Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique,
d'Hydraulique et des Télécommunications

Programmation Arbre n-aire Arbre quaternaire Application à la compression d'images

DOURNAC Fabien

Mastère Informatique 2004/2005

Table des matières

Introduction

Le projet proposé ici consiste à développer des paquetages génériques sur les arbres n-aires et quaternaires. Ces paquetages seront réutilisés pour implanter un autre paquetage permettant de faire de la compression et décompression d'images.

Chapitre 1

Spécifications du paquetage générique arbre n-aire

Un arbre n-aire est un arbre dont les noeuds peuvent avoir un nombre quelconque de fils.

1.1 Représentation

Un arbre n-aire est représenté par :

- Une valeur à la racine de l'arbre
- Un arbre père
- Un arbre fils
- Un arbre frère

Cette structure devra être implantée à l'aide de pointeurs.

1.2 Opérations

Les opérations que l'on doit faire sur un arbre n-aire se classent en 3 catégories : création, consultation et modification. Voici la liste des procédures à implanter.

1.2.1 Création

- An_Creer_Vide : crée un arbre n-aire vide.
- An_Creer_Feuille : crée un arbre n-aire avec une valeur, mais sans fils, ni frere, ni pere.

1.2.2 Consultation

- An_Vide : qui détecte si un arbre est vide ou non.
- An_Valeur : qui retourne la valeur rangée à la racine d'un arbre.
- An_Pere : qui retourne l'arbre père d'un arbre.
- An_Fils : qui retourne l'arbre nième fils d'un arbre.
- An_Frere : qui retourne l'arbre nième frère d'un arbre.
- An_Afficher : qui affiche le contenu complet de l'arbre.
- An_Rechercher : qui recherche une valeur dans un arbre et retourne l'arbre dont la valeur est racine si elle est trouvée, un arbre vide sinon.
- An_Nombres_Fils : qui retourne le nombre de fils au premier niveau d'un arbre.
- An_Est_Feuille : qui indique si un arbre est une feuille (pas de fils).
- An_Est_Racine : qui indique si un arbre est sans père.

1.2.3 Modification

- An_Changer_Valeur : qui change la valeur rangée à la racine d'un arbre.
- An_Inserer_Fils : qui insère un arbre sans frère en position de premier fils d'un arbre a. L'ancien fils de a devient le premier frère de l'arbre inséré.
- An_Inserer_Frere : qui insère un arbre sans frère en position de premier frère d'un arbre a.
- An_Supprimer_Fils : qui supprime le nième fils d'un arbre a.
- An_Supprimer_Frere : qui supprime le nième frère d'un arbre a.
- An_Changer : qui applique une fonction à chaque élément de l'arbre.

Chapitre 2

Conception du paquetage générique arbre n-aire

2.1 Structuration des données

Un noeud est représenté par un enregistrement contenant la valeur de la racine, un pointeur sur le premier fils, un pointeur sur le frère et un pointeur sur le père de l'arbre. Le type arb_nr est mis en private pour cacher à l'utilisateur du paquetage la structure de données choisie pour représenter l'arbre n-aire.

```
type noeud;
type arb_nr is access noeud;
type noeud is record
   val: T;
   premier_fils: arb_nr;
   frere: arb_nr;
   pere: arb_nr;
end record;
```

Nous avons choisi de passer en paramètres génériques le type de la valeur de la racine, la procédure qui permet d'afficher un élément et la fonction qui sera appliquée à chaque élément

```
generic
type T is private;
with procedure ecrire ( a: in T );
with function fonction_main ( a: in T) return T;
```

2.2 Raffinages

Nous allons décrire les procédures et les fonctions que nous avons conçus.

2.2.1 fonction An_Creer_Vide

```
function An_Creer_Vide return arb_nr is

begin

return null;

end An_Creer_Vide;
```

2.2.2 fonction An_Creer_Feuille

```
function An_Creer_Feuille ( valeur : in T ) return arb_nr is

a: arb_nr;

begin

a:=new noeud;
a.val:=valeur;
a.premier_fils:=null;
a.frere:=null;
a.pere:=null;
return a;
```

```
end An_Creer_Feuille;
```

2.2.3 fonction An_Vide

```
function An_Vide ( a: in arb_nr ) return boolean is

begin

return (a=null);

end An_Vide;
```

2.2.4 fonction An_Valeur

```
function An_Valeur ( a: in arb_nr ) return T is

begin

return (a.val);
exception
when constraint_error => raise arbre_vide;

end An_Valeur;
```

Nous levons l'exception arbre_vide s'i y a une constraint_error. Nous la laissons se propager jusqu'au programme de test où elle sera traitée.

2.2.5 fonction An_Pere

```
function An_Pere ( a: in arb_nr ) return arb_nr is

begin

if An_Vide(a) then
   raise arbre_vide;
  else if a.pere=null then
         raise parente_vide;
        else
         return a.pere;
      end if;
end if;

end An_pere;
```

Si l'arbre n'a pas de père, on lève l'exception parente_vide.

2.2.6 fonction An_Fils

```
function An_Fils ( a: in arb_nr; n: in integer ) return arb_nr is

arb_cour: arb_nr;

begin

if An_Vide(a) then
   raise arbre_vide;
  else
     arb_cour:=a.premier_fils;
     for i in 1..n-1 loop
       arb_cour:=arb_cour.frere;
     end loop;
     return arb_cour;
end if;

exception
     when constraint_error => raise parente_vide;
end An_Fils;
```

On lève l'exception parente_vide s'il y a une constraint error.

2.2.7 fonction An_Frere

```
function An_Frere ( a: in arb_nr; n: in integer ) return arb_nr is

arb_cour: arb_nr;

begin

if An_Vide(a) then
   raise arbre_vide;
  else
     arb_cour:=a;
     for i in 1..n loop
       arb_cour:=arb_cour.frere;
     end loop;
     return arb_cour;
end if;

exception
     when constraint_error => raise parente_vide;
end An_Frere;
```

2.2.8 procédure An_Afficher

Pour l'affichage de l'arbre, nous utilisons une procédure auxiliaire récursive qui permettra de parcourir l'arbre.

```
procedure An_Afficher( a: in arb_nr ) is

begin

if An_Vide(a) then
   raise arbre_vide;
else
   An_Afficher_Aux(a,"");
end if;


end An_Afficher;
```

La procédure auxiliaire est la suivante.

```
procedure An_Afficher_Aux( a: in arb_nr ; str_decal: in string ) is

str_inc: string(1..4);

begin

if a=null then
   null;
  else
   ecrire(a.val);
   new_line;

   if (not An_Vide(a.premier_fils)) then
      put(str_decal&"|");
      new_line;
      put(str_decal&"+-- ");
      if (not An_Vide(a.frere)) then
         str_inc:="|  ";
  else
         str_inc:="   ";
      end if;

      -- on affiche le fils
      An_Afficher_Aux(a.premier_fils,str_decal&str_inc);
```

```
   end if;

   if (not An_Vide(a.frere)) then
      put(str_decal);
      -- on affiche le frere
      An_Afficher_Aux(a.frere,str_decal);
   end if;

end if;

end An_Afficher_Aux;
```

Nous parcourons tout d'abord dans le sens des premiers fils avec un décalage dans l'affichage puis dans le sens des frères en gardant le même décalage pour tous les frères.

2.2.9 fonction An_Rechercher

Nous utilisons ici aussi une procédure auxiliaire qui permettra de parcourir l'arbre dans les deux sens de récursivité, fils et frères.

```
function An_Rechercher ( a: in arb_nr; e: in T ) return arb_nr is

 -- arbre courant
arb_cour :arb_nr;
-- booleen pour continuer ou arreter la recherche
arret: boolean;
begin

if An_Vide(a) then
   raise arbre_vide;
  -- cas terminal
  else if a.val=e then
    return a;
   -- cas général: on utilise la procédure auxiliaire An_Rechercher_Aux
   -- l'arbre à retourner est arb_cour
   else
         arret:=false;
    arb_cour:=An_Creer_Vide;
    An_Rechercher_Aux(a.premier_fils,e,arb_cour,arret);
    return arb_cour;
      end if;
end if;

end An_Rechercher;
```

La procédure auxiliaire est la suivante. Nous nous servons d'un booléen passé en données/résultats pour terminer les appels récursifs dans le cas où la valeur est trouvée. Si elle n'est pas trouvée, arb_cour sera un pointeur null.

```
procedure An_Rechercher_Aux ( arb :in arb_nr ; e: in T ; arb_cour: out arb_nr ;
 arret: in out boolean ) is

begin

 -- cas terminal
if arb=null then
   null;
  else if arb.val=e then
         arret:=true;
         arb_cour:=arb;
   -- on recherche d'abord dans la lignée des fils
        else if not arret then
               An_Rechercher_Aux(arb.premier_fils,e,arb_cour,arret);
             else null;
            end if;
            --on recherche ensuite dans la lignée des frères
            if not arret then
```

```
               An_Rechercher_Aux (arb.frere,e,arb_cour,arret);
              else null;
             end if;
      end if;
end if;

end An_Rechercher_Aux;
```

2.2.10 fonction An_Nombre_Fils

Nous comptons le nombre de fils de l'arbre rentré en paramètre.

```
function An_Nombre_Fils( a: in arb_nr ) return integer is

n:integer;
arb_cour:arb_nr;

begin

if An_Vide(a) then
   raise arbre_vide;
 else if a.premier_fils /= null then
        n:=1;
        arb_cour:=a.premier_fils;
        while arb_cour.frere /= null loop
             n:=n+1;
             arb_cour:=arb_cour.frere;
        end loop;
        return n;
       else
   return 0;
       end if;
end if;

end An_Nombre_Fils;
```

2.2.11 fonction An_Est_Feuille

```
function An_Est_Feuille( a: in arb_nr )return boolean is

begin

if a.premier_fils=null then
   return true;
  else return false;
end if;

exception
  when constraint_error => raise arbre_vide;

end An_Est_Feuille;
```

2.2.12 fonction An_Est_Racine

```
function An_Est_Racine( a: in arb_nr )return boolean is

begin

if a.pere=null then
   return true;
  else return false;
end if;

exception
```

```
  when constraint_error => raise arbre_vide;

end An_Est_Racine;
```

2.2.13 procédure An_Changer_Valeur

```
procedure An_Changer_Valeur ( a:in out arb_nr; e: in T) is

begin

a.val:=e;
exception
    when constraint_error => raise arbre_vide;

end An_Changer_Valeur;
```

2.2.14 procédure An_Inserer_Fils

```
procedure An_Inserer_Fils ( arb: in out arb_nr ; arb_i: in out arb_nr ) is

begin

if An_Vide(arb) then
   raise arbre_vide;
-- si l'arbre à insérer est vide, on ne fait rien
  else if arb_i=null then
         null;
   -- 2 cas possibles: l'arbre arb a un fils ou n'a pas de fils
        else if arb.premier_fils=null then
          arb.premier_fils:=arb_i;
        -- on insère arb_i en position de premier fils de arb et
   -- l'ancien fils devient le premier frere du fils inséré
         else
               arb_i.frere:=arb.premier_fils;
               arb_i.pere:=arb;
               arb.premier_fils:=arb_i;
            end if;
      end if;
end if;

end An_Inserer_Fils;
```

2.2.15 procédure An_Inserer_Frere

```
procedure An_Inserer_Frere( arb: in out arb_nr ; arb_i: in out arb_nr ) is

begin

if An_Vide(arb) then
   raise arbre_vide;
-- si l'arbre à insérer est vide, on ne fait rien
  else if arb_i=null then
         null;
        else
-- le premier frère de arb devient le frère de l'arbre à insérer arb_i
         arb_i.pere:=arb.pere;
    arb_i.frere:=arb.frere;
         arb.frere:=arb_i;
       end if;
end if;

end An_Inserer_Frere;
```

2.2.16 procédure An_Supprimer_Fils

```
procedure An_Supprimer_Fils ( a: in out arb_nr ; n: in integer ) is
```

```
-- arbre sauvegardés:
-- frère suivant le fils
a_s: arb_nr;
-- frère précédant le fils
a_pre: arb_nr;

begin

if An_Vide(a) then
   raise arbre_vide;
   -- si le n ieme fils n'existe pas, on ne fait rien
  else if n>An_nombre_fils(a) then
         null;
   -- si n=1, suppression du premier fils
        else if n=1 then
               a_s:=a.premier_fils;
          a_s.pere:=null;
               a.premier_fils:=a_s.frere;
     -- on sauvegarde le pointeur sur le n-1 ième fils
     -- et le pointeur sur le frere du fils à supprimer
          else --
               a_pre:=An_Fils(a,n-1);
               a_s:=a_pre.frere.frere;
               a_pre.frere.frere:=null;
          a_pre.frere.pere:=null;
               a_pre.frere:=a_s;
             end if;
      end if;
end if;

end An_Supprimer_Fils;
```

2.2.17 procédure An_Supprimer_Frere

```
procedure An_Supprimer_Frere ( a :in out arb_nr ; n: in integer) is

-- arbres sauvegardés: précedant et suivant le frère
a_pre:arb_nr;
a_s:arb_nr;
begin
if An_Vide(a) then
   raise arbre_vide;
  -- suppression du nième frère
  else
-- arbre précédant le frère à supprimer
-- si l'exeption parente_vide est levée
-- dans An_Frere, elle se propage
   a_pre:=An_Frere(a,n-1);
-- arbre suivant le frère à supprimer
-- si a_pre ou a_pre.frere vaut null, l'exception
-- constraint_error est levée et l'on ne fait rien
   a_s:=a_pre.frere.frere;
-- cas où a_pre ou a_pre.frere différent de null
   a_pre.frere.pere:=null;
   a_pre.frere.frere:=null;
   a_pre.frere:=a_s;
end if;

exception

 -- si a_pre ou a_pre.frere vaut null, on ne fait rien
 when constraint_error => null;

end An_Supprimer_Frere;
```

2.2.18 procédure An_Changer

la fonction fonction_main est passée en paramètre générique lors de l'instanciation du paquetage arbre_nr.

```
procedure An_Changer( a: in out arb_nr ) is

begin

if An_Vide(a) then
   null;
  else
   a.val:=fonction_main(a.val);
   An_Changer(a.premier_fils);
   An_Changer(a.frere);
end if;

end An_Changer;
```

Chapitre

3 Spécifications du paquetage générique arbre quaternaire

3.1 Représentation

Un arbre quaternaire est un arbre ayant 0 ou 4 fils. Les quatre fils sont appelés fils nord-ouest, nord-est, sud-ouest, sud-est.

3.2 Opérations

Les opérations suivantes sont les mêmes que celles demandées pour le paquetage arbre n-aire :

Aq_Creer_Vide, Aq_Creer_Feuille, Aq_Vide, Aq_Valeur, Aq_Pere, Aq_Afficher, Aq_Rechercher, Aq_Est_Feuille, Aq_Est_Racine, Aq_Changer_Valeur, Aq_Changer.

Les nouvelles fonctions à implanter sont :

- Aq_Construire : crée un arbre quaternaire à partir d'une valeur et de 4 arbres fils.
- Aq_Nord_Ouest : qui retourne le fils nord-ouest d'un arbre quaternaire.
- Aq_Nord_Est : qui retourne le fils nord-est d'un arbre quaternaire.
- Aq_Sud_Ouest : qui retourne le fils sud-ouest d'un arbre quaternaire.
- Aq_Sud_Est : qui retourne le fils sud-est d'un arbre quaternaire.

Chapitre 4

Conception du paquetage générique arbre quaternaire

4.1 Structuration des données

Les arbres quaternaires étant une sous-catégorie des arbres n-aires, nous définissons un type arbre quaternaire sous-type des arbres n-aires. Les paramètres génériques sont les mêmes que ceux du paquetage arbre n-aire.

4.2 Raffinages

Nous utilisons la directive ADA "renames" dans la partie spécification qui nous permet de réutiliser les procédures et fonctions implantés dans le paquetage arbre n-aire.
exemple :

```
function Aq_Creer_Vide return arb_quat renames An_Creer_Vide;
```

Nous faisons de même pour les exceptions :

```
-- arbre vide
arbreq_vide: exception renames arbre_vide ;
-- pas de parenté
parenteq_vide: exception renames parente_vide;
```

Nous définissons une exception filsq_vide pour la procédure Aq_Construire.

4.2.1 procédure Aq_Construire

```
-----------------------------------------------------------------
---- procédure Aq_Construire: crée un arbre 4-aire à partir d'une
---- valeur et de 4 arbres fils
---- paramètres: arb_res, l'arbre construit
--             val, la valeur du père
----           a_no, a_ne, a_so, a_se, respectivement les fils
----           nord ouest, nord est, sud ouest, sud est.
---- post-conditions: si un des fils est vide, on leve l'exception
---- filsq_vide
-----------------------------------------------------------------

procedure Aq_Construire( valeur: in T; a_no, a_ne, a_so, a_se: in out arb_quat;
arb_res: out arb_quat ) is

begin

-- Si un des fils est vide, on leve l'exception filsq_vide
if (Aq_Vide(a_no) or Aq_Vide(a_ne) or Aq_Vide(a_so) or Aq_Vide(a_se)) then
   raise filsq_vide;
-- sinon on construit l'arbre 4-aire
  else
   arb_res:=Aq_Creer_Feuille(valeur);
   An_Inserer_Fils(arb_res,a_se);
   An_Inserer_Fils(arb_res,a_so);
   An_Inserer_Fils(arb_res,a_ne);
   An_Inserer_Fils(arb_res,a_no);
```

```
end if;

end Aq_Construire;
```

4.2.2 fonction Aq_Nord_Ouest

Nous nous servons de la fonction An_Fils qui retourne l'arbre nième fils d'un arbre.

```
function Aq_Nord_Ouest( a: in arb_quat ) return arb_quat is

begin

return An_fils(a,1);

end Aq_Nord_Ouest;
```

Nous faisons de même pour Aq_Nord_Est (return An_fils(a,2)), Aq_Sud_Ouest (return An_fils(a,3)) et Aq_Sud_Est(return An_fils(a,4)).

Chapitre 5

Spécifications du paquetage pimage avec compression

Le but est de compresser et décompresser une image en utilisant un arbre quaternaire. Les images que nous utilisons sont carrées et leur dimension est une puissance de 2. Une couleur est définie à partir des 3 couleurs de base rouge, vert, bleu et est codée sur un entier.

Le principe d'encodage récursif d'une image de dimension n est le suivant : si l'image est homogène, on crée une feuille dont la valeur est la couleur de l'image. Si l'image n'est pas homogène, on la découpe en 4 sous-images de dimension n/2, et on crée un arbre quaternaire dont les quatre fils sont les encodages des quatre sous-images. En parcourant en profondeur l'arbre quaternaire, nous obtenons une suite de valeurs représentant l'image compressée.
Il est demandé de réaliser un paquetage permettant de :

— créer ue image à partir d'une dimension et d'une matrice.
— charger une image à partir d'un fichier.
— enregistrer une image dans un fichier.
— afficher une image.
— charger une image encodée depuis un fichier.
— enregistrer une image encodée dans un fichier.
— appliquer une opération sur une image

Chapitre 6

Conception du paquetage pimage

6.1 Structuration des données

Nous choisissons de représenter une couleur par une valeur codée sur 32 bits (pixel). L'image est une matrice de pixels. Nous utilisons des tableaux non contraints.

```
-- sous type arb_im
subtype arb_im is arb_quat;
-- définition de la couleur primaire
subtype couleur_prim is integer range 0..255;
-- définition du pixel: valeur codée sur 32 bits
type pixel is mod 2**32;
-- tableau contenant l'image
type image is array (integer range <>,integer range <>) of pixel;
```

6.2 Raffinages pour la compression

6.2.1 procédure im_to_arb

Cette procédure permet de contruire un arbre à partir d'une image. Nous suivons l'algorithme d'encodage.

```
-------------------------------------------------------------------
---- procedure im_to_arb: crée un arbre à partir d'une image
---- paramètres: im l'image, n sa dimension, a l'arbre à créer,
---- tol la tolérance utilisé par la fonction est_homogene
-------------------------------------------------------------------

procedure im_to_arb( im: in image; n: in integer; a: out arb_im; tol: in intensite) is

im_no,im_ne,im_so,im_se: image(1..n/2,1..n/2);
a_no,a_ne,a_so,a_se: arb_im;

begin
-- cas où la dimension vaut 1
if n=1 then
   a:=Aq_Creer_Feuille(integer(im(1,1)));
  -- si elle est homogène, on crée une feuille avec la couleur moyenne
  else if est_homogene(im,n,tol) then
         a:=Aq_Creer_Feuille(integer(coul_moy(im,n)));
        else -- on construit l'arbre en divisant les dimensions par deux
          for i in 1..n/2 loop
        for j in 1..n/2 loop
           im_no(i,j):=im(i,j);
      im_ne(i,j):=im(i,j+n/2);
      im_so(i,j):=im(i+n/2,j);
      im_se(i,j):=im(i+n/2,j+n/2);
        end loop;
    end loop;

         im_to_arb(im_no,n/2,a_no,tol);
    im_to_arb(im_ne,n/2,a_ne,tol);
    im_to_arb(im_so,n/2,a_so,tol);
```

```
     im_to_arb(im_se,n/2,a_se,tol);
     Aq_Construire(-1,a_no,a_ne,a_so,a_se,a);

       end if;
end if;

end im_to_arb;
```

Il nous reste à parcourir l'arbre et enregistrer la séquence dans un fichier, ce qui est fait dans la procédure enregistrer_imagec et arb_to_imc.

6.2.2 fonction est_homogene

Nous détaillons ici la fonction est_homogene en distinguant 3 cas : la tolérance est minimale (0), maximale(100) ou entre les deux. Si elle est maximale, alors nous retournons le booléen égal à true et une feuille est créée avec la couleur moyenne de la zone concernée (cf procédure im_to_arb). Si elle est minimale, on s'assure que tous les pixels ont la même valeur. Enfin, pour le cas intermédiaire, nous regardons si l'inégalité abs(couleur_primaire-moyenne) <= tolerance est vérifiée pour chaque pixel et couleur primaire.

```
-------------------------------------------------------------------------
---- fonction est_homogene: fonctionne qui détermine si la zone de
---- taille n est homogène. retourne un booléen
---- paramètres: im l'image
----           taille n de la zone
----           tol la tolérance pour l'homogénéité
-------------------------------------------------------------------------

function est_homogene( im: in image; n: in integer; tol: in intensite ) return boolean is

result: boolean;
val,i,j: integer;
c_moy: integer;

begin
result:=true;

case tol is
     -- tolérance maximale autorisée, l'image est considérée comme homogène
     when intensite'last => return true;
     -- tolérance minimale, on compare le premier point avec tous les
     -- autres
     when intensite'first => val:=integer(im(1,1));
                             i:=1;
                             while i<=n and result loop
                j:=1;
                while j<=n and result loop
                result:=(im(i,j)=pixel(val));
           j:=j+1;
           end loop;
                i:=i+1;
           end loop;
           return result;
      -- cas général: on calcule la couleur moyenne et on extrait les 3
      -- couleurs primaires moyennes que l'on va comparer aux pixels
      -- de l'image avec la condition que abs(couleur_primaire-moyenne) <=
      -- tolerance
      when others => c_moy:=integer(coul_moy(im,n));
                     i:=1;
         while i<=n and result loop
                   j:=1;
         while j<=n and result loop
result:=(abs(get_bleu(im(i,j))-get_bleu(pixel(c_moy)))<=tol) and (abs(get_vert(im(i,j))-get_vert(
    pixel(c_moy)))<=tol) and (abs(get_rouge(im(i,j))-get_rouge(pixel(c_moy)))<=tol);
         j:=j+1;
         end loop;
         i:=i+1;
         end loop;
                     return result;
```

```
end case;

end est_homogene;
```

6.2.3 procédure enregistrer_imagec

```
-----------------------------------------------------------------------
---- procedure enregistrer_imagec: enregistre une image compressé dans
---- un fichier à partir de l'arbre.
---- paramètres: a l'arbre construit, n la dimension de l'image
-----------------------------------------------------------------------

procedure enregistrer_imagec( a: in arb_im ; n: in pixel ) is

filename: string(1..4);
f: pack_pixel.file_type;

begin

put("Entrer un nom de fichier (4 caractères): ");
get(filename);
skip_line;
create(f,out_file,filename);
write(f,n);
arb_to_imc(a,f);
close(f);

end enregistrer_imagec;
```

Nous appelons la procédure récursive arb_to_imc qui va parcourir l'arbre.

6.2.4 procédure arb_to_imc

```
-----------------------------------------------------------------
----- procedure arb_to_imc: permet de passer de l'arbre à la séquence
----- compressée.
----- paramètres: a l'arbre construit, f le nom logique du fichier
-----------------------------------------------------------------

procedure arb_to_imc( a: in arb_im; f: in pack_pixel.file_type ) is

begin

if Aq_Vide(a) then
   null;
  else
   if Aq_Valeur(a)=-1 then
      write(f,2**24);
     else
      write(f,pixel(Aq_Valeur(a)));
   end if;
   arb_to_imc(Aq_Fils(a,1),f);
   arb_to_imc(Aq_Frere(a,1),f);
end if;

end arb_to_imc;
```

6.3 Raffinages pour la décompression

Il s'agit tout d'abord de construire l'arbre à partir de la séquence encodée (procédure charger_im_encod et imc_to_arb) puis de reconstruire l'image à partir de cet arbre (procédure arb_to_im).

6.3.1 procédure charger_im_encod

```
procedure charger_im_encod( a: out arb_im ; n: out pixel ) is

filename: string(1..4);
f: pack_pixel.file_type;
-- dimension de l'image originale
dim:pixel;

begin

put("Entrer un nom de fichier (4 caractères): ");
get(filename);
skip_line;
open(f,in_file,filename);
-- on lit la dimension
read(f,dim);
n:=dim;
a:=imc_to_arb(f);
close(f);

end charger_im_encod;
```

Nous utilisons la fonction auxiliaire imc_to_arb.

6.3.2 fonction imc_to_arb

cette fonction permet de construire l'arbre à partir de la séquence de l'image compressée.

```
function imc_to_arb( f: in pack_pixel.file_type ) return arb_im is

-- fils de l'arbre quaternaire
a,a_no,a_ne,a_so,a_se: arb_im;
v: pixel;

begin

read(f,v);
-- si la valeur vaut -1 (codé par 2**24), on crée une feuille
if v /= 2**24 then
   a:=Aq_Creer_Feuille(integer(v));
  else
   a_no:=imc_to_arb(f);
   a_ne:=imc_to_arb(f);
   a_so:=imc_to_arb(f);
   a_se:=imc_to_arb(f);
   Aq_Construire(-1,a_no,a_ne,a_so,a_se,a);
end if;
return a;

end imc_to_arb;
```

6.3.3 procédure arb_to_im

Nous devons reconstituer l'image à partir de l'arbre. l'arbre est arb_im et l'image résultante est im. Nous nous servons des indices i_d,i_f,j_d,j_f pour obtenir le décalage entre les appels récursifs.

```
-----------------------------------------------------------------------
---- procedure arb_to_im: permet de reconstituer l'image originale
---- à partir de l'image compressée.
---- paramètres: a l'arbre reconstitué, im l'image à construire,
---- i_d,i_f,j_d,j_f les indices de début et de fin, n la dimension
---- de l'image originale
-----------------------------------------------------------------------

procedure arb_to_im ( a: in arb_im ; im: out image ; i_d,i_f,j_d,j_f: in integer ) is

begin
```

```
-- cas terminal
if Aq_Vide(a) then
   null;-- cas général
  else if Aq_Valeur(a) /= -1 then
         for i in i_d..i_f loop
            for j in j_d..j_f loop
          im(i,j):=pixel(Aq_Valeur(a));
       end loop;
    end loop;

     else -- on parcourt tous les fils d'une cellule
          arb_to_im(Aq_Nord_Ouest(a),im,i_d,i_f-(i_f-i_d+1)/2,j_d,j_f-(j_f-j_d+1)/2);
          arb_to_im(Aq_Nord_Est(a),im,i_d,i_f-(i_f-i_d+1)/2,j_f-(j_f-j_d+1)/2+1,j_f);
          arb_to_im(Aq_Sud_Ouest(a),im,i_f-(i_f-i_d+1)/2+1,i_f,j_d,j_f-(j_f-j_d+1)/2);
          arb_to_im(Aq_Sud_Est(a),im,i_f-(i_f-i_d+1)/2+1,i_f,j_f-(j_f-j_d+1)/2+1,j_f);
    end if;

end if;

end arb_to_im;
```

6.3.4 procédure thresh_im

Nous devons faire une opération sur l'image. Nous avons choisi de faire un seuillage binaire.

```
-------------------------------------------------------
----- procedure thresh_im: fait un seuillage sur l'image
----- paramètres: im l'image à traiter, n sa dimension,
----- le seuil
-------------------------------------------------------

procedure thresh_im( im: in out image ; n: in integer ; seuil: in integer ) is

begin

for i in 1..n loop
    for j in 1..n loop
        if (integer(im(i,j))<= seuil) then
     im(i,j):=0;
    else
      im(i,j):=1;
  end if;
  end loop;
end loop;

end thresh_im;
```

Chapitre

7 Validation des paquetages

7.1 paquetage arbre n-aire et quaternaire

Les programmes de test main_arb_nr.adb et main_arb_quater.adb utilisent respectivement les paquetages arbre_nr.db et arbre_quater.adb. Il est possible de construire un arbre pour vérifier le bon fonctionnement des différentes procédures et fonctions.

7.2 paquetage pimage

Le programme de test main_pimage.adb permet de créer une image pour valider la compression et décompression. L'affichage des couleurs de cette image est représenté sur la figure suivante :

Figure 7.1 – *Contenu de l'image test 8×8*

Sa taille est égale à 260 octets (64×4+4) car nous stockons aussi la dimension. On vérifie maintenant la compression. Nous prenons en compte une tolérance pour déterminer si une image est homogène. L'arbre construit avec une tolérance minimale (0) est représenté sur la figure suivante :

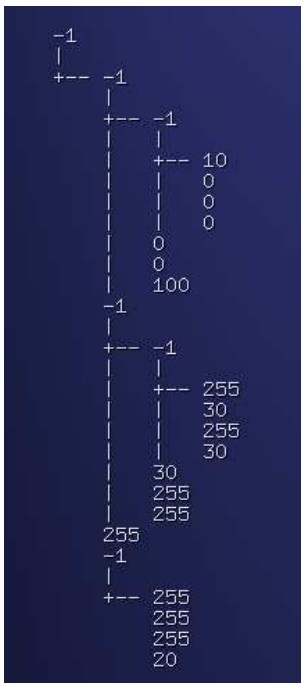

Figure 7.2 – *Affichage de l'arbre obtenu pour la compression avec une tolérance minimale*

La taille de l'image compressée est égale à 104 octets (25×4+4) dont 4 octets pour la dimension, ce qui est bien le résultat attendu. Pour une tolérance maximale (100), nous obtenons une feuille avec la couleur moyenne

de l'image (161) car celle-ci est considérée comme homogène d'après le critère dans la fonction est_homogene.

Avec une tolérance égale à 80, nous obtenons l'arbre suivant :

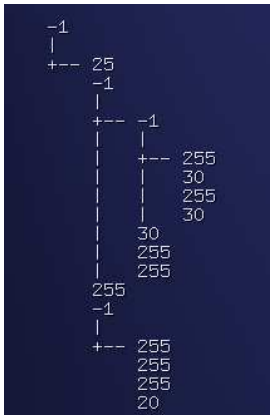

FIGURE 7.3 – *Affichage de l'arbre obtenu pour la compression avec une tolérance de 80*

Nous remarquons qu'il y aura une perte d'information par rapport à l'image test dans la décompression car la partie Nord Ouest de l'image est considérée comme homogène avec cette tolérance. Ceci est illustré sur la figure suivante :

25	25	25	25	255	30	30	30
25	25	25	25	255	30	30	30
25	25	25	25	255	255	255	255
25	25	25	25	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	20	20
255	255	255	255	255	255	20	20

FIGURE 7.4 – *Contenu de l'image 8×8 décompréssée avec une tolérance de 80*

Nous enregistrons pour chaque cas l'image compréssée puis vérifions la décompression.

Conclusion

Ce projet nous a permis de créer un paquetage permettant de faire de la compression et décompression d'images. Les opérations implantées pour les arbres quaternaires ont été réutilisées et validées dans l'algorithme d'encodage.

Les sources de ce projet sont téléchargeables ici :