda : in LCA; Caractere : out Caractere; Trouve : out Booléen
- Fin Si

R0 : Vider l'espace mémoire occupé par une LCA (VIDER_LCA)

Sda : in out LCA

R1 : Comment "vider l'espace mémoire occupé par une LCA" ?

- Si Sda /= null alors Sda : in LCA
    - Vider_Lca(Sda^.Suivant) Sda : in out LCA
    - Free(Sda) Sda : out LCA
- Sinon
    - Rien
- Fin Si

R0 : Renvoyer d'un octet (unbounded_string de 8 bits) son caractère correspondant. (CONVERTIR_BINAIRE_CARACTERE)

Texte_Binaire : in UNBOUNDED_STRING

R1 : Comment "renvoyer d'un octet (unbounded_string de 8 bits) son caractère correspondant" ?

- indice <-0 indice : out T_Octet
- Pour k allant de 1 à 8 Faire
    - indice <- indice * 2 ou Boolean'pos(To_String(Texte_Binaire)(k)='1')
- Fin Pour
- Retourne Character'val(indice)

R0 : Ajouter tous les caractères de manière récursive dans le Texte (AJOUTER_CARACTERE_CODE)

Code : in UNBOUNDED_STRING
Texte : in out UNBOUNDED_STRING
Dict : in T_DICTIONNAIRE
Code_Fin : in UNBOUNDED_STRING

Lauriol François
Gras Yael

R1 : Comment “ajouter tous les caractères de manière récursive dans le Texte” ?

- Si Code/=Code_Fin et alors CODE_EST_PRESENT_DICT(Code, Dict) Alors Code : in UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE; Code_Fin : in UNBOUNDED_STRING
    - Texte <- Texte & LE_CARACTERE_DICT(Dict, Code) Code : in UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE; Texte : in out UNBOUNDED_STRING
- Sinon Si Code/=Code_Fin Alors
    - AJOUTER_CARACTERE_CODE(Code & “0”, Texte, Dict, Code_Fin) Code : in UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE; Texte : in out UNBOUNDED_STRING; Code_Fin : in UNBOUNDED_STRING
    - AJOUTER_CARACTERE_CODE(Code & “1”, Texte, Dict, Code_Fin) Code : in UNBOUNDED_STRING; Dict : in T_DICTIONNAIRE; Texte : in out UNBOUNDED_STRING; Code_Fin : in UNBOUNDED_STRING
- Fin Si

R0 : Afficher l’arbre de Huffman comme la figure 20 du polycopié de projet de manière récursive à partir du code Code (AFFICHER_ARBRE_RECURSIF)

Code : in UNBOUNDED_STRING
Arbre_Huff : in ARBRE

R1 : Comment “afficher l’arbre de Huffman comme la figure 20 du polycopié de projet de manière récursive à partir du code Code” ?

- Si Arbre_Huff^.Fils_G=null Alors Arbre_Huff : in ARBRE
    - Afficher une feuille Code : in UNBOUNDED_STRING ; Arbre_Huff : in ARBRE
- Sinon
    - Afficher une racine Code : in UNBOUNDED_STRING ; Arbre_Huff : in ARBRE
- Fin Si
- Nouvelle Ligne

R2 : Comment “afficher une feuille” ?

- Afficher l’espace à gauche
- Si Arbre_Huff^.Frequence=0 Alors
    - Écrire(“\--0--(0) ‘\$’”)
- Sinon si Caractère’pos(Arbre_Huff^.Caractere)=10 Alors
    - Écrire(“\--” & To_String(Code)(length(Code)) & “--(“ & Entier’Image(Arbre_Huff^.Frequence) & “) ‘\n‘“)
- Sinon
    - Écrire(“\--” & To_String(Code)(length(Code)) & “--(“ & Entier’Image(Arbre_Huff^.Frequence) & “) ‘” & Arbre_Huff^.Caractere & “‘“)

- Fin Si

R2 : Comment “afficher une racine” ?
- Afficher l’espace à gauche
- Écrire(“\--” & To_String(Code)(length(Code)) & “--(“ & Entier’Image(Arbre_Huff^.Frequence) & “)”)
- Nouvelle ligne
- AFFICHER_ARBRE_RECURSIF(Code & “0”, Arbre_Huff^.Fils_G)
- AFFICHER_ARBRE_RECURSIF(Code & “1”, Arbre_Huff^.Fils_D)

R3 : Comment “afficher l’espace à gauche” ?
- Pour i allant de 1 à length(Code)-1
  - Afficher l’espace selon le bit correspondant
- Fin Pour

R4 : Comment “afficher l’espace selon le bit correspondant” ?
- Si To_String(Code)(i)=’0’ Alors
  - Écrire(“| ”)
- Sinon
  - Écrire(“ ”)
- Fin Si

R0 : Renvoyer d’un caractère son octet (unbounded_string de 8 bits) correspondant.
(CONVERTIR_CARACTERE_BINAIRE)

Caractere : in Caractère

R1 : Comment “renvoyer d’un octet (unbounded_string de 8 bits) son caractère correspondant” ?
- indice <- Character’pos(Caractere) indice : out T_Octet ; Caractere : in Caractère
- Texte_Binaire <- To_Unbounded_String(“”)
- Pour k allant de 1 à 8 Faire
  - Ajouter le k -ième bit
  - indice <- indice/2
- Fin Pour
- Retourne Texte_Binaire

Lauriol François
Gras Yael

R2 : Comment “ajouter le k -ième bit” ?

- Si indice-2*(indice/2)=1 Alors
    - Texte_Binaire <- “1” & Texte_Binaire
- Sinon
    - Texte_Binaire <- “0” & Texte_Binaire
- Fin Si

Lauriol François
Gras Yael

SOUS-PROGRAMMES

—-- nom : LIRE
---- sémantique : Enregistrer le texte contenu dans le Fichier (.txt ou .hff) dans le Texte
---- paramètres : Fichier : in File_Type
---- Texte : out UNBOUNDED_STRING
---- pré-condition : Fichier est du type .txt ou .hff
procedure LIRE(Fichier : in File_Type; Texte : out UNBOUNDED_STRING)

—-- nom : ECRIRE_COMPRESSION
---- sémantique : Enregistrer le texte hff qui en binaire en caractere ASCII dans le Fichier hff
---- paramètres : Fichier : out File_Type
---- Texte : in out UNBOUNDED_STRING
---- pré-condition : Fichier est du type .hff
procedure ECRIRE_COMPRESSION(Fichier : out File_Type; Texte : in
UNBOUNDED_STRING)

—-- nom : ECRIRE_DECOMPRESSION
---- sémantique : Enregistrer le texte txt dans le Fichier txt
---- paramètres : Fichier : in out File_Type
---- Caractere : in Caractère
---- pré-condition : Fichier est du type .txt
procedure ECRIRE_DECOMPRESSION(Fichier : out File_Type; Caractere : in Caractère)

Lauriol François
Gras Yael

Groupe MN - 1

RAFFINAGES DES SOUS-PROGRAMMES

R0 : Enregistrer le texte contenu dans le Fichier dans le Texte (LIRE)

Fichier : in File_Type
Texte : out UNBOUNDED_STRING

R1 : Comment “enregistrer le texte contenu dans le Fichier dans le Texte” ?

- Open(Fichier, In_File, “.txt”) — Fichier : in File_Type
- S <- Stream(Fichier) — Fichier : in File_Type; S : out Stream_Access
- Texte <- To_Unbounded_String(“”) — Texte : out Unbounded_String
- TantQue Non End_of_File(Fichier) Faire — Fichier : in File_Type
  - Texte <- Texte & Character’Input(S) — Texte : in out Unbounded_String; S : in Stream_Access
- Fin TQ
- Close(Fichier) — Fichier : in File_Type

R0 : Enregistrer le texte hff binaire en caractère ASCII dans le Fichier hff (ECRIRE_COMPRESSION)

Fichier : in out File_Type
Texte : in out UNBOUNDED_STRING

R1 : Comment “enregistrer le texte hff binaire en caractere ASCII dans le Fichier hff” ?

- S <- Stream(Fichier) — Fichier : out File_Type; S : out Stream_Access
- CONVERTIR_CARACTERES(Texte) — Texte : in out UNBOUNDED_STRING
- Pour k allant de 1 à length(Texte) — Texte : in UNBOUNDED_STRING
  - Character'Write(S,To_String(Texte)(k)) — Texte : in out UNBOUNDED_STRING; S : in out Stream_Access
- Fin Pour

R0 : Enregistrer le texte txt dans le Fichier txt (ECRIRE_DECOMPRESSION)

Fichier : in out File_Type
Caractere : in Caractère

R1 : Comment “enregistrer le texte txt dans le Fichier txt” ?

- S <- Stream(Fichier) Fichier : out File_Type; S : out Stream_Access
- Character'Write(S, Caractere) S : in out Stream_Access; Caractere : in Caractère

Lauriol François
Gras Yael

Groupe MN - 1

EVALUATION PAR LES ÉTUDIANTS

Raffinage Compression

		Evaluation (I/P/A/+)
Forme (D-21)	Respect de la syntaxe Ri : Comment "... une action complexe ..." ? des actions combinées avec des structures de contrôle Rj : ...	+
	Verbes à l'infinitif pour les actions complexes	+
	Noms ou équivalent pour expressions complexes	+
	Tous les Ri sont écrits contre la marge et espacés	+
	Les flots de données sont définis	+
	Une seule décision ou répétition par raffinage	A
	Pas trop d'actions dans un raffinage (moins de 5 ou 6)	A
	Bonne présentation des structures de contrôle	A
Fond (D21-D22)	Le vocabulaire est précis	A
	Le raffinage d'une action décrit complètement cette action	+
	Le raffinage d'une action ne décrit que cette action	+
	Les flots de données sont cohérents	+
	Pas de structure de contrôle déguisée	+
	Qualité des actions complexes	+

Lauriol François
Gras Yael

Groupe MN - 1

Raffinage Décompression

		Evaluation (I/P/A/+)
Forme (D-21)	Respect de la syntaxe Ri : Comment "... une action complexe ..." ? des actions combinées avec des structures de contrôle Rj : ...	+
	Verbes à l'infinitif pour les actions complexes	+
	Noms ou équivalent pour expressions complexes	+
	Tous les Ri sont écrits contre la marge et espacés	+
	Les flots de données sont définis	+
	Une seule décision ou répétition par raffinage	A
	Pas trop d'actions dans un raffinage (moins de 5 ou 6)	A
	Bonne présentation des structures de contrôle	A
Fond (D21-D22)	Le vocabulaire est précis	A
	Le raffinage d'une action décrit complètement cette action	+
	Le raffinage d'une action ne décrit que cette action	+
	Les flots de données sont cohérents	+
	Pas de structure de contrôle déguisée	+
	Qualité des actions complexes	+