Institut Galilée - Université Paris 13

Structure de Données Avancées
Rapport

Travaux Pratiques - TP 04

Étudiants :
Tiziri OULD HADDA

Enseignants :
M. Olivier BODINI
M. Julien DAVID
M. Sergey DOVGAL

30 décembre 2019

Table des matières

1 Structures de données de données

— Lien des sources de la structure b-tree utilisée : Lien github
— Lien des sources de la structure avl utilisée : Lien github

1.1 B-Arbres

L'implémentation trouvée dans les sources sur github, est basée sur le livre "Introduction à l'algorithmique" référence en la matière dans les structures de données et des algorithmes. On retrouve dans le code téléchargé, les même noms des fonctions décrite dans le livre et les même algorithmes.

Le B-arbre est modélisé en mémoire par la structure de donnée BTree ci dessous ou :
— order : entier désignant l'ordre t de l'arbre
— root : pointeur vers la racine du noeud.

A chaque fois qu'on souhaite ajouter un nouvel élement au B-Arbre on crée un noeud du type node_t :
— is_leaf : boolean, indique si le noeud est une feuille ou pas.
— n_keys : entier, déduit depuis l'ordre et le type du noeud (racine ou pas).
— children : tableau de poiteurs vers des noeuds node_t : Alloué dynamiquent en fonction de n_keys (tableau de n_keys + 1 élements de type node_t*).
— keys : : tableau de poiteurs vers des clés de type pair_t, il est alloué dynamiquent en fonction de n_keys (tableau de n_keys élements de type pair_t*), crée pour faire une genéralisation de type de clés (dans ce cas on peut ajouter à la fois une clé et une valeur.

Vu que le nombre de clés et d'enfants dépendent de l'ordre, il est donc le seul paramètre utilisé à la création du B-Arbre. C'est ce paramètre qui a plus d'influence sur le scindage et la fusion.

```
typedef struct pair_t {
int key;
void *value;
} pair_t;

typedef struct node_t {
bool is_leaf;
int n_keys;
struct node_t **children;
pair_t **keys;
} node_t;

typedef struct btree_t {
int order;
node_t *root;
} BTree;
```

1.2 AVL

L'arbre AVL est avant tout un arbre binaire de recherche qui reste tout le temps équilibré.

- Dans la structure on met souvent un pointeur sur le sous arbre gauche et un autre vers le sous arbre droit afin de permettre le parcours de tout les élements de l'arbre depuis la racine.
- Le champs key contient la clé à insérer dans l'AVL.
- Le champs height permet de stocker la hauteur de chaque noeud. Ce paramêtre est mis à jour à chaque insertion, suppression ou rotation. Il permet de ne pas parcourir l'arbre à chaque fois pour verifier le facteur d'équilibre.

```
struct AVLnode
{
    int key;
    struct AVLnode *left;
    struct AVLnode *right;
    int height;
};
typedef struct AVLnode avlNode;
```

L'insertion dans un AVL, est réalisée dans le source utilisé comme une insertion dans un arbre binaire de recherche sauf que à chaque fois l'algorithme regarde si l'arbre est équilibré afin d'éffectuer une rotation selon le cas qui se présente (rotation gauche, droite, gauche droite, droite gauche).

Condition de rotation : Les hauteurs des deux sous-arbres d'un même noeud diffèrent au plus de un. le code suivant résume tous ces cas :

```
avlNode *insert(avlNode *node, int key)
{
  if(node == NULL)
    return (newNode(key));
  /*Binary Search Tree insertion*/
  if(key < node->key)
    node->left = insert(node->left, key);/*Recursive insertion in L subtree*/
  else if(key > node->key)
    node->right = insert(node->right, key);  /*Recursive insertion in R subtree*/

  /* Node  Height as per the AVL formula*/
  node->height = (max(nodeHeight(node->left), nodeHeight(node->right)) + 1);
  /*Checking for the balance condition*/
  int balance = heightDiff(node);

  /*Left Left */
  if(balance>1 && key < (node->left->key))
    return rightRotate(node);
  /*Right Right */
```

```
  if(balance<-1 && key > (node->right->key))
    return leftRotate(node);
  /*Left Right */
  if (balance>1 && key > (node->left->key))
    node = LeftRightRotate(node);
  /*Right Left */
  if (balance<-1 && key < (node->right->key))
    node = RightLeftRotate(node);

  return node;
}
```

La suppression suit le même procédé que l'insertion.

2 Analyse des expériences réalisées

Le cout amorti du B-Arbre lorsque les cles insérés sont croissantes est beaucoup plus petit par contre lorsque les clès sont aléatoires ou décroissantes cout amorti de l'algorithme utilisé monte grandement. Cela est peut être dû à la compléxité de cette structure. Quant au code de l'AVL, le cout amorti, dans tout les cas réalisés, reste très apréciable et semble ne pas beaucoup varier.

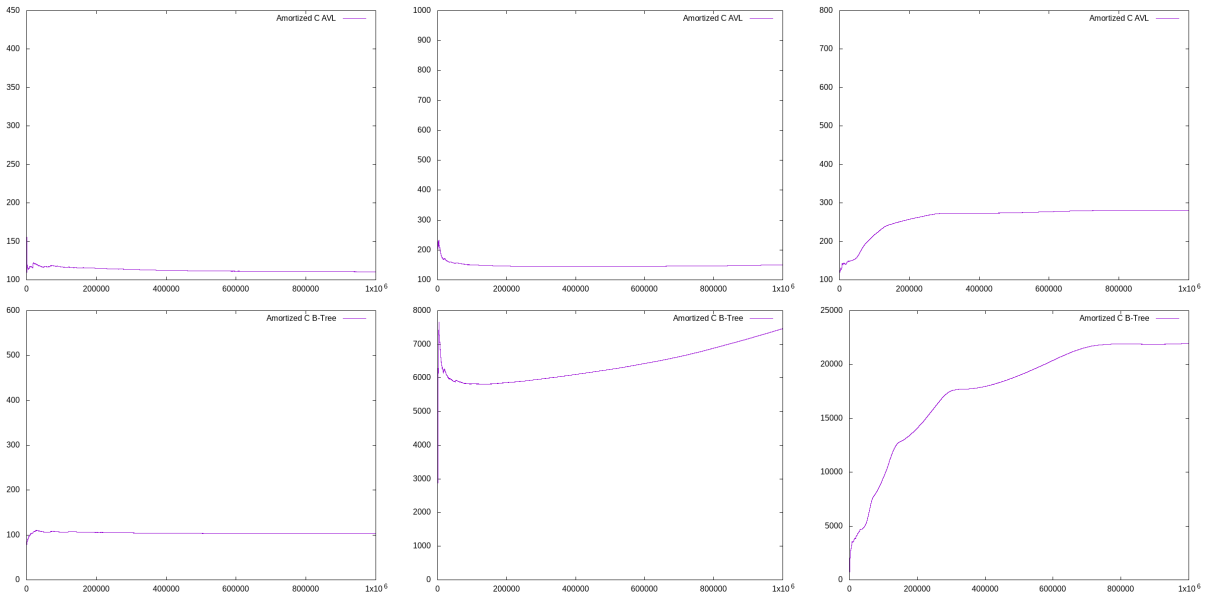

FIGURE 1 – Cout amorti - Ajout sur un AVL/Btree(ordre 1000),, ordre de valeurs croissantes(haut)/décroissantes(milieu)/aleatoires(bas)

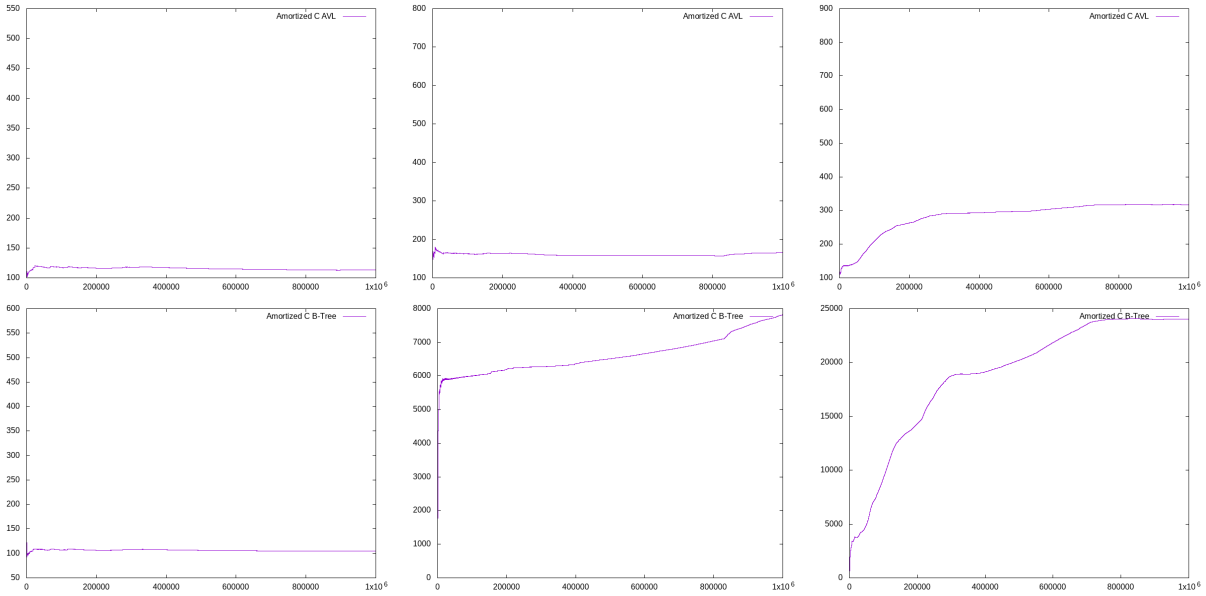

FIGURE 2 – Cout amorti - Ajout et suppression sur un AVL/Btree(ordre 1000), ordre de valeurs croissantes(haut)/décroissantes(milieu)/aleatoires(bas)

3 Choix de l'ordre t du B-Arbre

La figure 3 montre certains tests effectués pour differents ordres du B-Arbre et qui semblent toutes déssiner une courbe quadratique avec un minimum à l'ordre 8. D'autres tests, ont été effectués avec différentes valeurs pour confirmé cette hypothèse. Et on remarque aussi que plus l'ordre est supérieur à la valeur optimal plus le cout amorti augmente rapidement.

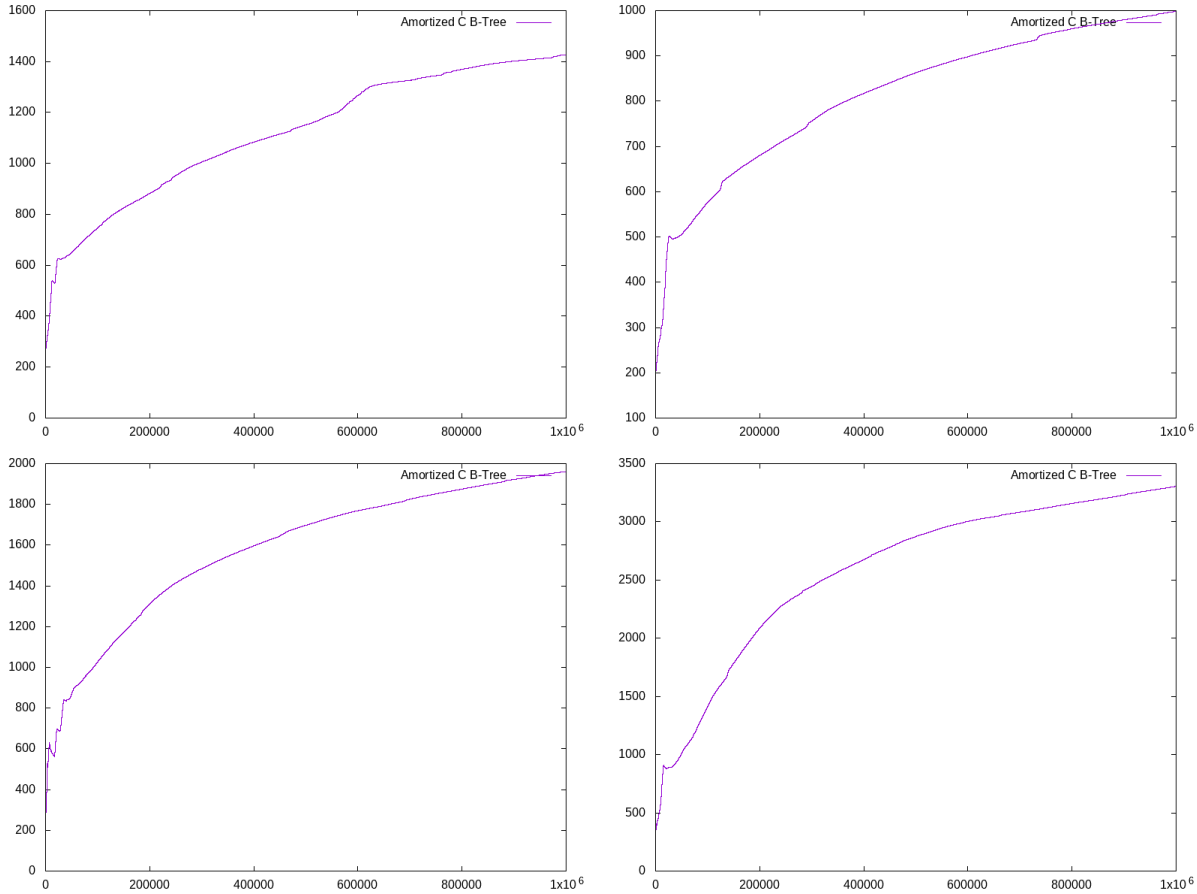

FIGURE 3 – Cout amorti du B-Arbre, insertion et suppression de clés aléatoires pour ordre = 4, 8, 64 et 1024 (de gauche à droite et de haut en bas)

4 Conclusion

D'après les expériences réalisés, il semble que le cout amorti de l'AVL est beaucoup plus petit car, en effet, on peut voir le B Arbre comme étant un arbre equilibré (sorte AVL) mais plus complèxes, ou on aurait beaucoup de clés à gérer par noeud. Les arbres B sont souvent utilisés dans les systèmes de fichiers et les bases de données afin minimiser le nombre de recherches requises par une opération de lecture ou d'écriture.