Quadtree.

Exemple de contrôle sur machine – durée 1h30

NOTES : éléments de notation

- la bonne utilisation du pouvoir expressif du langage C sera prise en compte.
- la présentation et la lisibilité du code écrit seront prises en compte.

Extension d'un Type Abstrait de Données

L'adaptation d'un type abstrait de données aux besoins d'une application est réalisée grâce à l'*extension* de ce TAD et de son implantation.
L'*extension* du TAD consiste en la définition d'un nouvel invariant de structure venant étendre l'invariant initial du TAD.
L'*extension* de son implantation consiste en la programmation efficace de la représentation concrète du TAD et de ses opérateurs.

Ce contrôle a pour objectif l'extension d'un TAD à des besoins applicatifs spécifiques.

1 Contexte

Les arbres binaires de recherche, et leurs variantes, définissent un type abstrait de données très efficace pour la réalisation des opérations de dictionnaire, ADD, REMOVE et SEARCH qui peuvent toutes être réalisées par des opérateurs de complexité dans le pire cas en $O(log(n))$ pour un arbre contenant n clés.
Lorsque l'on souhaite gérer des données dont les clés sont de dimension 2, comme des points dans le plan cartésien $2D$, la notion d'arbre binaire peut être étendue à un arbre quaternaire ou `QuadTree`.
Un `QuadTree` est un arbre de recherche dans lequel chaque nœud possède 4 fils, notés *upleft*, *upright*, *downleft* et *downright*. Ces 4 fils permettent de diviser chaque dimension en 2 (*left* et *right* pour la 1ère dimension, *down* et *up* pour la seconde dimension. Un tel arbre permet de définir une partition **irrégulière**, **adaptative** du plan $2D$ dans laquelle les données sont associées uniquement aux feuilles.
Cette structure de données a été proposée initialement par R. Finkel et J.L. Bentley en 1974 et possède les propriétés suivantes :

- Un `QuadTree` décompose l'espace en cellules d'étendue spatiale variable.
- Chaque cellule possède une capacité (un nombre de points) maximale identique. Lorsque cette capacité est dépassée, la cellule est subdivisée en 4 et les points sont répartis dans chaque sous-cellule.
- Les opérations d'insertion, de recherche et de suppression d'un point dans un `QuadTree` ont une complexité en temps de l'ordre de $O(log_4(n)) \sim O(log(n))$
- La structure de l'arbre suit la décomposition spatiale du plan.

La figure 1 montre l'organisation spatiale et la topologie de l'arbre associé pour un `QuadTree` de capacité 1.
Nous nous intéressons, dans ce contrôle, à l'implantation du TAD `QuadTree` sous forme d'une extension d'une structure d'arbre binaire permettant de respecter les propriétés des `QuadTree` énoncées ci-dessus.
À partir du code fourni dans l'archive `controleTP.tar.gz` associée à ce sujet et construite sur le même modèle que les archives de TP, l'objectif de ce contrôle est de définir la représentation d'un quadtree ainsi que les opérateurs principaux pour sa construction et son parcours.

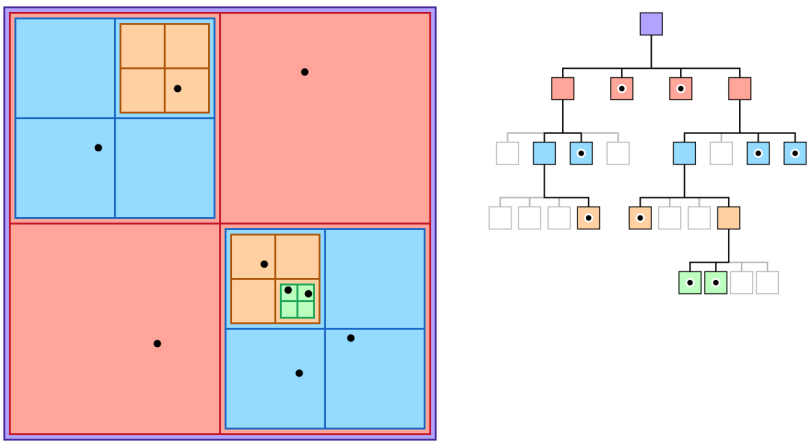

FIGURE 1 – Exemple d'un quatree de capacité 1.

1.1 Description du code fourni

Le code fourni contient :

- Un sous répertoire `Code` contenant :
  - un fichier Makefile permettant de compiler l'application.
  - un fichier `main.c` proposant un programme de test de l'implantation,
  - un fichier `quadtree.h` définissant l'interface publique d'un `QuadTree`, interface très similaire à celle des arbres binaires de recherche vus en TP,
  - un fichier `quadtree.c` définissant l'implantation et l'interface privée du TAD et à compléter pour ce contrôle.
- Un répertoire `Test` contenant des fichiers de test.

Le code fourni ne devra en aucun cas être modifié. Seul devra être complété le fichier `quadtree.c` en dessous du cartouche à compléter par votre nom, prénom et numéro d'étudiant.

```
/*************************************************************/
/**                      Control start here                 **/
/*************************************************************/
/**
 *  Nom          :
 *  Prenom       :
 *  Num Etud     :
 **/
```

1.2 Description des algorithmes à mettre en place.

1.2.1 Subdivision d'une feuille

Un nœud de l'arbre (structure `struct _treeNode` dans le fichier `quadtree.c`) peut représenter un nœud interne ou une feuille de l'arbre.
Un nœud de l'arbre possède une étendue spatiale définie par les coordonnées du point inférieur gauche du nœud (`Point min`) et du point supérieur droit (`Point max`).
Un nœud interne possède des pointeurs vers ses 4 fils rassemblés dans le champs `data.childs`.
Une feuille possède l'ensemble des points recouverts par son étendue spatiale dans un tableau dynamique accessible dans le champs `data.pointset`.
Il est à noter que, comme ces deux informations sont exclusives, elles sont rassemblées dans le champ `data` sous forme d'une union en langage C. On ne peut donc accéder qu'à l'une ou l'autre des informations selon

le type du nœud (interne ou feuille).

L'algorithme de subdivision d'une feuille est le suivant :

1. Calcul du centre de l'étendue spatiale de la feuille : `splitpos.x = (min.x+max.x)/2` et `splitpos.y = (min.y+max.y)/2`.
2. A partir du centre, création de 4 nouvelles feuilles dont les étendues spatiales sont calculées à partir des points `min`, `max` et `splitpos` (vous aider d'un dessin).
3. Pour chaque point `p` de la feuille, ajout du point dans sa nouvelle feuille d'appartenance en comparant sa position au point central `splitpos` :
   — si `p.x<splitpos.x` le point sera ajouté dans une des feuilles de gauche. Sinon, il sera ajouté dans une des feuilles de droite.
   — si `p.y<splitpos.y` le point sera ajouté dans une des feuilles du bas. Sinon, il sera ajouté dans une des feuilles du haut.
4. Destruction du tableau de points de l'ancienne feuille qui devient un nœud interne.

1.2.2 Ajout d'un point dans un quadtree.

La fonction `void quadtree_add(QuadTree *t, Point p)`, a pour objectif d'insérer la valeur p dans l'arbre t en respectant l'invariant des `QuadTree`.

Invariant des `QuadTree`

Dans un `QuadTree`, un point est stocké sur une feuille dont l'étendue spatiale recouvre la donnée. Toutes les feuilles possèdent la même capacité maximale de stockage qui ne peut pas être dépassée.

L'insertion d'un point dans un `QuadTree` commence donc par la recherche de la feuille dans laquelle le point doit être inséré. Lors de cette recherche, si le nœud n'est pas une feuille, on pourra récupérer sa position de découpe par la fonction `Point node_splitpos(const Node * const n)` et choisir ainsi dans quel fils doit se poursuivre la recherche.
Lorsque la feuille d'insertion est trouvée, il faut alors vérifier qu'elle n'a pas atteint sa capacité maximale avant d'ajouter le point à la feuille (fonction `void node_add_point(Node *n, Point p)`. Si la capacité maximale a été atteinte, il faut alors subdiviser la feuille en 4 (fonction `void node_subdivide(Node *n)`) avant de choisir la nouvelle feuille d'insertion.
Il est à noter que cette subdivision doit se faire récursivement tant que la feuille d'insertion a atteint sa capacité maximale.

2 Travail à réaliser

2.1 Définition de la représentation interne d'un `QuadTree`

1. En prenant soin de ne stocker que les informations nécessaires et en utilisant l'implantation fournie pour la gestion des nœuds (le TAD `Node`), écrire la représentation interne du TAD `QuadTree` dans la structure `struct _quadtree`.
2. Programmer les fonctions `QuadTree * quadtree_create(int npoints, Point min, Point max)`, construisant un `QuadTree` dont l'étendue spatiale est définie par les points `min` et `max` et dont la capacité des feuilles est `npoints`, et `bool quadtree_empty(const QuadTree *t)` retournant vrai si un `QuadTree` est vide.

2.2 Opérateur d'ajout d'un élément dans un `QuadTree`

Sans vérifier la capacité de la feuille d'insertion ni effectuer la subdivision éventuelle de cette feuille, programmer l'opérateur `void quadtree_add(QuadTree *t, Point p)` d'ajout du point `p` dans l'arbre `t`.

2.3 Opérateur de parcours préfixes de l'arbre

Programmer, de façon récursive, l'opérateur `void node_depth_prefix(const Node *n, OperateFunctor f, void *userData)` effectuant un parcours préfixe en profondeur d'abord à partir du nœud `n`.
En utilisant la fonction précédente, programmer l'opérateur `void quadtree_depth_prefix(const QuadTree *t, OperateFunctor f, void *userData)` de parcours du `QuadTree t`.
Pour vérifier ces 3 premières questions, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./quadtree_test ../Test/testfilesimple.txt`.
Vous pouvez produire le fichier pdf de visualisation de l'arbre que vous avez créé en tapant `$make pdf`. Si vous ouvrez le fichier pdf `testfilesimple.dot.pdf` contenu dans le répertoire `Test`, vous obtiendrez un arbre réduit à la racine, qui est une feuille possédant 6 points.

2.4 Programmer l'établissement de l'invariant des quadtree

Programmer la fonction `void node_subdivide(Node *n)` qui subdivise la feuille `n` en 4 nouvelles feuilles selon l'algorithme décrit précédemment.
Après avoir programmé cette fonction, modifiez votre fonction `void quadtree_add(QuadTree *t, Point p)` pour insérer un point en conservant l'invariant.
Pour vérifier la bonne construction du quadtree et de vos invariants de structure, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./quadtree_test ../Test/testfilesimple.txt`.
Le fichier pdf produit par `$make pdf` correspond à l'arbre de la figure 2.

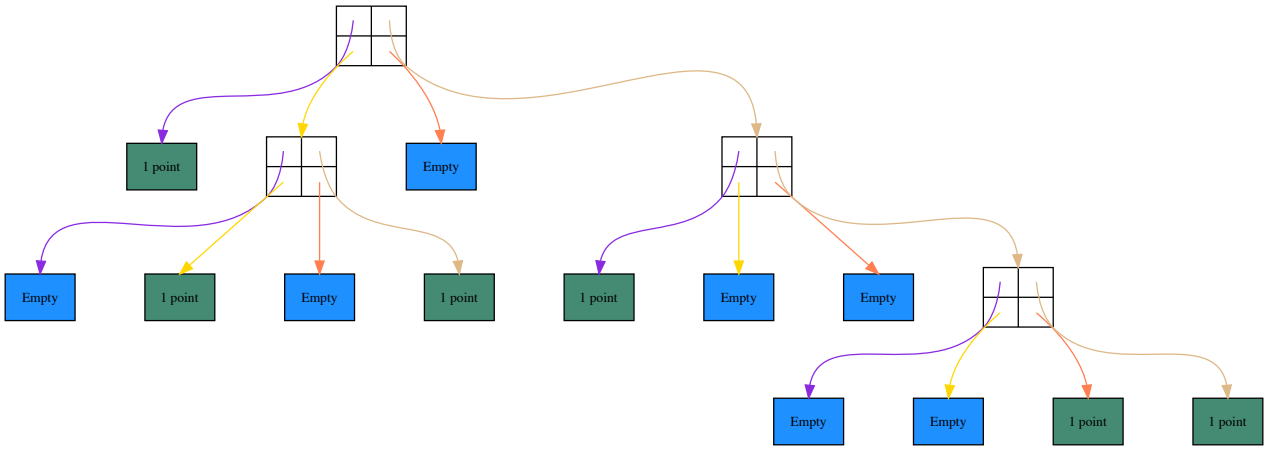

FIGURE 2 – Arbre construit avec le ficher de données testfilesimple.txt.