Deux usages des arbres

Luc.Maranget@inria.fr
http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Deux exemples d'arbres

- Les arbres-termes : le calcul propositionnel.
- Les arbres structurants : diviser le plan en quatre.

Un grand classique

Une expression booléenne e (ou proposition) est :

- Vrai ou faux, T ou F,
- Une variable $x_0, \ldots x_{N-1}$,
- La négation $\neg(e)$,
- Une conjonction $(e_1 \wedge e_2)$, une disjonction $(e_1 \vee e_2)$.

Note :

- Conceptuellement c'est aussi simple que les expressions arithmétiques.
- Techniquement, c'est un rien plus complexe.

La classe des propositions

C'est assez moche.

```
class Prop {
  final static int FALSE=0, TRUE=1, VAR=2, NOT=3, OR=4, AND=5 ;
  int nature ; int asVar ; Prop left, right ;

  private Prop (int nature) { this.nature = nature ; }

  private Prop (int nature, int asVar) {
    this.nature = nature ; this.asVar = asVar ;
  }

  private Prop (int nature, Prop left) {
    this.nature = nature ; this.left = left ;
  }

  private Prop (int nature, Prop left, Prop right) {
    this.nature = nature ; this.left = left ; this.right = right ;
  }
}
```

Construction des termes

Il devient hasardeux de se fier aux constructeurs.

On utilisera plutôt des méthodes statiques bien nommées.

```
static Prop mkTrue() { return new Prop(TRUE) ; }

static Prop mkVar(int no) { return new Prop(VAR, no) ; }

static Prop mkNot(Prop e) { return new Prop(NOT, e) ; }

static Prop mkAnd(Prop e1, Prop e2) {
  return new Prop(AND, e1, e2) ;
}
...
```

On peut aussi ajouter d'autres connecteurs logiques, par ex.

```
static Prop mkImplies (Prop e1, Prop e2) {
  return mkOr(mkNot(e1), e2) ;
}
```

Affichage

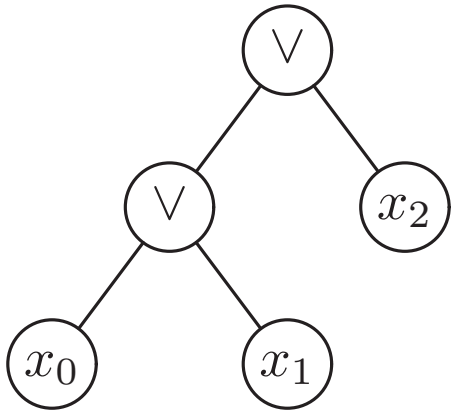
On souhaite cette fois redéfinir `toString`.

```
public String toString() {
  switch (nature) {
  case TRUE: return "true" ;
  case FALSE: return "false" ;
  case VAR: return "x" + asVar ;
  case NOT: return  "!(" + left + ")" ;
  case OR: return  "(" + left + " || " + right + ")" ;
  case AND: return  "(" + left + " && " + right + ")" ;
  default: throw new Error("Prop (toString)") ;
  }
}
```

Bénéfice : `System.out.println(e)` fonctionne.

Parcours ? Infixe en quelque sorte.

Fonctionnement



- `toString()` du sous-arbre de gauche :
  - ▷ Deux feuilles *"x0"* et *"x1"*.
  - ▷ Concaténation :*"(" + "x0" + " || " + "x1" + ")"*.

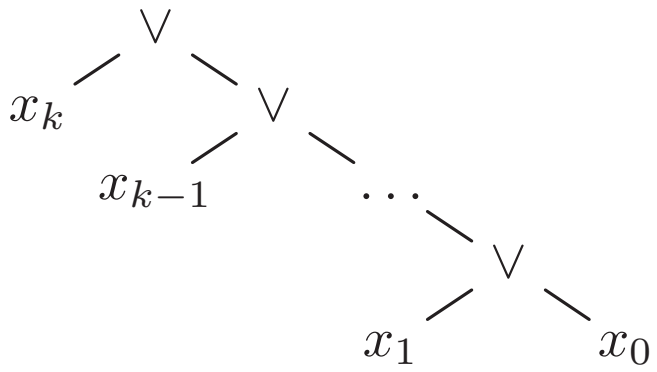
  Soit au final pour ce sous-arbre *"(x0 || x1)"*.
- `toString()` du sous-arbre de droite : *"x2"*.
- Concaténation :
  "(" + "(x0 || x1)" + " || " + "x2" + ")"

Coût de `toString()`

Considérer la suite d'expressions

$$E_0 = x_0, \qquad E_{k+1} = x_{k+1} \vee E_k$$

- L'arbre E_k ressemble plutôt à une liste.



- L'arbre E_k possède $2k + 1$ nœuds.
- E_k`.toString()` est quadratique, les contaténations produisent des chaînes de taille au moins $3 + 5 + \cdots + (2k + 1)$.

toString « linéaire »

Avec le `StringBuilder` et sa méthode `append(String s)`, qui ajoute la chaîne s à la fin du `StringBuilder`.

```
private void inStringBuilder(StringBuilder r) {
   switch (nature) {
   case TRUE:
     r.append("true") ; break ;
   case FALSE:
     r.append("false") ; break ;
   case VAR:
     r.append("x" + asVar) ; break ;
   case NOT:
     r.append("!(") ;
     left.inStringBuilder(r) ;
     r.append(")") ;
     break ;
...
```

toString « linéaire » II

```
    ...
    case AND:
      r.append("(") ;
      left.inStringBuilder(r) ;
      r.append(" && ") ;
      right.inStringBuilder(r) ;
      r.append(")") ;
      break ;
    }
  }

  public String toString() {
    StringBuilder r = new StringBuilder () ;
    inStringBuilder(r) ;
    return r.toString() ;
  }
}
```

Évaluation des propositions

Les règles sont normalement bien connues :

e	$\neg(e)$
F	T
T	F

e_1	e_2	$(e_1 \vee e_2)$	$(e_1 \wedge e_2)$
F	F	F	F
F	T	T	F
T	F	T	F
T	T	T	T

L'évaluation se fait respectivement à un environnement (une table d'associations : $x_i \mapsto b$).

Les associations ont été vues lors de l'amphi 04. Ici on peut utiliser un tableau directement (variables indicées).

```
static boolean eval(Prop e, boolean [] env) { ... }
```

Évaluation

```
static boolean eval(Prop e, boolean [] env) {
  switch (e.nature) {
  case TRUE: return true ;
  case FALSE: return false ;
  case VAR: return env[e.asVar] ;
  case NOT: return !eval(e.left, env) ;
  case OR:
    return eval(e.left, env) ||  eval(e.right, env) ;
  case AND:
    return eval(e.left, env) &&  eval(e.right, env) ;
  default:
    throw new Error("Prop (eval)") ;
  }
}
```

Application du calcul des propositions

La direction d'une crèche souhaite réglementer les jouets apportés par les enfants.

Les jouets sont décrits selon cinq critères : petit/grand, vert/pas vert, peluche/pas peluche, électrique/pas électrique, avec piles/sans piles.

C'est à dire, nous avons cinq variables booléennes.

```
Prop petit = mkVar(0) ;
Prop vert = mkVar(1) ;
...
Prop piles = mkVar(4) ;
```

Les règles de la crèche

- Les jouets doivent être de petite taille, sauf les peluches.

```
Prop r1 = mkOr(petit, peluche) ;
```

- Un jouet est soit, vert, soit grand, soit les deux.

```
Prop r2 = mkOr(vert, mkNot(petit)) ;
```

- Les jouets électriques sont accompagnés de leurs piles,

```
Prop r3 = mkImplies(electrique, piles) ;
```

- Il est interdit d'apporter des piles et une peluche.

```
Prop r4 = mkNot(mkAnd(piles, peluche)) ;
```

- Touts les jouets verts sont des peluches.

```
Prop r5 = mkImplies(vert, peluche) ;
```

Les règles de la crèche II

Les règles de la crèche sont la conjonction des cinq règles élémentaires.

```
Prop rs = mkAnd(r1, mkAnd(r2, mkAnd(r3, mkAnd(r4, r5)))) ;
```

Le contrôle à l'entrée

Oscar arrive avec son (grand) train électrique rouge et ses piles, peut-il-rentrer ?

- Le jouet n'est pas petit, n'est pas vert, n'est pas une peluche, est électrique, et il y a des piles :

  ```
  boolean [] oscar = { false, false, false, true, true } ;
  ```

- On vérifie facilement que le train d'Oscar est interdit (à cause de la première règle). La machine le vérifiera encore plus facilement.

  ```
  boolean okOscar = eval(rs, oscar) ;
  ```

Une question bien légitime

- Y-a-t-il des jouets autorisés ?
- Existe-t-il un environnement tel que : `eval(rs, env)` renvoie **true** ?

Cette tâche se décompose clairement en deux :

1. Fabriquer tous les environnements,
2. Et pour chacun, évaluer les règles.

Afficher tous les environnements

```
static void println(boolean [] env) {
  for (int i = 0 ; i < env.length ; i++) {
    System.out.print(env[i] ? 'T' : 'F') ;
    // NB: expression conditionnelle « e1 ? e2 : e3 »
  }
  System.out.println() ;
}
static void printAll(int v, boolean [] env) {
  if (v >= env.length) {
    println(env) ;
  } else {
     env[v] = true ;  printAll(v+1, env) ;
     env[v] = false ; printAll(v+1, env) ;
  }
}
```

Note : Mission de `printAll(v, env)` : afficher tous les environnements qui commencent par $b_0 b_1 \ldots b_{v-1}$ donnés.

Afficher tous les environnements

Voici une représentation imagée de l'induction pratiquée.

$$A_{n+1} = \begin{array}{c|l} T & \\ T & \\ \vdots & A_n \\ T & \\ \hline F & \\ F & \\ \vdots & A_n \\ F & \end{array}$$

Si on sait énumérer tous les environnemnts à n variables, alors on sait énumérer tous les environnemnts à $n+1$ variables.

Afficher tous les jouets autorisés

Comme l'affichage de tous les environements, avec une vérific ation supplémentaire.

```
static void printAll(Prop rs, int v, boolean [] env) {
  if (v >= env.length) {
     if (eval(rs, env)) println(env) ;
  } else {
     env[v] = true ;
     printAll(rs, env) ;
     env[v] = false ;
     printAll(rs, env) ;
  }
}
```

Existe-t-il au moins un jouet autorisé ?

Pour tous les environements, on évalue `rs`, jusqu'à trouver **true**.

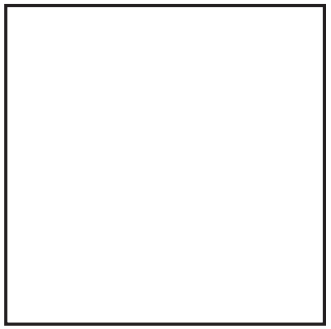
```
static boolean satisfiable(Prop rs, int v, boolean [] env) {
  if (v >= env.length) {
     return eval(rs, env) ;
  } else {
     env[v] = true ;
     if (satisfiable(rs, v+1, env)) return true ;
     env[v] = false ;
     return satisfiable(rs, v+1, env) ;
  }
}
```

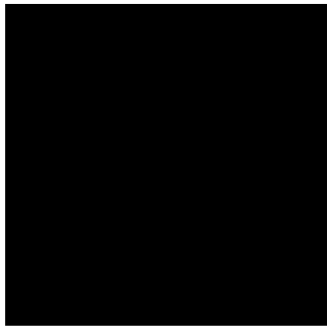
Remarquer Le parcours interrompu par une évaluation positive.

Un usage géométrique

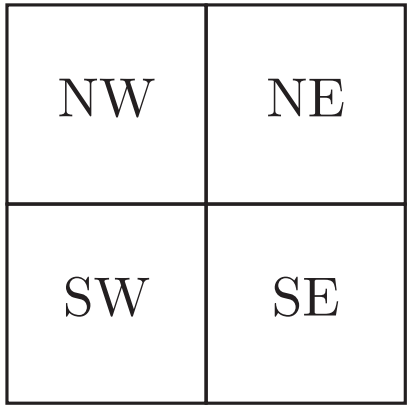
Une vision hiérarchique des images (carrées). Une image est :

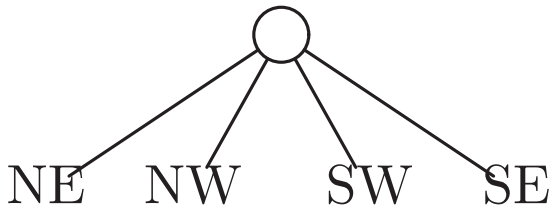
- Soit toute blanche ou toute noire (de couleur uniforme).





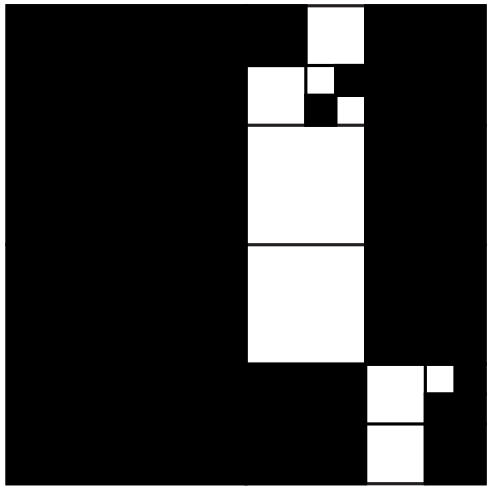
- Soit formée de quatre images.



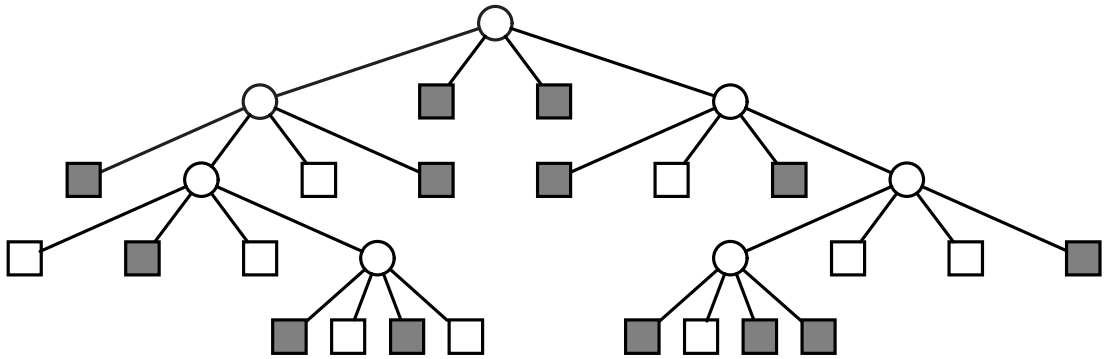


Exemple

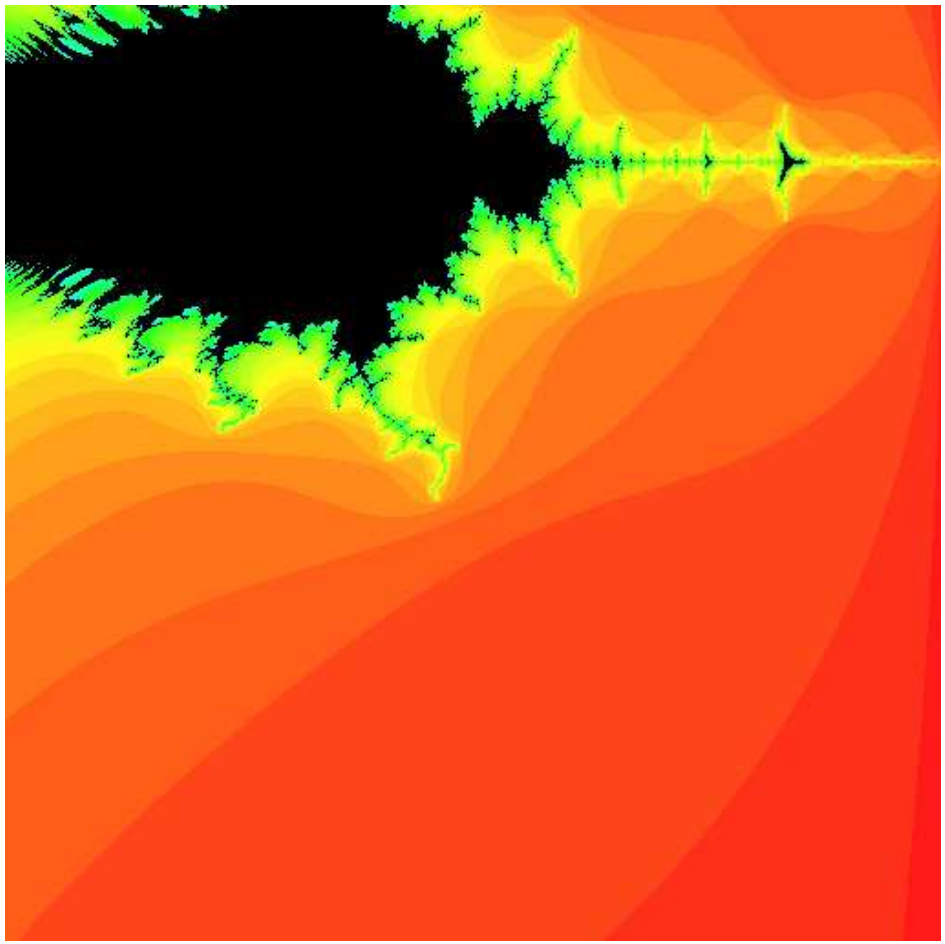
- Une image 16×16.



- Son quadtree

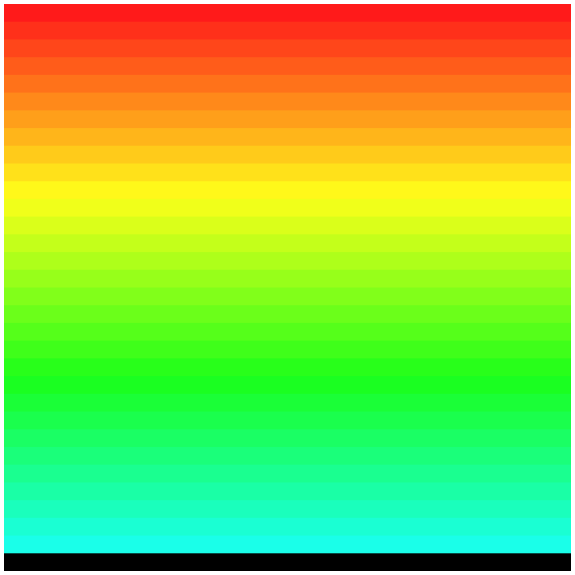


Exemple en couleurs, une image



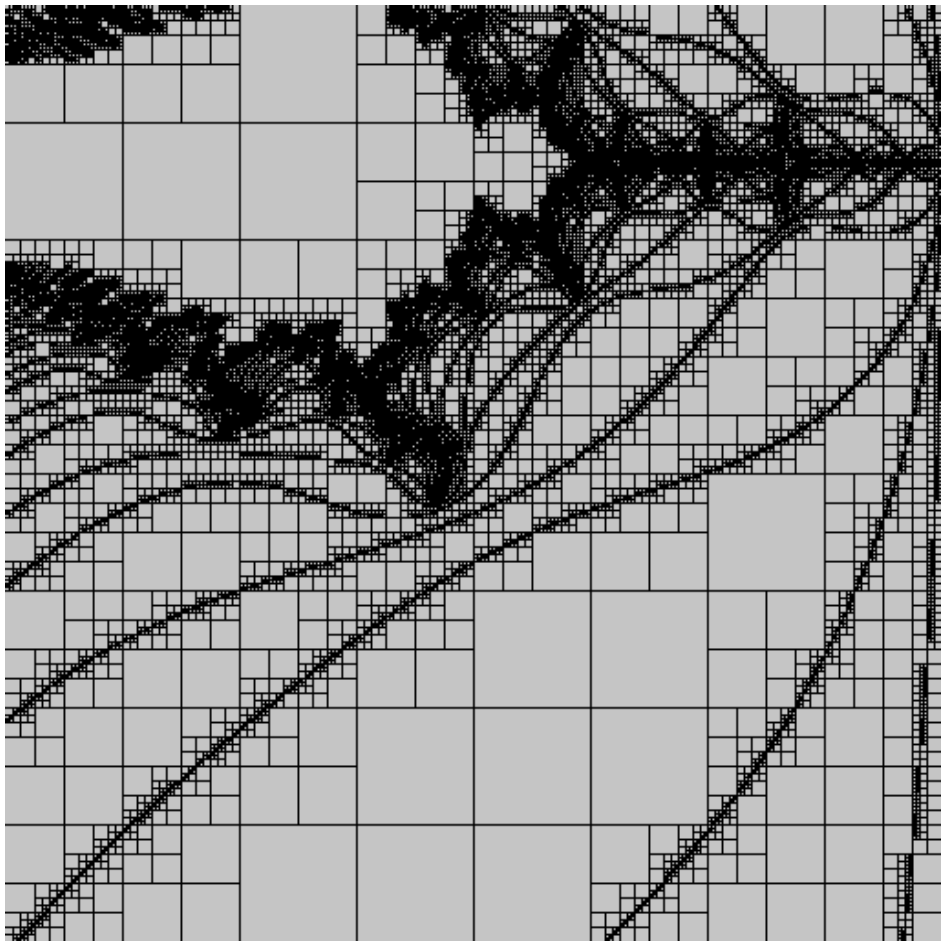
Une échelle de couleurs

Notre image contient peu de couleurs, chaque couleur est en fait une valeur (ici de 0 à 31).

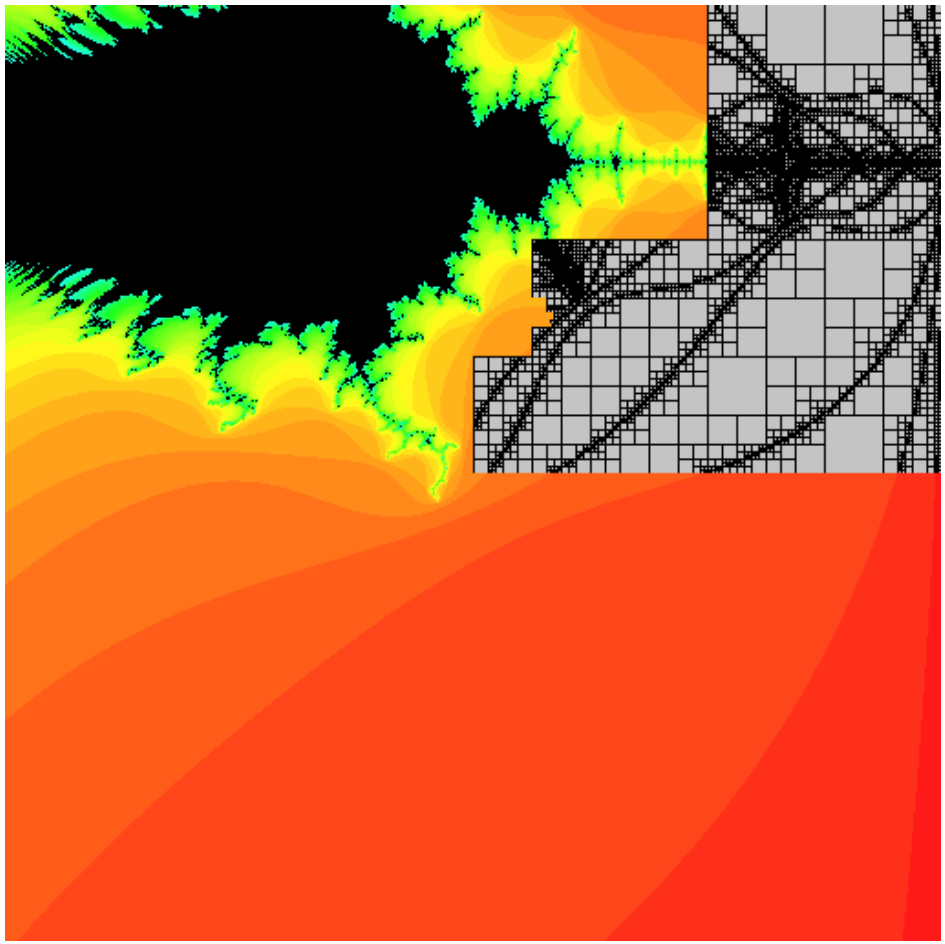


On rencontre souvent ce style d'images qui rendent compte de phénomènes physiques ou mathématiques.

Exemple en couleurs, le quadtree



Exemple en couleurs, un peu des deux



Type des quadtrees

```
class Quad {
  int nature ;
  final static int LEAF=0, NODE=1 ;

  int color ; // Les feuilles
  Quad (int color) {
    this.nature = LEAF ; this.color = color ;
  }

  Quad sw, nw, ne, se ; // Les noeuds internes
  Quad(Quad sw, Quad nw, Quad ne, Quad se) {
    this.nature = NODE ;
    this.sw = sw ; this.nw = nw ;
    this.ne = ne ; this.se = se ;
  }
}
```

Implicitement Un `Quad` est une image (carrée).

Deux représentations possibles de nos images

Rappelons que ici une couleur est un entier entre 0 et 31.

- Une matrice `SIZE` $\times$ `SIZE` de couleurs.

```
int [] [] img = new int[SIZE][SIZE] ;
```

- Ou un objet de la classe `Quad`.

Ces deux structures de données sont deux réalisations machine de la même chose (l'image).

Remarque importante

- Tout le quadtree représente une image carrée (coté `SIZE`).
- Si `q` représente le carré de côté t et de coordonnées (i, j), ...
- Alors, `q.nw` (par ex) est l'image de côté $t/2$ et de coordonnées $(i, j + t/2)$.

Exemple d'opération : trouver la couleur

- La matrice :

```
static int getColor(int [] [] img int i, int j) {
  return img[i][j] ;
}
```

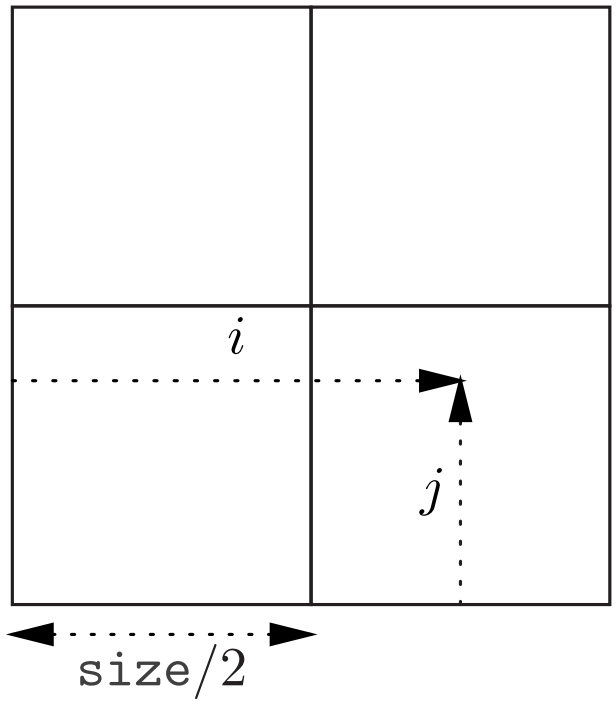
- Le quadtree :

```
static int getColor(Quad q, int i, int j) {
  return getColor(q, SIZE, i, j) ;
}
```

  - ▷ Si le quadtree est une feuille, c'est la couleur de la feuille.
  - ▷ Sinon recherche la couleur dans le bon quadrant.

Trouver le bon quadrant

- Les coordonnées i et j sont relatives au point origine d'un carré de côté `size`.



- Ici, le point défini par (i, j) devient le point défni par $(i-\texttt{size}/2, j)$ dans le quadrant sud-est.

Trouver la couleur d'un point dans le quadtree

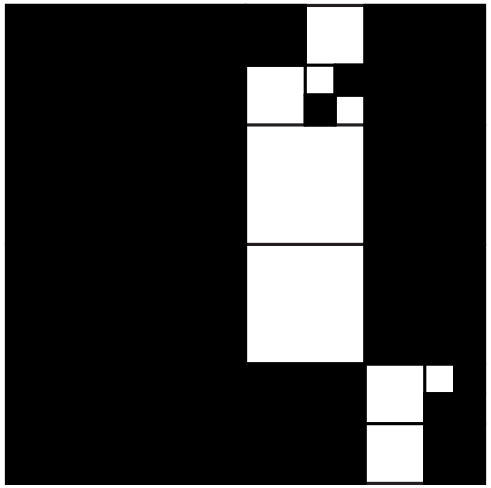
```
static int getColor(Quad q, int size, int i, int j) {
  if (q.nature == LEAF) {
    return q.color ;
  } else {
     int t = size/2 ;
     if (i < t) { // A l'ouest.
       if (j < t)
          return getColor(q.sw, t, i, j) ;
        else
          return getColor(q.nw, t, i, j-t) ;
     } else { // A l'est.
       if (j < t)
          return getColor(q.se, t, i-t, j) ;
        else
          return getColor(q.ne, t, i-t, j-t) ;
     }
  }
}
```

Code itératif

```
static int getColor(Quad q, int size, int i, int j) {
  while (q.nature == NODE) {
     size /= 2 ; // pour size = size / 2 ;
     if (i < size) { // A l'ouest.
       if (j < size) {
          q = q.nw ;
        } else {
          q = q.sw ; j -= size ;
        }
     } else { // A l'est.
       i -= size ;
       if (j < size) {
         q = q.ne ;
       } else {
         q = q.se ; j -= size ;
       }
    }
  }
  return q.color ;
}
```

Intérêt du quadtree

- Une méthode simple de compression d'une image. Plaçons nous en noir et blanc.
  - ▷ Une image 16×16.



```
1111111111001111
1111111111001111
1111111100011111
1111111100101111
1111111100001111
1111111100001111
1111111100001111
1111111100001111
1111111100001111
1111111100001111
1111111100001111
1111111100001111
1111111111110001
1111111111110011
1111111111110011
1111111111110011
```

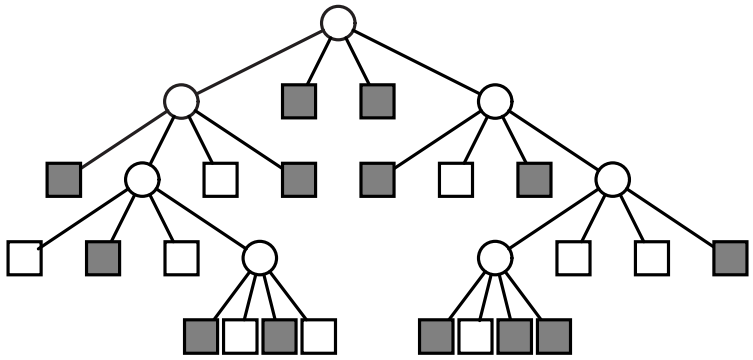
▷ Traduction du quadtree (parcours préfixe) :

★ Une feuille blanche : 00

★ Une feuille noire : 01

★ Un nœud NE,NW,SW,SE : 1$\underline{\text{NE}}\,\underline{\text{NW}}\,\underline{\text{SW}}\,\underline{\text{SE}}$

▷ Par exemple :



$1\underline{T_1}\,\underline{T_2}\,\underline{T_3}\,\underline{T_4}$

$1\underline{T_1}0101\underline{T_4}$

...

1101100010010

1000100000101

0110100011101

000101000001

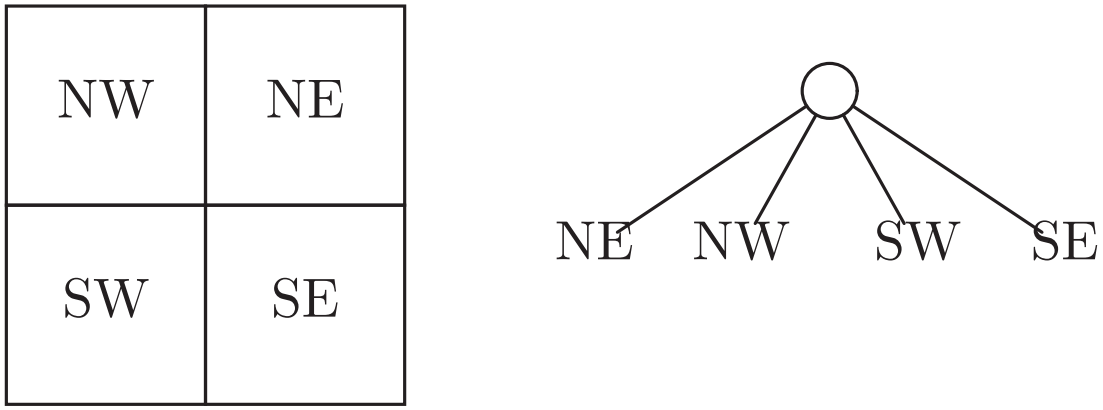
Au final, 51 bits.

Autre intérêt

Certaines opérations sont plus faciles/naturelles/voire efficaces que sur les tableaux de couleurs.

- Les rotations,
- Les changements de couleurs.

Exemple : la rotation



La rotation positive d'un quart de tour :

$$\text{NE} \to \text{NW} \to \text{SW} \to \text{SE} \to \text{NE}$$

C'est à dire NE (tourné d'un quart de tour) devient NW, etc.

Programmation de la rotation

```
static Quad rot(Quad q) {
  if (q.nature == LEAF) {
    return q ;
  } else {
    Quad sw = rot(q.sw) ;
    Quad nw = rot(q.nw) ;
    Quad ne = rot(q.ne) ;
    Quad se = rot(q.se) ;
    return new Quad (nw, ne, se, sw) ;
  }
}
```

Autre intérêt

L'affichage : si afficher un carré de couleur ne dépend pas de la taille du carré.

Hypothèse réaliste

- Un dessin demande une communication sur le réseau (coût d'une communication peu dépendant de la taille).
- On souhaite rafraîchir l'écran à chaque changement (coût du rafraîchissement bien plus important que le dessin lui même).

Dès lors le quadtree est intéressant, car il minimise les opérations de dessin.

Dessin traditionnel

```
static void draw(int [] [] img) {
  for (int i = 0 ; i < SIZE ; i++) {
    for (int j = 0 ; j < SIZE ; j++) {
      fillSquare(i,j,1,img[i][j]) ;
      // Remplir un carré avec une couleur
      // arguments (i, j, t, c)
      //   * (i,j) -> position coin en bas à gauche
      //   * t -> côté du carré
      //   * c -> couleur
    }
  }
}
```

Dessin du Quadtree

On considère qu'un quatree représente un carré, de coordonnées (i, j) et de taille sz.

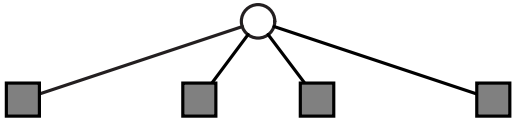
```
static void draw(Quad q, int i, int j, int sz) {
    if (q.nature == LEAF) {
      fillSquare(i, j, sz, q.color) ;
    } else {
      int nsz = sz/2 ;
      draw(q.sw, i, j, nsz) ;
      draw(q.nw, i, j+nsz, nsz) ;
      draw(q.ne, i+nsz, j+nsz, nsz) ;
      draw(q.se, i+nsz, j, nsz) ;
    }
  }

static void draw(Quad q) { draw(q, 0, 0, SIZE) ; }
```

Fabrication du quadtree

Toute l'astuce est bien entendu d'éviter les divisions inutiles.

- Éviter :



- Préférer :

```
static boolean monochrome(Quad q1, Quad q2, Quad q3, Quad q4) {
 return
   (q1.nature == LEAF && q2.nature == LEAF &&
    q3.nature == LEAF && q4.nature == LEAF) &&
   (q1.color == q2.color && q2.color == q3.color &&
    q3.color == q4.color) ;
 }
```

Fabrication du quadtree

```
static Quad toQuad(int [] [] t, int i, int j, int sz) {
  if (sz == 1) {
    return new Quad(t[i][j]) ;
  } else {
    int nsz = sz/2 ;
    Quad sw = toQuad(t, i, j, nsz) ;
    Quad nw = toQuad(t, i, j+nsz, nsz) ;
    Quad ne = toQuad(t, i+nsz, j+nsz, nsz) ;
    Quad se = toQuad(t, i+nsz, j, nsz) ;
    if (monochrome(sw, nw, ne, se)) {
      return sw ;
    } else {
      return new Quad (sw, nw, ne, se) ;
    }
  }
}
```