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

Programming N-Ary trees
Quaternary trees
Applications to Image Compression

DOURNAC Fabien

Mastère Informatique 2004/2005

Contents

1 Specifications of N-Ary generic package **5**
1.1 Representation 5
1.2 Operations 5
1.2.1 Creation 5
1.2.2 Consultation 5
1.2.3 Modification 6

2 Conception of N-Ary generic package **7**
2.1 Structuration of data 7
2.2 Algorithmic refining 7
2.2.1 function Nt_Create_Empty 7
2.2.2 function Nt_Create_Leaf 7
2.2.3 function Nt_Empty 8
2.2.4 function Nt_Value 8
2.2.5 function Nt_Father 8
2.2.6 function Nt_Child 8
2.2.7 function Nt_Brother 9
2.2.8 procedure Nt_Display 9
2.2.9 function Nt_Search 10
2.2.10 function Nt_Number_Children 11
2.2.11 function Nt_Is_Leaf 11
2.2.12 function Nt_Is_Root 11
2.2.13 procedure Nt_Change_Value 11
2.2.14 procedure Nt_Insert_Child 12
2.2.15 procedure Nt_Insert_Brother 12
2.2.16 procedure Nt_Delete_Child 12
2.2.17 procedure Nt_Delete_Brother 13
2.2.18 procedure Nt_Change 13

3 Specifications of Quaternary generic package **15**
3.1 Representation 15
3.2 Operations 15

4 Conception of Quaternay tree generic package **16**
4.1 Structuration of data 16
4.2 Algorithmic refining 16
4.2.1 procedure Qt_Build 16
4.2.2 function Qt_North_West 17

5 Specifications of pimage package with compression **18**

6 Conception of pimage package **19**
6.1 Structuration of data 19
6.2 Algorithmic refining 19
6.2.1 procedure im_to_tree 19
6.2.2 function is_homogeneous 20
6.2.3 procedure save_imagec 21
6.2.4 procedure tree_to_imc 21
6.3 Algorithmic refining for decompression 21
6.3.1 procedure load_im_encod 21
6.3.2 function imc_to_tree 22
6.3.3 procedure tree_to_im 22

6.3.4 procedure thresh_im . . . 23

7 Validation of packages **24**
7.1 n-ary and quaternary packages . . . 24
7.2 pimage package . . . 24

Introduction

The aim of this project is to develop generic packages on n-ary and quaternary trees. These packages will be reused to implement another package allowing to achieve image compression and decompression.

Chapter 1

Specifications of N-Ary generic package

A n-ary tree is a rooted tree in which each node has no more than n children.

1.1 Representation

it can be represented by :

- A value at the root of tree
- A father tree
- A children tree
- A brother tree

This structure will be implemented thanks to pointers.

1.2 Operations

Operations that one has to operate on n-ary tree are grouped into 3 categories : creation, consultation and modification. Here are the list of procedures to code.

1.2.1 Creation

- Nt_Create_Empty: create an empty n-ary tree.
- Nt_Create_Leaf: create a n-ary tree with a value, without brother or father.

1.2.2 Consultation

- Nt_Empty: check if tree is empty or not.
- Nt_Value: return value at the root of tree.
- Nt_Father: return father tree of tree.
- Nt_Child: return the n-th children tree of tree.
- Nt_Brother: return the n-th brother tree of tree.
- Nt_Display: display all content of tree.
- Nt_Search: search for a value into tree and return tree whose root value is found. If nothing is found, return an empty tree.
- Nt_Number_Children: return the number of chidren tree at first level.
- Nt_Is_Leaf: check if tree is a leaf (no child).
- Nt_Is_Root: check if tree has no father.

1.2.3 Modification

- Nt_Change_Value: change value at the root of tree.
- Nt_Insert_Child: insert a tree without brother at position of first child. Old child becomes the first brother of inserted tree.
- Nt_Insert_Brother: insert a tree without brother at position of first brother.
- Nt_Delete_Child: delete the n-th child of tree.
- Nt_Delete_Brother: delete the n-th brother of tree.
- Nt_Change: apply a function to each element of tree.

Chapter 2

Conception of N-Ary generic package

2.1 Structuration of data

A node is represented by a record type containing the value of the root, a pointer on the first child, a pointer on the brother and another one on the father tree. The tree_nr type is set to private for hiding from user package the data structure chosen to represent the n-ary tree.

```
type node;
type tree_nr is access node;
type node is record
   val: T;
   first_child: tree_nr;
   brother: tree_nr;
   father: tree_nr;
end record;
```

We chose to pass as generic parameters the type of value of the root, the procedure which displays element and the function that will be applied to each element.

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

2.2 Algorithmic refining

We describe now procedures and functions that we have coded.

2.2.1 function Nt_Create_Empty

```
function Nt_Create_Empty return tree_nr is

begin

return null;

end Nt_Create_Empty;
```

2.2.2 function Nt_Create_Leaf

```
function Nt_Create_Leaf ( value : in T ) return tree_nr is

a: tree_nr;

begin

a:=new node;
a.val:=value;
a.first_child:=null;
a.brother:=null;
a.father:=null;
return a;
```

```
end Nt_Create_Leaf;
```

2.2.3 function Nt_Empty

```
function Nt_Empty ( a: in tree_nr ) return boolean is

begin

return (a=null);

end Nt_Empty;
```

2.2.4 function Nt_Value

```
function Nt_Value ( a: in tree_nr ) return T is

begin

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

end Nt_Value;
```

We raise tree_empty exception if there is a constraint_error. We let it propagate up to test program where it is processed.

2.2.5 function Nt_Father

```
function Nt_Father ( a: in tree_nr ) return tree_nr is

begin

if Nt_Empty(a) then
   raise tree_empty;
  else if a.father=null then
         raise relation_empty;
        else
         return a.father;
      end if;
end if;

end Nt_Father;
```

If tree has no father, we raise relation_empty exception.

2.2.6 function Nt_Child

```
function Nt_Child ( a: in tree_nr; n: in integer ) return tree_nr is

tree_curr: tree_nr;

begin

if Nt_Empty(a) then
   raise tree_empty;
  else
     tree_curr:=a.first_child;
     for i in 1..n-1 loop
       tree_curr:=tree_curr.brother;
     end loop;
     return tree_curr;
end if;

exception
     when constraint_error => raise relation_empty;
end Nt_Child;
```

We raise relation_empty exception if there is a constraint error.

2.2.7 function Nt_Brother

```
function Nt_Brother ( a: in tree_nr; n: in integer ) return tree_nr is

tree_curr: tree_nr;

begin

if Nt_Empty(a) then
   raise tree_empty;
  else
     tree_curr:=a;
     for i in 1..n loop
       tree_curr:=tree_curr.brother;
     end loop;
     return tree_curr;
end if;

exception
     when constraint_error => raise relation_empty;
end Nt_Brother;
```

2.2.8 procedure Nt_Display

For displaying the tree, we use a recursive auxiliary procedure which will allow to browse tree.

```
procedure Nt_Display( a: in tree_nr ) is

begin

if Nt_Empty(a) then
   raise tree_empty;
else
   put("   ");
   Nt_Display_Aux(a,"  ");
end if;

end Nt_Display;
```

Auxiliay procedure is as follow:

```
procedure Nt_Display_Aux( a: in tree_nr ; str_shift: in string ) is

str_inc: string(1..4);

begin

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

   if (not Nt_Empty(a.first_child)) then
      put(str_shift&"|");
      new_line;
      put(str_shift&"+-- ");
      if (not Nt_Empty(a.brother)) then
         str_inc:="|  ";
  else
   str_inc:="   ";
      end if;

      -- we display child
      Nt_Display_Aux(a.first_child,str_shift&str_inc);
   end if;
```

```
   if (not Nt_Empty(a.brother)) then
      put(str_shift);
      -- we display brothers
      Nt_Display_Aux(a.brother,str_shift);
   end if;

end if;

end Nt_Display_Aux;
```

We browse firstly along the first children with a shift on display and along the brothers keeping the same shift for all brothers.

2.2.9 function Nt_Search

We use also here an auxiliary procedure that make browse the tree along two ways of recursivity, the children one and the brothers one.

```
function Nt_Search ( a: in tree_nr; e: in T ) return tree_nr is

-- current tree
tree_curr :tree_nr;
-- boolean to keep on or not searching
stop: boolean;
begin

if Nt_Empty(a) then
   raise tree_empty;
  -- terminal case
  else if a.val=e then
    return a;
   -- general case: we use Nt_Search_Aux auxiliary procedure
   -- tree_curr is returned
   else
         stop:=false;
    tree_curr:=Nt_Create_Empty;
    Nt_Search_Aux(a.first_child,e,tree_curr,stop);
    return tree_curr;
       end if;
end if;

end Nt_Search;
```

The auxiliary procedure is as follow : we use a boolean, passed as data/results parameters, to finish the recursive calls if value is found. If it is found, tree_curr is set to a null pointer.

```
procedure Nt_Search_Aux ( tree :in tree_nr ; e: in T ; tree_curr: out tree_nr ; stop: in out boolean
     ) is

begin

-- terminal case
if tree=null then
   null;
  else if tree.val=e then
         stop:=true;
         tree_curr:=tree;
   -- we search firstly on child relation
        else if not stop then
                Nt_Search_Aux(tree.first_child,e,tree_curr,stop);
               else null;
             end if;
             -- then we search on brothers relation
             if not stop then
                Nt_Search_Aux (tree.brother,e,tree_curr,stop);
               else null;
             end if;
       end if;
end if;
```

```
end Nt_Search_Aux;
```

2.2.10 function Nt_Number_Children

We count the number of children of a tree passed as parameter.

```
function Nt_Number_Children( a: in tree_nr ) return integer is

n:integer;
tree_curr:tree_nr;

begin

if nt_Empty(a) then
   raise tree_empty;
 else if a.first_child /= null then
        n:=1;
        tree_curr:=a.first_child;
        while tree_curr.brother /= null loop
              n:=n+1;
              tree_curr:=tree_curr.brother;
        end loop;
        return n;
       else
   return 1;
       end if;
end if;

end Nt_Number_Children;
```

2.2.11 function Nt_Is_Leaf

```
function Nt_Is_Leaf( a: in tree_nr )return boolean is

begin

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

exception
  when constraint_error => raise tree_empty;

end Nt_Is_Leaf;
```

2.2.12 function Nt_Is_Root

```
function Nt_Is_Root( a: in tree_nr )return boolean is

begin

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

exception
  when constraint_error => raise tree_empty;

end Nt_Is_Root;
```

2.2.13 procedure Nt_Change_Value

```
procedure Nt_Change_Value( a:in out tree_nr; e: in T) is

begin

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

end Nt_Change_Value;
```

2.2.14 procedure Nt_Insert_Child

```
procedure Nt_Insert_Child ( tree: in out tree_nr ; tree_i: in out tree_nr ) is

begin

if Nt_Empty(tree) then
   raise tree_empty;
-- if tree to insert is empty, we do noting
  else if tree_i=null then
         null;
   -- 2 possible cases: tree has a child or not
        else if tree.first_child=null then
          tree.first_child:=tree_i;
   -- we insert tree_i at first child position and old child
   -- become the first brother
         else
                tree_i.brother:=tree.first_child;
                tree_i.father:=tree;
                tree.first_child:=tree_i;
             end if;
      end if;
end if;

end Nt_Insert_Child;
```

2.2.15 procedure Nt_Insert_Brother

```
procedure Nt_Insert_Brother( tree: in out tree_nr ; tree_i: in out tree_nr ) is

begin

if Nt_Empty(tree) then
   raise tree_empty;
  else if tree_i=null then
         null;
        else
         tree_i.father:=tree.father;
    tree_i.brother:=tree.brother;
         tree.brother:=tree_i;
      end if;
end if;

end Nt_Insert_Brother;
```

2.2.16 procedure Nt_Delete_Child

```
procedure Nt_Delete_Child ( a: in out tree_nr ; n: in integer ) is

-- saved trees: previous and following the child
a_prev: tree_nr;
a_next: tree_nr;

begin

if Nt_Empty(a) then
   raise tree_empty;
   -- if the n-th child does not exist, we do nothing
```

```
  else if n>Nt_Number_Children(a) then
          null;
   -- if n=1, deleting the first child
         else if n=1 then
                 a_next:=a.first_child;
           a_next.father:=null;
                 a.first_child:=a_next.brother;
     -- we save pointer on n-th child
     -- ans pointer on child's brother to delete
          else
                 a_prev:=Nt_Child(a,n-1);
                 a_next:=a_prev.brother.brother;
                 a_prev.brother.brother:=null;
           a_prev.brother.father:=null;
                 a_prev.brother:=a_next;
              end if;
      end if;
end if;

end Nt_Delete_Child;
```

2.2.17 procedure Nt_Delete_Brother

```
procedure Nt_Delete_Brother ( a :in out tree_nr ; n: in integer) is

-- saved trees: previous and following the child
a_prev: tree_nr;
a_next: tree_nr;

begin
if Nt_Empty(a) then
   raise tree_empty;
  -- deleting the n-th brother
  else
  -- previous tree before brother to delete
  -- if relation_empty is raised into Nt_Brother,
  -- it propagates
  a_prev:=Nt_Brother(a,n-1);
  -- next tree after brother to delete
  -- if a_prev or a_prev.brother is raised,
  -- constraint_error is raised, then it propagates
  a_next:=a_prev.brother.brother;
  -- case where a_prev or a_prev.brother is not null
  a_prev.brother.father:=null;
  a_prev.brother.brother:=null;
  a_prev.brother:=a_next;
end if;

exception

 -- if a_prev or a_prev.brother is null, we do noting
 when constraint_error => null;

end Nt_Delete_Brother;
```

2.2.18 procedure Nt_Change

The main function "main_function" is passed as generic parameter when tree_nary package is instancied.

```
procedure Nt_Change( a: in out tree_nr ) is

begin

if Nt_Empty(a) then
   null;
  else
   a.val:=main_function(a.val);
   Nt_Change(a.first_child);
   Nt_Change(a.brother);
```

```
end if;

end Nt_Change;
```

Chapter 3

Specifications of Quaternary generic package

3.1 Representation

A quaternary tree is a tree with 0 or 4 children. These 4 children are called north-west, north-east, south-west and south-east child.

3.2 Operations

The following operations are the same as the implemented ones for tree_nary package:

Qt_Create_Empty, Qt_Create_Leaf, Qt_Empty, Qt_Value, Qt_Father, Qt_Display, Qt_Search, Qt_Is_Leaf, Qt_Is_Root, Qt_Change_Value, Qt_Change.

New functions to code are:

- Qt_Build: create quaternary tree from a value and 4 child trees.
- Qt_North_West: returns the north-west child of quaternary tree.
- Qt_North_East: returns the north-east child of quaternary tree.
- Qt_South_West: returns the south-west child of quaternary tree.
- Qt_South_East: returns the south-east child of quaternary tree.

Chapter 4

Conception of Quaternay tree generic package

4.1 Structuration of data

Quaternary tree being a subcategory of n-ary tree, we declare a type "quaternary tree", which is a subtype of n-ary tree. The generic parameters are the same as those of n-ary tree package.

4.2 Algorithmic refining

We use "renames" ADA directive in specification part. This will allow to reuse all procedures and functions implemented into n-ary tree package. Example :

```
function Qt_Create_Empty return quat_tree renames Nt_Create_Empty;
```

We do the same thing for exceptions :

```
-- empty tree
treeq_empty: exception renames tree_empty;
-- no relation
relationq_empty: exception renames relation_empty;
```

We define a childq_empty exception for Qt_Build procedure.

4.2.1 procedure Qt_Build

```
-----------------------------------------------------------------
---- procedure Qt_Build: create quaternary tree from a value and
----                  and 4 child trees
---- parameters: tree_res, built tree
--             val, father value
----          a_nw, a_ne, a_sw, a_se, respectively north-west,
----          north-east, south-west and south-east children
---- post-conditions: if one of child is empty, we raise
----                childq_empty exception
-----------------------------------------------------------------

procedure Qt_Build( value: in T; t_nw, t_ne, t_sw, t_se: in out tree_quat; tree_res: out tree_quat )
     is

begin

-- if one of child is empty, we raise childq_empty exception
if (Qt_Empty(t_nw) or Qt_Empty(t_ne) or Qt_Empty(t_sw) or Qt_Empty(t_se)) then
   raise childq_empty;
-- otherwise we build quaternary tree
  else
   tree_res:=Qt_Create_Leaf(value);
   Nt_Insert_Child(tree_res,t_se);
   Nt_Insert_Child(tree_res,t_sw);
   Nt_Insert_Child(tree_res,t_ne);
   Nt_Insert_Child(tree_res,t_nw);

end if;
```

```
end Qt_Build;
```

4.2.2 function Qt_North_West

We use here Nt_Child function which returns the n-th child of a tree.

```
function Qt_North_West( a: in tree_quat ) return tree_quat is

begin

return Nt_Child(a,1);

end Qt_North_West;
```

We do the same thing for Qt_North_East (return Nt_Child(a,2)), Qt_South_West (return Nt_Child(a,3)), Qt_South_East (return Nt_Child(a,4)).

Chapter 5

Specifications of pimage package with compression

The aim est to achieve compression and decompression on an image using a quaternary tree. The images that we use are squares and their dimension is a power of 2. A color is defined from the 3 primary colors (red, green, blue) and is integer coded.

The recursive compression algorithm of a image with dimension n is as follows : if image is homogeneous, then we create a leaf whose root value is the color of image. If image is not homogeneous, we split it into 4 sub-images of dimension n/2, and we create a quaternary tree whose the four children are the 4 encoded sub-images.

Browsing all the quaternary tree, we get a set of values representing the compressed image.
It is specified that pimage package allows to :

- create a test image from a dimension and a matrix.
- load an image from a file.
- save an image into a file.
- display an image.
- load an encoded image from a file.
- save an encoded image into a file.
- achieve an operation on an image.

Chapter 6

Conception of pimage package

6.1 Structuration of data

We choose to define a color by a 32 bits value (pixel). Image is a matrix of pixels. We use dynamic arrays.

```
-- subtype tree_im
subtype tree_im is tree_quat;
-- definition of primary color
subtype color_prim is integer range 0..255;
-- definition of pixel: value coded on 32 bits
type pixel is mod 2**32;
-- image array
type image is array (integer range <>,integer range <>) of pixel;
```

6.2 Algorithmic refining

6.2.1 procedure im_to_tree

This procedure is building a tree from an image. We follows the encoding algorithm :

```
-------------------------------------------------------------------
---- procedure im_to_tree: crate a tree from an image
---- parameters: im the image, n its dimesnion, a the created tree
----            tol the tolerance used by function is_homogeneous
-------------------------------------------------------------------

procedure im_to_tree( im: in image; n: in integer; a: out tree_im; tol: in intensity) is

im_nw,im_ne,im_sw,im_se: image(1..n/2,1..n/2);
a_nw,a_ne,a_sw,a_se: tree_im;

begin
-- case where dimension equals 1
if n=1 then
   a:=Qt_Create_Leaf(integer(im(1,1)));
   -- if it is homogeneous, we create a leaf with average color
  else if is_homogeneous(im,n,tol) then
          a:=Qt_Create_Leaf(integer(col_aver(im,n)));
        else -- we build the tree dividing dimensions by two
          for i in 1..n/2 loop
        for j in 1..n/2 loop
            im_nw(i,j):=im(i,j);
      im_ne(i,j):=im(i,j+n/2);
      im_sw(i,j):=im(i+n/2,j);
      im_se(i,j):=im(i+n/2,j+n/2);
        end loop;
    end loop;

          im_to_tree(im_nw,n/2,a_nw,tol);
    im_to_tree(im_ne,n/2,a_ne,tol);
    im_to_tree(im_sw,n/2,a_sw,tol);
    im_to_tree(im_se,n/2,a_se,tol);
    Qt_Build(-1,a_nw,a_ne,a_sw,a_se,a);

      end if;
end if;
```

```
end im_to_tree;
```

One has to browse all the tree and save the sequence into a file. That is done with save_imagec and tree_to_imc.

6.2.2 function is_homogeneous

We explain here how is_homogeneous function deals with 3 possibles cases : tolerance is minimal (0), maximal (100), or between these two values. If it is maximal, then we return a boolean set to true and a leaf is created with the average value of the image (see procedure im_to_tree). If it is minimal, we check that all pixels have the same value. Finally, for the middle case, we test the inequality abs(primary_color-average) <= tolerance on each pixel.

```
--------------------------------------------------------------------
---- function is_homogeneous: function which checks if area of size n
----                       homogeneous. return a boolean
---- parameters: im image
----            size n of a area
----            tol tolerance for homogeneity
--------------------------------------------------------------------

function is_homogeneous( im: in image; n: in integer; tol: in intensity ) return boolean is

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

begin
result:=true;

case tol is
     -- maximal tolerance allowed, image is considered homegeneous
     when intensity'last => return true;
     -- minimal tolerance, we compare first point with all others
     when intensity'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;
      -- general case: we compute the average color and we extract the
      -- 3 average primary colors that we compare to image pixels with
      -- condition : abs(average primary color) <= tolerance
      when others => c_aver:=integer(col_aver(im,n));
                    i:=1;
         while i<=n and result loop
              j:=1;
         while j<=n and result loop
result:=(abs(get_blue(im(i,j))-get_blue(pixel(c_aver)))<=tol) and
(abs(get_green(im(i,j))-get_green(pixel(c_aver)))<=tol) and (abs(get_red(im(i,j))-get_red(pixel(
    c_aver)))<=tol);
         j:=j+1;
         end loop;
         i:=i+1;
         end loop;
                    return result;
end case;

end is_homogeneous;
```

6.2.3 procedure save_imagec

```
-----------------------------------------------------------------
---- procedure save_imagec: save a compressed image into file from
----                        a tree
---- parameters: a built tree, n the dimension of image
-----------------------------------------------------------------

procedure save_imagec( a: in tree_im ; n: in pixel ) is

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

begin

put("Enter a filename(4 characters): ");
get(filename);
skip_line;
create(f,out_file,filename);
write(f,n);
tree_to_imc(a,f);
close(f);

end save_imagec;
```

We call the recursive procedure tree_to_imc which browses all the tree.

6.2.4 procedure tree_to_imc

```
---------------------------------------------------------------
----- procedure tree_to_imc: allow to convert tree to compressed
-----                        sequence
----- parameters: a the built tree, f the logic name of file
---------------------------------------------------------------

procedure tree_to_imc( a: in tree_im; f: in pack_pixel.file_type ) is

begin

if Qt_Empty(a) then
   null;
  else
   if Qt_Value(a)=-1 then
      write(f,2**24);
     else
      write(f,pixel(Qt_Value(a)));
   end if;
   tree_to_imc(Qt_Child(a,1),f);
   tree_to_imc(Qt_Brother(a,1),f);
end if;

end tree_to_imc;
```

6.3 Algorithmic refining for decompression

Firstly, we must build a tree from encoded sequence (procedure load_im_encod and imc_to__tree) and then rebuild image from this tree (procedure tree_to_im).

6.3.1 procedure load_im_encod

```
---------------------------------------------------------------
---- procedure load_im_encod : build a tree from compressed
----                           sequence
---- parameters: a the rebuilt tree, n the dimension of original
----             image
---------------------------------------------------------------
```

```
procedure load_im_encod( a: out tree_im ; n: out pixel ) is

filename: string(1..4);
f: pack_pixel.file_type;
-- dimension of original image
dim:pixel;

begin

put("Enter a filename(4 characters): ");
get(filename);
skip_line;
open(f,in_file,filename);
-- we read dimension
read(f,dim);
n:=dim;
a:=imc_to_tree(f);
close(f);

end load_im_encod;
```

Here we use auxiliary function imc_to_tree.

6.3.2 function imc_to_tree

This function allows to build a tree from encoded sequence of a compressed image.

```
-------------------------------------------------------------
---- function imc_to_tree: return a tree from of a compressed
----                        image
---- parameters: f the logic name of compressed file
-------------------------------------------------------------

function imc_to_tree( f: in pack_pixel.file_type ) return tree_im is

-- children of quaternary tree
a,a_nw,a_ne,a_sw,a_se: tree_im;
v: pixel;

begin

read(f,v);
-- if value equals -1 (coded by 2**24), we create a leaf
if v /= 2**24 then
   a:=Qt_Create_Leaf(integer(v));
  else
   a_nw:=imc_to_tree(f);
   a_ne:=imc_to_tree(f);
   a_sw:=imc_to_tree(f);
   a_se:=imc_to_tree(f);
   Qt_Build(-1,a_nw,a_ne,a_sw,a_se,a);
end if;
return a;

end imc_to_tree;
```

6.3.3 procedure tree_to_im

Image can be rebuilt from tree. Tree is tree_im and output image is im. We use i_b, i_f, j_b, j_f to get the shift between recursive calls.

```
-------------------------------------------------------------------
---- procedure tree_to_im: allow to reconstruct original image from
----                        compressed image
---- parameters: a the rebuilt tree, im image to build
----             i_b,i_f,j_b,j_f the indexed (begin and final)
----             n the dimension of original image
```

```
------------------------------------------------------------------

procedure tree_to_im ( a: in tree_im ; im: out image ; i_b,i_f,j_b,j_f: in integer ) is

begin

-- terminal case
if Qt_Empty(a) then
   null;-- general case
  else if Qt_Value(a) /= -1 then
         for i in i_b..i_f loop
       for j in j_b..j_f loop
           im(i,j):=pixel(Qt_Value(a));
    end loop;
    end loop;
        else -- we browse the children
    tree_to_im(Qt_North_West(a),im,i_b,i_f-(i_f-i_b+1)/2,j_b,j_f-(j_f-j_b+1)/2);
    tree_to_im(Qt_North_East(a),im,i_b,i_f-(i_f-i_b+1)/2,j_f-(j_f-j_b+1)/2+1,j_f);
    tree_to_im(Qt_South_West(a),im,i_f-(i_f-i_b+1)/2+1,i_f,j_b,j_f-(j_f-j_b+1)/2);
    tree_to_im(Qt_South_East(a),im,i_f-(i_f-i_b+1)/2+1,i_f,j_f-(j_f-j_b+1)/2+1,j_f);
        end if;
end if;

end tree_to_im;
```

6.3.4 procedure thresh_im

For an example of operation, we choose to do a binary threshold.

```
---------------------------------------------------------
----- procedure thresh_im: perform a threshold on image
----- parameters: im the image to process, n its dimension
-----           threshold: the threshold
---------------------------------------------------------

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

begin

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

end thresh_im;

end pimage;
```

Chapter

7 Validation of packages

7.1 n-ary and quaternary packages

Test program main_tree_nary.adb and main_tree_quater.adb use respectively tree_nary.adb and tree_quater.adb package. It is possible to make a tree for checking the good operating of all main above procedures and functions.

7.2 pimage package

Tets program main_pimage.adb allows to create an image and validate compression and decompression. An example of image is displayed as :

10	0	0	0	255	30	30	30
0	0	0	0	255	30	30	30
0	0	100	100	255	255	255	255
0	0	100	100	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 7.1: *Content of test image 8×8*

Its size equals to 260 bytes (64×4+4) because we save also the dimension. We take into account a tolerance to see if image is homogeneous. The tree built with a minimal tolerance (0) is represented on the following figure :

```
-1
|
+-- -1
    |
    +-- -1
    |   |
    |   +-- 10
    |   |   0
    |   |   0
    |   |   0
    |   0
    |   0
    |   100
    -1
    |
    +-- -1
    |   |
    |   +-- 255
    |   |   30
    |   |   255
    |   |   30
    |   30
    |   255
    |   255
    255
    -1
    |
    +-- 255
        255
        255
        20
```

Figure 7.2: *Content of tree for compression with minimal tolerance*

Image compressed size equals to 104 bytes (25×4+4) with 4 bytes for dimension, which was expected. With a maximal tolerance (100), we get a leaf with the average color of image (161) because this one is considered as homogeneous from criterion into is_homogeneous function.

with a tolerance equal to 80, we get this tree :

```
-1
|
+-- 25
    -1
    |
    +-- -1
    |   |
    |   +-- 255
    |   |   30
    |   |   255
    |   |   30
    |   30
    |   255
    |   255
    255
    -1
    |
    +-- 255
        255
        255
        20
```

Figure 7.3: *Content of tree for compression with tolerance equal to 80*

We can notice that there will be a loss of information regards to test image with compression. Indeed, the north-west part of image is seen as homogeneous with this tolerance. This is illustrated on this figure :

```
 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: *Content of decompressed image 8×8 a tolerance equal to 80*

For each case, we save compressed image and check decompression.

Conclusion

This project has enabled us to create a package to make compression and decompression of images. The operations implemented for quaternary trees were reused and validated for encoding algorithm.

Sources of this project can be downloaded here :