

Rapport du Projet ASD3

Commandes de compilation et exécution.....	1
Algorithmes pseudo-code.....	1
Construction du QuadTree.....	1
Conversion du QuadTree en image / surface.....	2
Compression Lambda.....	3
Compression dynamique - rho.....	3
Autres algorithmes utilisés par la compression rho.....	4
Lister les brindilles (cf. ligne 6 de “compressRho”).....	4
Mise à jour de la liste de brindilles (cf. ligne 10 de “compressRho”).....	5
Class Java et détail des structures de données utilisées.....	5
Classe QuadTree.....	5
Classe PGM.....	6
Classe TwigList.....	7
Classe ChildOf.....	8
Classe AVL.....	8
Schématisation des structures de données.....	9
Exemple de résultats d'exécutions:.....	11

Commandes de compilation et exécution

Il est également possible de tester ce programme en mode non-interactif avec des images png et jpg.

```
../HamonRimbert $ javac -cp src -d bin src/*.java
../HamonRimbert $ java -cp bin Main
../HamonRimbert $ java -cp bin Main filename.pgm 15
```

Algorithmes pseudo-code

Construction du QuadTree

Fonction : créer(Matrice<Entier> surface, Point 0, Entier cote) : QuadTree, Entier

- Rôle : construction, découpe la surface et crée récursivement le QuadTree en renvoyant le nombre de nœuds du quadtree créé.
- Entrée : la surface globale; les coordonnées de la sous-surface représentée par notre arbre (donnée par un point d'origine et une longueur de côté).
- Sortie : le quadtree nouvellement créé; le nombre de noeuds dans le quadtree
- Précondition : $cote = 2^n \times 2^n$
- Complexité : parcours de la surface : $O(cote^2)$

Variables

QuadTree A; Entier n1, n2, n3, n4;

Début

```

1 | Si longueur = 1 alors
2 | | retourner( QuadTree( val ← surface[1][1], fils ← null) ; 1 )
3 | Sinon
4 | | (nordOuest(A), n1) ← créer(surface, 0, cote/2)
5 | | (nordEst(A), n2) ← créer(surface, Point(0.x, 0.y+cote/2), cote/2)
6 | | (sudEst(A), n3) ← créer(surface, Point(0.x+cote/2, 0.y), cote/2)
7 | | (sudOuest(A), n4) ← créer(surface, Point(0.x+cote/2, 0.y+cote/2),cote/2)
8 | | Si ( fuseEqualSon(A) = 1 ) alors //cf (1) ci-dessous
9 | | retourner ( A ; 1 )
10| | Sinon
11| | retourner ( A ; n1+n2+n3+n4 )
12| Fin Si-Sinon

```

Fin

(1)- Complexité de “fuseEqualSon(QuadTree A)” : effectue 3 comparaisons pour vérifier l'égalité entre les fils, puis 5 affectations si ceux-ci sont égaux : cela revient à un temps constant.

Fonction : créer(Matrice<Entier> surface) : QuadTree, Entier

- Rôle : construction, initialise la découpe de la surface et crée récursivement le QuadTree tout en renvoyant le nombre de nœuds en fin de construction.
- Entrée : La surface à représenter
- Sortie : le quadtree nouvellement créé; le nombre de noeuds dans le quadtree
- Complexité : parcours de la surface : $O(|surface|^2)$

Début

```

1 | Si vide(surface) alors
2 | | retourner( QuadTree( ) ; 0 )
3 | Sinon
4 | | retourner créer(surface, (0;0), |surface|)
5 | Fin Si-Sinon

```

Fin

Conversion du QuadTree en image / surface

Procédure : versSurface(QuadTree A, Matrice<Entier> surface, Point 0, Entier cote)

- Rôle : complète la surface allouée au préalable en parcourant récursivement le quadtree
- Entrée : la surface allouée incomplète; les coordonnées de la sous-surface correspondant au quadtree (donnée par un point d'origine et une longueur de côté).
- Post-condition : la partie de la matrice de surface correspondant aux coordonnées doit être entièrement remplie avec les données correspondantes du quadtree.
- Précondition : $cote = 2^n \times 2^n$
- Complexité : remplissage de la surface : $O(cote^2)$

Début

```

1 | Si estCouleur(A) alors
2 |     Pour ligne allant de 0.y à 0.y + cote faire
3 |         Pour colonne allant de 0.x à 0.x + cote faire
4 |             surface[ligne][colonne] ← valeur(A)
5 |         Fin Pour
6 |     Fin Pour
7 | Sinon
8 |     versSurface(nordOuest(A), surface, 0, cote/2)
9 |     versSurface(nordEst(A), surface, Point(0.x, 0.y + cote/2), cote/2)
10|     versSurface(sudEst(A), surface, Point(0.x + cote/2, 0.y + cote/2), cote/2)
11|     versSurface(sudOuest(A), surface, Point(0.x + cote/2, 0.y), cote/2)
12| Fin Si-Sinon

```

Fin

Fonction : versSurface(QuadTree A, Entier longueur) : Matrice<Entier>

- Rôle : convertit le quadtree en surface en allouant celle-ci selon la longueur de son côté puis appelle la procédure récursive, ci-dessus, afin de remplir la surface.
- Entrée : la dimension de la surface à allouer (la longueur du côté suffit, la surface étant carrée).
- Sortie : la surface correspondant au quadtree.
- Complexité : remplissage de la surface : $O(longueur^2)$

Début

```

1 | Si vide(A) alors retourner Matrice_vide(0, 0) //Matrice de taille 0x0
2 |
3 | surface ← Matrice_vide(longueur + 1, longueur + 1) //Matrice de taille
   |             longueur + 1 x longueur + 1
4 | versSurface(A, surface, Point(0, 0), longueur)
5 | retourner surface

```

Fin

Compression Lambda

Fonction : compressLambda(QuadTree A) : Entier

- Rôle : fusionne toutes les brindilles présentes dans l'arbre à un instant t, et maintient l'arbre en forme correcte (i.e. un noeud n'a jamais quatre feuilles identiques pour fils immédiat)

- Entrée : QuadTree A l'arbre à compresser
- Complexité : $O(n)$ n étant le nombre de noeud dans l'arbre

VariablesEntier varNode**Début**

```

1 | Si non vide(A) et estZone(A) alors
2 | | Si estBrindille(A) alors
3 | | | A.luminosité ← moyenneLogarithmique(A) //cf (1) ci-dessous
4 | | | retourner - 4
5 | | Sinon
6 | | | varNode ← compressLambda(A.1)
7 | | | varNode ← varNode + compressLambda(A.2)
8 | | | varNode ← varNode + compressLambda(A.3)
9 | | | varNode ← varNode + compressLambda(A.4)
10| | | varNode ← varNode - 4 * fuseEqualSon(A) //cf (2) ci-dessous
11| | | retourner varNode
12| | Fin Si-Sinon
13| Sinon
14| | retourner 0
15| Fin Si-Sinon
Fin

```

(1)- Complexité de “moyenneLogarithmique(QuadTree A)” : calcule une moyenne logarithmique sur 4 éléments, d'où un temps constant.

(2)- Complexité de “fuseEqualSon(QuadTree A)” : voir construction ci-dessus.

Compression dynamique - rho

Procédure : compressRho(QuadTree A, Entier rho, Entier nbNoeudsInit)

- Rôle : Comprime en interdisant une perte trop importante d'information
- Entrée : Un entier rho qui correspond au pourcentage de noeuds qui doivent rester par rapport au nombre initial de noeud; A le QuadTree compressé
- Complexité : $O(\text{nb_noeud} * (\text{hauteur}(\text{QuadTree}) + \log_2(\text{nb_brindille})))$

VariableTwigList brindilleListe**Début**

```

1 | Si non vide(A) ou rho == 100 alors
2 | | retourner 0
3 | Fin Si
4 |
5 | nbNoeudsCourant ← nbNoeudsInit
6 | listerBrindille(brindilleListe, null, A) //A n'a pas de père
7 |
8 | Tant que (non vide(brindilleListe) et nbNoeudsCourant > nbNoeudsInit * rho%)
9 | | faire :
10| | | brindille ← noeudACompresser(brindilleListe) //cf (1) ci-dessous
11| | | nbNoeudsCourant ← nbNoeudsCourant + compressLambda(brindille)
12| | | //cf (2) ci-dessous
13| | | nbNoeudsCourant ← nbNoeudsCourant + update(brindilleListe)
14| | Fin tant que
15| retourner (nbNoeudsCourant - nbNoeudsinit)
Fin

```

(1)- Complexité de “noeudACompresser” (ligne 9)

La brindille / le nœud à compresser est celle dont “l’epsilon maximal” est minimal par rapport aux autres brindilles. En structurant notre liste de brindilles comme une sorte d’AVL (cf ci-dessous) on maintient le temps d’accès à ce minimum en logarithme du nombre de brindilles.

(2)- Complexité de “compressLambda” dans le cas particulier ci-dessus (ligne 10)

On effectue la compression lambda systématiquement sur une brindille (par construction de la liste des brindilles). Reprenons le pseudo code de la compression lambda : la conditionnelle en ligne 2 est alors toujours vraie : on exécute donc les lignes 3 et 4 (temps constant) et jamais les lignes 5 à 12, évitant les appels récursifs. Cette instruction s’exécute donc en temps constant.

Autres algorithmes utilisés par la compression rho

Lister les brindilles (cf. ligne 6 de “compressRho”)

Procédure : `listerBrindille(TwigList brindilles, ChildOf parent, QuadTree A)`

- Rôle : Recherche et les brindilles du quadtree et les stocker dans un dictionnaire (pour la structure de ce dictionnaire, voir [])
- Entrée : la liste de brindilles à compléter; le lien de parenté de l’arbre actuel; l’arbre dont on veut chercher les brindilles
- Complexité : $O(\text{nb_noeud} * \log(\text{nb_brindille}))$

Variable :

`ChildOf` nouvelleParente

Début

```

1 | Si non vide(A) et estZone(A) alors
2 | |     nouvelleParente ← filsDe(A, parent)
3 | |     Si estBrindille(A) alors
4 | | |     ajout(brindilles, nouvelleParente)           //cf note ci-dessous
5 | | |     Sinon
6 | | | |     listerBrindille(brindilles, nouvelleParente, nordOuest(A))
7 | | | |     listerBrindille(brindilles, nouvelleParente, nordEst(A))
8 | | | |     listerBrindille(brindilles, nouvelleParente, sudEst(A))
9 | | | |     listerBrindille(brindilles, nouvelleParente, sudOuest(A))
10| | |     Fin Si-Sinon
11| Fin Si
Fin
```

Complexité de “ajout” (ligne 4 ci-dessus) : Comme indiqué précédemment, on va vouloir structurer notre liste de brindilles comme une sorte d’AVL (le tri se fait suivant l’epsilon maximal de la brindille) pour un ajout en logarithme du nombre de brindilles.

Mise à jour de la liste de brindilles (cf. ligne 10 de “compressRho”)

Fonction : `update(TwigList brindilles) : Entier`

- Rôle : supprimer la brindille d’epsilon_{max} minimal, puis si son père dans le quadtree est une brindille l’ajouter à la liste ou fusionner les fils en cas d’égalité.
- Entrée : la liste de brindilles à mettre à jour
- Précondition : la brindille d’epsilon minimal **et uniquement celle-ci** doit avoir été compressée
- Complexité : $O(\text{hauteur}(\text{QuadTree}) + \log_2(\text{nb_brindille}))$

Variable :

Entier varNode //la variation du nombre de noeud dans le quadtree
ChildOf parentMin //ref. vers le père (dans le quadtree) de la brindille min

Début

```

1 | parentMin ← parent( extraireMin(brindilles) ) //0(log2(nb_brindilles))
2 | varNode ← 0
3 | Si ( non vide(parentMin) ) alors
  |   /* on remonte le long des pères du minimum jusqu'à ce que l'un d'entre
  |   * eux ne soit pas une brindille fusionnable 0(hauteur(QuadTree))*/
5 |   Tant que (non vide(parentMin) et fuseEqualSon(parentMin.noed) = 1)
  |   faire:
6 |       parentMin ← parent( parentMin ) ; varNode ← varNode - 4;
7 |   Fin Tant que
8 |
  |   //0(log2(nb_brindilles))
9 |   Si (non vide(parentMin) et estBrindille(parentMin.noed)) alors
10|       ajout(brindilles, parentMin) //cf note ci-dessus
11|   Fin Si
12| Fin Si
13| retourner varNode
Fin

```

Class Java et détail des structures de données utilisées

Classe QuadTree

La classe QuadTree permet de manipuler un quadtree région, soit une représentation d'une "surface colorée". Par soucis de simplicité, toute les surfaces de ce projet seront carrées de taille ($2^n \times 2^n$) avec "n" entier. Les différentes "teintes" ou "valeurs" de couleur seront représentées par des entiers.

Ainsi, un nœud d'un quadtree est lui même un quadtree (sous arbre du quadtree principal) :

- c'est une couleur si la région qu'il représente est monochrome (il n'a alors pas de fils)
- sinon c'est une sous-région incolore, avec au moins 2 fils non vides, sachant que la contrainte sur le dimensionnement des surfaces entraîne que les 4 fils soient toujours non vides.

Il est important de noter ici que cette structure vise à sauvegarder les données d'une image PGM (cf Classe PGM) dans une optique d'alléger l'empreinte mémoire de cette image. Ainsi :

- Le référentiel utilisé pour la surface sera paramétré avec un axe des abscisses orienté de gauche à droite, celui des ordonnées est orienté de haut en bas, et son point d'origine est le coin supérieur gauche de la surface considérée.
- Les coordonnées de la surface sont à sauvegarder si besoin en dehors du quadtree. Afin d'alléger au plus possible la structure, chaque nœud du quadtree ne contiendra que sa valeur de luminosité, ainsi que les 4 références vers ses fils immédiats.

Fonction : créer(Matrice<Entier> surface, Point origine, Entier longueur) : QuadTree, Entier

- Rôle : Construction, découpe la surface et crée récursivement le QuadTree en renvoyant le nombre de nœuds du quadtree créé.
- Entrée : la surface globale; les coordonnées de la sous-surface représentée par notre arbre (donnée par un point d'origine et une longueur de côté).
- Sortie : le quadtree nouvellement créé; le nombre de noeuds dans le quadtree

Fonction : créer(Matrice<Entier> surface) : QuadTree, Entier

- Rôle : Construction, initialise la découpe de la surface et crée récursivement le QuadTree tout en renvoyant le nombre de nœuds en fin de construction.
- Entrée : La surface à représenter
- Sortie : le quadtree nouvellement créé; le nombre de noeuds dans le quadtree

Procédure : versSurface(QuadTree A, Matrice<Entier> surface, Point origine, Entier longueur)

- Rôle : complète la surface allouée au préalable en parcourant récursivement le quadtree
- Entrée : la surface allouée incomplète; les coordonnées de la sous-surface correspondant au quadtree (donnée par un point d'origine et une longueur de côté).
- Post-condition : la partie de la matrice de surface correspondant aux coordonnées doit être entièrement remplie avec les données correspondantes du quadtree.

Fonction : versSurface(QuadTree A, Entier longueur) : Matrice<Entier>

- Rôle : Converti le quadtree en surface en allouant celle-ci selon la longueur de son côté puis appelle la procédure récursive, ci-dessus, afin de remplir la surface.
- Entrée : la dimension de la surface à allouer (la longueur du côté suffit, la surface étant carrée).
- Sortie : la surface correspondant au quadtree.

Fonction : fuseEqualSon(QuadTree A) : Entier

- Rôle : vérifie si les fils sont monochrome, et de la même teinte, et le cas échéant, fusionne la brindille.
- Sortie : renvoie 1 si les fils ont effectivement été fusionnés, 0 sinon.

Fonction : estSurface(QuadTree), estMonochrome(QuadTree), estBrindille(QuadTree) : Booleen

- Rôle : chacune de ces fonctions indique la nature du quadtree et renvoie vraie si : deux des fils sont non vides (estSurface) / s'ils sont tous vides (estMonochrome) / s'ils sont tous monochrome (estBrindille)

Classe PGM

La classe PGM permet de manipuler des fichiers PGM P2 à l'aide d'un QuadTree. Il contient le QuadTree ainsi qu'un champ qui gardera le nombre de nœuds actuels dans celui-ci. Il gardera le contenu essentiel d'un fichier pgm P2 : la luminosité maximum et la taille dans notre cas, une puissance de 2.

Fonction : PGM(Chaîne de Caractères pathfile) → PGM

- Rôle : Constructeur, crée une nouvelle instance de PGM selon un fichier donnée
- Entrée : pathfile Chaîne de caractères étant le chemin relatif vers un fichier PGM d'une taille $2^n \times 2^n$
- Sortie : PGM le nouvel objet associé à ce fichier

Procédure : toPGM(PGM objet, Chaîne de Caractères pathname)

- Rôle : Crée un nouveau fichier PGM selon l'objet PGM actuel si le fichier n'existe pas déjà
- Entrée : PGM la représentation objet du fichier, pathname une chaîne de caractère représentant le chemin relatif avec le nom du nouveau fichier y compris

Procédure : compressLambda(PGM obj)

- Rôle : Effectue un effeuillage des brindilles de l'arbre associé au PGM selon la méthode lamda
- Entrée : PGM la représentation objet du fichier
- Sortie : Un PGM pour lequel son arbre a été compressé

Procédure : compressRho(PGM obj, Entier rho)

- Rôle : Comprime l'arbre associé au PGM
- Entrée : rho un entier entre 0 et 100 inclus, le facteur de nœud qu'il doit rester. nbCourant/nbInit
- Sortie : Un PGM pour lequel son arbre a été compressé

Fonction : toString(PGM obj) → Chaîne de Caractere

- Rôle : Retourne la chaîne de caractère associé à l'objet PGM
- Entrée : un PGM
- Sortie : Chaîne de caractère qui représente l'objet PGM

Classe TwigList

La classe TwigList se chargera de manipuler la liste des brindilles sous la forme d'un AVL. Elle effectuera dans un premier la recherche de toutes les brindilles en mappant le chemin de ce nœud jusqu'à la racine du QuadTree avec des objets ChildOf. Elle contiendra un AVL qui triera les brindilles en fonction de leur valeur epsilon. Lorsqu'un nœud sera compressé, on ira fusionner ses ancêtres si nécessaire jusqu'à que l'on ne puisse plus ou que l'on tombe sur une brindille au quel cas, on l'ajoutera dans l'AVL qui contient les brindilles.

Fonction : TwigList(QuadTree A) : TwigList

- Rôle : Constructeur, crée une nouvelle instance de TwigList selon le quadtree donnée
- Entrée : A, un QuadTree que l'on va compresser.
- Sortie : une nouvelle instance de TwigList avec la liste des brindilles établie et le mappage des parents des nœuds.

Procédure : fillAllTwigs(AVL<ChildOf> listeBrindille, ChildOf parent, QuadTree noeud)

- Rôle : Remplit l'AVL contenant les brindilles triés et garde un mappage de tree avec son parent.
- Entrée : listeBrindille, l'AVL contenant les brindilles triés selon leur valeur epsilon; parent la pair qui associe le père du nœud avec leurs ancêtres; noeud un QuadTree le nœud courant étant le fils de parent.
- Sortie : listeBrindille trié et rempli

Procédure : addSorted(AVL<ChildOf> listeBrindille, ChildOf paire)

- Rôle : Ajoute une paire brindille-parent dans l'AVL des brindilles de façon triés selon sa valeur epsilon
- Entrée : listeBrindille l'AVL trié selon la valeur epsilon des brindilles; pair, la paire à ajouter
- Sortie : listeBrindille trié et mise à jour

Fonction: update(AVL<ChildOf> listeBrindille) : Entier

- Rôle : Retire la première brindille qui doit être compressée (c'est-à-dire la brindille avec la valeur epsilon epsilon) et ajoute si possible le premier de ses ancêtres étant une brindille.
- Entrée : listeBrindille, l'AVL de ChildOf trié selon la valeur epsilon des brindilles à modifier
- Sortie : un entier étant la variation du nombre de nœuds de l'arbre si en cherchant un ancêtre de la brindille, on a dû fusionner ses fils égaux afin de maintenir un QuadTree valide.

Classe ChildOf

La classe ChildOf permet d'associer un nœud du QuadTree avec ses ancêtres. Dans la cas de la racine du QuadTree initial, sa paire d'ancêtre sera vide. On peut visualiser cela par un arbre dont on ne pointe pas vers les fils mais vers leur parent unique.

Fonction : ChildOf(QuadTree noeud, ChildOf parent) : ChildOf

- Rôle : Constructeur, crée une nouvelle instance de ChildOf, lie un noeud avec ses ancêtres
- Entrée : noeud le QuadTree courant; parent une paire ChildOf qui relie le père du noeud avec leurs ancêtres
- Sortie : une nouvelle instance de ChildOf tel que parent est la paire ChildOf qui contient le père et les ancêtres du nœud et le fils le nœud courant.

Classe AVL

La classe AVL utilise la définition du cours où l'élément est un réel et où on y stockera également des Files de brindilles. Chaque nœud de notre AVL sera donc défini par un réel (epsilon), une File(ChildOf) correspondant aux brindilles associées de leurs ancêtres possédant cette valeur epsilon, et les deux fils de l'AVL. On généralise l'implémentation de l'AVL dans le code. On utilisera la librairie LinkedList qui est une liste doublement chaînée avec un accès à la tête et à la queue constant.

Fonction : **AVL(Réel epsilon, ChildOf pair) : AVL<ChildOf>**

- Rôle : Constructeur, crée une nouvelle instance de AVL
- Entrée : pair ChildOf la brindille associé à ses ancêtres; epsilon un Réel, la valeur epsilon de cette brindille
- Sortie : une nouvelle instance d'AVL avec comme valeur epsilon et pair ajoutée dans la File de ce nœud.

Fonction : **minExtractAVL(AVL<ChildOf> obj) : ChildOf**

- Rôle : Recherche, retourne et supprime la première valeur de la File<ChildOf> du noeud minimum de l'AVL
- Sortie : ChildOf la brindille associé à ses ancêtres; AVL la nouvelle racine de l'AVL

Procédure : **add(AVL<ChildOf> obj, Réel epsilon, ChildOf pair)**

- Rôle : Ajoute un nouveau noeud dans l'AVL si la valeur epsilon n'existe pas, sinon ajoute à la fin de la File du noeud qui a pour élément epsilon
- Entrée : ChildOf la brindille associé à ses ancêtres; Réel la valeur epsilon de cette brindille

Procédure : **rotationDroite(AVL<ChildOf> obj)**

- Rôle : Effectue une rotation à droite à la racine de l'AVL sans changer l'adresse de la racine.
- Entrée : AVL la racine de l'AVL entier
- Sortie : L'AVL modifié tel que la balance soit entre $-1 \leq \text{bal} \leq 1$

Procédure : **rotationGauche(AVL<ChildOf> obj)**

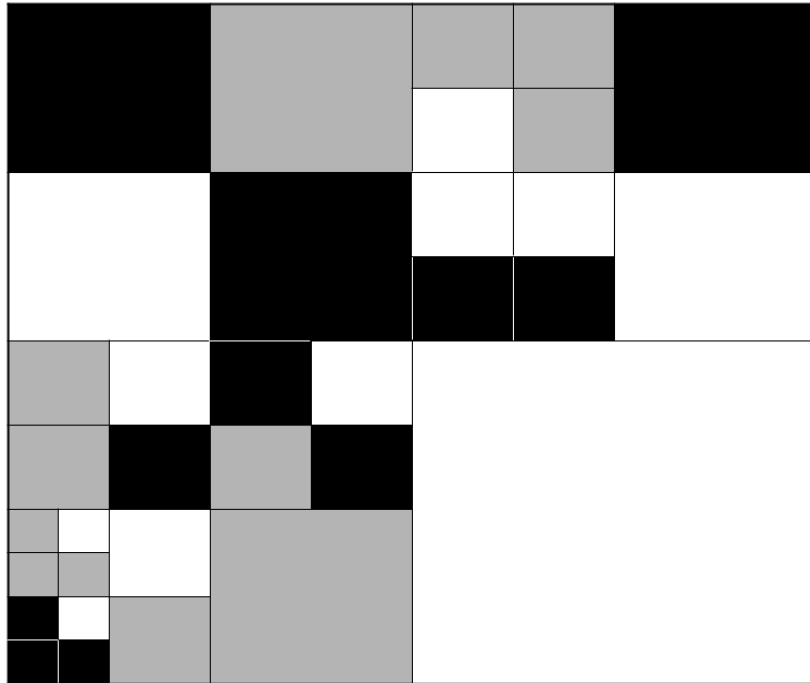
- Rôle : Effectue une rotation à gauche à la racine de l'AVL sans changer l'adresse de la racine.
- Entrée : AVL la racine de l'AVL entier
- Sortie : L'AVL modifié tel que la balance soit entre $-1 \leq \text{bal} \leq 1$

Procédure : **équilibrer(AVL<ChildOf> obj)**

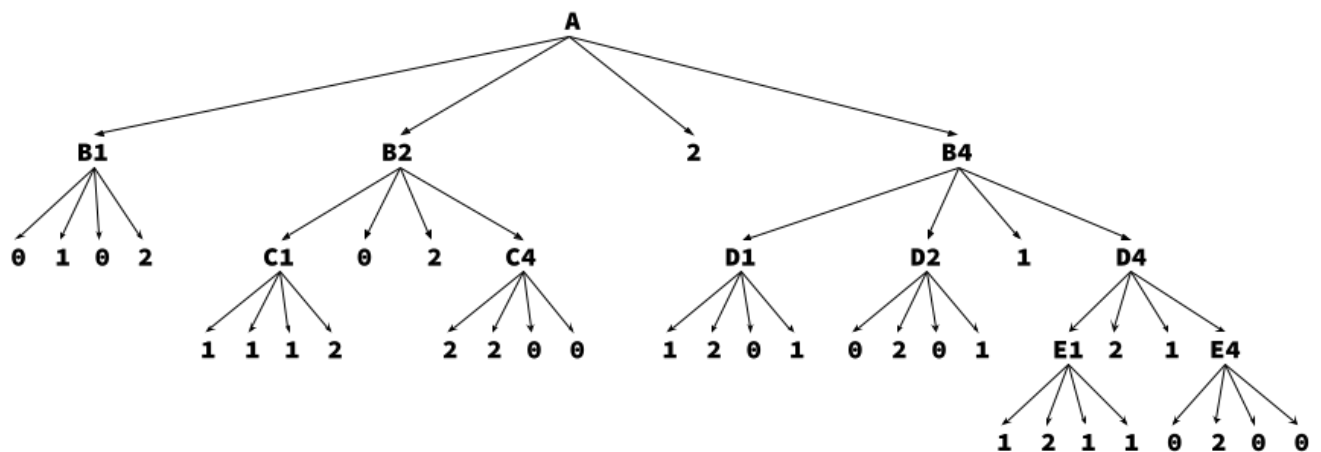
- Rôle : Effectue les rotations nécessaires afin d'équilibrer l'AVL
- Entrée : AVL la racine de l'AVL entier
- Sortie : L'AVL modifié tel que la balance soit entre $-1 \leq \text{bal} \leq 1$

Schématisation des structures de données

1 - Surface exemple avec 3 couleurs :
0 = noir, 1 = gris et 2 = blanc

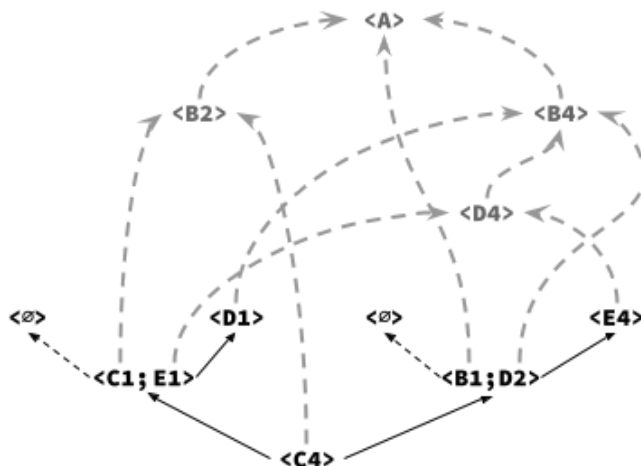


2 - QuadTree issu de la surface exemple



3 - TwigList issue du QuadTree

- AVL des brindilles (lien en noir) renversé par soucis de schématisation : racine <C4>.
- Les liens en gris représentent les référencement des pères dans le quadtree (construit via la classe ChidlOf).



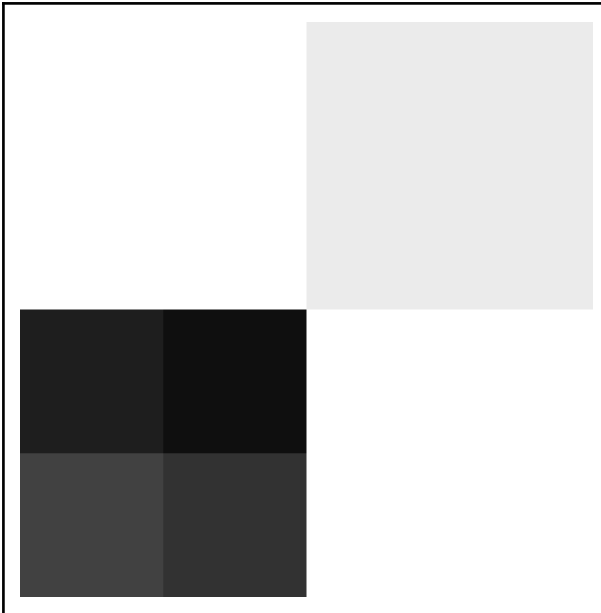
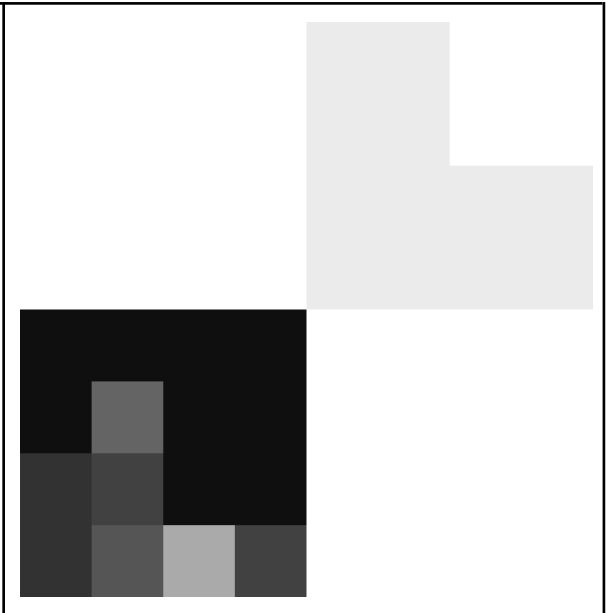
Epsilon des brindilles

Bri	ϵ
B1	1.61
C1	0.71
C4	1.54
D1	1.29
D2	1.61
E1	0.71
E4	1.79



Exemple de résultats d'executions:

<p>Nombre initial de noeuds: 1297713 Nombre Rho de noeuds : 129769 (nbCourant/nbInit)% : 9.999822765126034 % 907 millisecondes</p>	<p>Nombre initial de noeuds : 1297713 Nombre Lambda de noeuds : 332569 (nbCourant/nbInit)% : 25.62731513054119 % 47 millisecondes</p>

Exécution du Main avec "tree_big.pgm" (1024x1024) avec rho = 10

	
<div>Nombre initial de noeuds : 49 Nombre Rho de noeuds : 9 (nbCourant/nblnit)%:18.367346938775512 % 1 milliseconde</div>	<div>Nombre initial de noeuds : 49 Nombre Lambda de noeuds : 25 (nbCourant/nblnit)%: 51.02040816326531 % 0 milliseconde</div>

Exécution du Main avec "eval.pgm" (16x16) avec rho=25

	
<div>Nombre initial de noeuds : 4081849 Nombre Rho de noeuds : 408181 (nbCourant/nblnit)% : 9.999904455064359 %</div>	<div>Nombre initial de noeuds : 4081849 Nombre Lambda de noeuds : 1110321 (nbCourant/nblnit)%: 27.20142268859039 %</div>

Exécution du Main avec "mathieu_odin-unsplash.png" => "mathieu_odin-unsplash.pgm" (2048x2048) avec rho=10