

RAPPORT STRUCTURE DE DONNEES AVANCEES

TP4 B-ARBRE

Réalisé par :

LE Minh Hao

KEITA Aïssata

Table des matières

| | |
|---|----------|
| 1. Structure/classe de B-arbre | 3 |
| 2. Expliquez brièvement les choix d'implémentation qui ont été fait : | 4 |
| 3. Développez vous-même, ou cherchez sur internet, une structure/classe d'arbre AVL offrant les opérations ci-dessus. | 5 |
| 3. Effectuez des expériences pour comparer l'efficacité en temps et en mémoire des B -arbres et des ABRs..... | 6 |

1. Structure/classe de B-arbre

Développez vous-même, ou cherchez sur internet, une structure/classe de B-arbre offrant au moins les opérations suivantes : la création, la recherche d'une clé, l'insertion d'une clé, la suppression d'une clé.

Nous trouve sur Internet, une structure de B-arbre offrant les opérations nécessaires, dans la source - <https://www.codeproject.com/Articles/1158559/B-Tree-Another-Implementation-By-Java>

Ce sont les propriétés de ce B-arbre

```
7 /**
8  * Class BTree
9  * @author tnguyen
10 * Description: BTree implementation
11 */
12 public class BTree<K extends Comparable, V>
13 {
14     public final static int     REBALANCE_FOR_LEAF_NODE        = 1;
15     public final static int     REBALANCE_FOR_INTERNAL_NODE    = 2;
16
17     private BTNode<K, V> mRoot = null;
18     private long         mSize = 0L;
19     private BTNode<K, V> mIntermediateInternalNode = null;
20     private int mNodeIdx = 0;
21     private final Stack<StackInfo> mStackTracer = new Stack<StackInfo>();
22
23 }
```

Ce sont les méthodes de ce B-arbre:

```
BTree<K extends Comparable, V>
  REBALANCE_FOR_LEAF_NODE: int
  REBALANCE_FOR_INTERNAL_NODE: int
  mRoot: BTNode<K, V>
  mSize: long
  mIntermediateInternalNode: BTNode<K, V>
  mNodeIdx: int
  mStackTracer: Stack<StackInfo>
  getRootNode(): BTNode<K, V>
  size(): long
  clear(): void
  createNode(): BTNode<K, V>
  search(K): V
  insert(K, V): BTree
  delete(K): V
```

Tous les codes sont téléchargés sur GitHub dans le lien :
https://github.com/LEMinhHao/paris13_sda

2. Expliquez brièvement les choix d'implémentation qui ont été fait :

La représentation d'un noeud de l'arbre, la liste de ses clés et celle de ses enfants ; ainsi que l'effet que cela a sur les opérations de fusion et de scindage des noeuds.

```
50 /**
6  * Class BTreeNode
7  * @author tnguyen
8  */
9  public class BTreeNode<K extends Comparable, V>
10 {
11     public final static int MIN_DEGREE = 5;
12     public final static int LOWER_BOUND_KEYNUM = MIN_DEGREE - 1;
13     public final static int UPPER_BOUND_KEYNUM = (MIN_DEGREE * 2) - 1;
14
15     protected boolean mIsLeaf;
16     protected int mCurrentKeyNum;
17     protected BTreeNodeValue<K, V> mKeys[];
18     protected BTreeNode<K, V> mChildren[];
19 }
```

La classe **BTreeNode** comporte les propriétés comme suivant :

- Un nombre entier **MIN_DEGREE** stocke la valeur fixe d'ordre de B-tree. Nous va le changer dans la question prochaine.
- Une table fixe **mKeys[]** stocke des clés (en nombre entier) de ce noeud. On utilise une table fixe puisque le b-arbre a un nombre fixe de clés (de $t-1$ à $2*t-1$). Ou on peut utiliser une liste chaîné pour facilement faire des rotations ou des rétrécissements (dans les opérations de fusion et de scindage des noeuds). Puisque on peut faire plusieurs rotations dans un b-arbre avec la suppression d'une clé.

```
z.count = order - 1; // this is updated size

for (int j = 0; j < order - 1; j++) {
    z.key[j] = oldNode.key[j + order]; // copy right childs of oldNode (child) into front of newNode (parent)
}

if (!oldNode.isLeaf) // if not leaf we have to reassign child nodes.
{
    for (int k = 0; k < order; k++) {
        z.child[k] = oldNode.child[k + order]; // reassigning right childs of oldNode (child)
    }
}

// assigner les enfants des noeuds droites
for (int j = newNode.count; j > 0; j--) // if we push key into newNode (parent) we have
{ // to rearrange child nodes
    newNode.child[j + 1] = newNode.child[j]; // shift children of x
}
```

- Un nombre entier **mCurrentKeyNum** stocke le nombre courant d'élément dans un noeud.
- Une table fixe **mChildren[]** stocke des enfants d'un noeud.
- Un booléen **mIsLeaf** vérifie que ce noeud est un feuille ou pas

La classe BTree comporte les propriete comme suivant :

```
7 /**
8  * Class BTree
9  * @author tnguyen
10 * Description: BTree implementation
11 */
12 public class BTree<K extends Comparable, V>
13 {
14     public final static int     REBALANCE_FOR_LEAF_NODE        = 1;
15     public final static int     REBALANCE_FOR_INTERNAL_NODE    = 2;
16
17     private BTNode<K, V> mRoot = null;
18     private long mSize = 0L;
19     private BTNode<K, V> mIntermediateInternalNode = null;
20     private int mNodeIdx = 0;
21     private final Stack<StackInfo> mStackTracer = new Stack<StackInfo>();
22
23 }
```

- Les nombres entiers REBALANCE_FOR_LEAF_NODE, REBALANCE_FOR_INTERNAL_NODE défini des types de rééquilibrer.
- La propriete **mRoot** : BTNode est la racine de B-arbre
- Un nombre entier **mSize** est le nombre d'elements de B-arbre
- La propriete **mIntermediateInternalNode** stocke un parent-nœud temporaire qui est utilisé pour stocker le parent-nœud et passer globalement dans une suppression récursive.

```
682
683     // mIntermediateNode is done in findPrecessor
684     return deleteKey(mIntermediateInternalNode, predecessorNode, deletedKey.mKey, mNodeIdx);
685 }
```

- La propriété mNodeIdx stocke la position d'une clé dans un nœud en accompagnant la propriete **mIntermediateInternalNode** dans la méthode de suppression récursive.
- mStackTracer : Stack stocke le chemin d'équilibrer et évite une grande opération récursive (qui provoque une erreur StackOverflow).

3. Développez vous-même, ou cherchez sur internet, une structure/classe d'arbre AVL offrant les opérations ci-dessus.

Nous utilisons une structure d'arbre AVL offrant les opérations nécessaires:
Ce sont la classe Node et la class AVLTree :

```

class Node {
    int key, height;
    Node left, right;

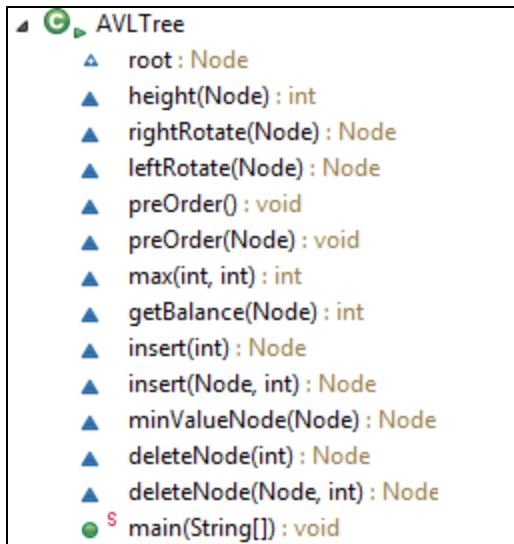
    Node(int d) {
        key = d;
        height = 1;
    }
}

public class AVLTree {

    Node root;

```

Ce sont ses méthodes nécessaires pour faire les opérations bases :



```

AVLTree
  ▲ root: Node
  ▲ height(Node): int
  ▲ rightRotate(Node): Node
  ▲ leftRotate(Node): Node
  ▲ preOrder(): void
  ▲ preOrder(Node): void
  ▲ max(int, int): int
  ▲ getBalance(Node): int
  ▲ insert(int): Node
  ▲ insert(Node, int): Node
  ▲ minValueNode(Node): Node
  ▲ deleteNode(int): Node
  ▲ deleteNode(Node, int): Node
  ● S main(String[]): void

```

3. Effectuez des expériences pour comparer l'efficacité en temps et en mémoire des B-arbres et des ABRs.

Dans un premier temps, vous ferez uniquement des ajouts dans les deux structures, puis des ajouts et des suppressions. Essayez au moins deux cas : insertion des valeurs croissantes et aléatoires.

On crée une méthode pour compter l'espace mémoire inutilisé de b-arbre. On utilise Stack pour éviter une grande boucle récursive.

```

// Count unused memory of btree
public int getUnusedMemory() {
    if (mRoot == null)
        return 0;

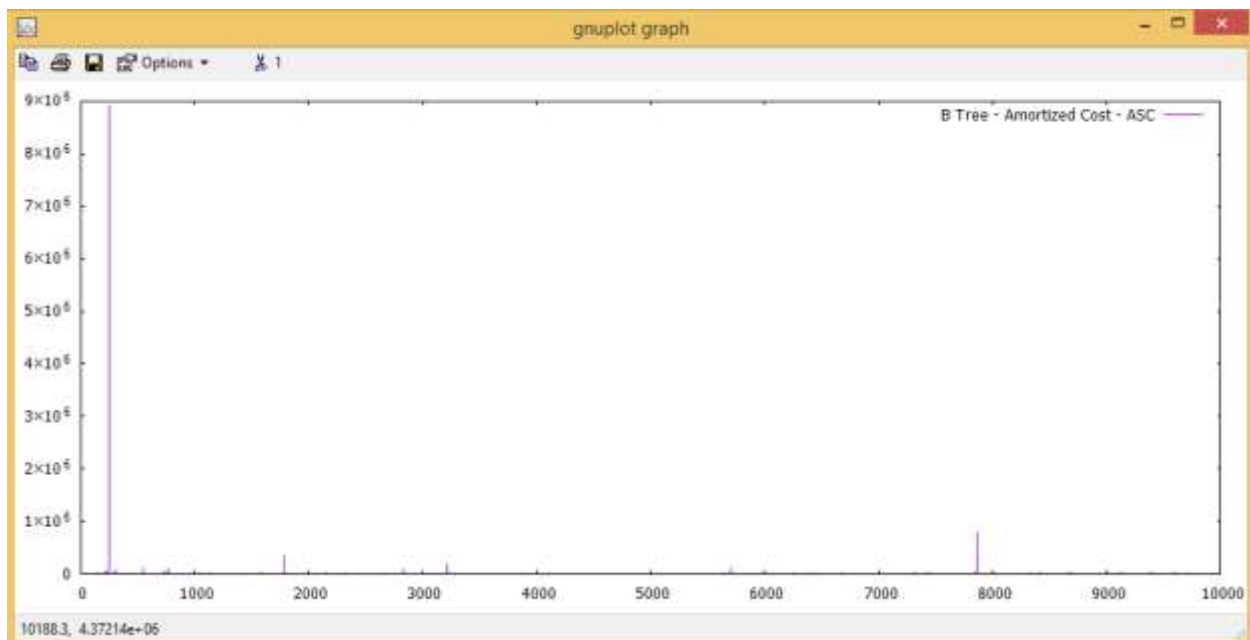
    int count = 0;
    Stack<BTNode> stack = new Stack<BTNode>();
    stack.add(mRoot);
    while (!stack.isEmpty()) {
        BTNode node = stack.pop();
        count += node.getUnusedMemory();
        for (int i = 0; i <= node.mCurrentKeyNum; i++) {
            stack.add(node.mChildren[i]);
        }
    }
    return count;
}

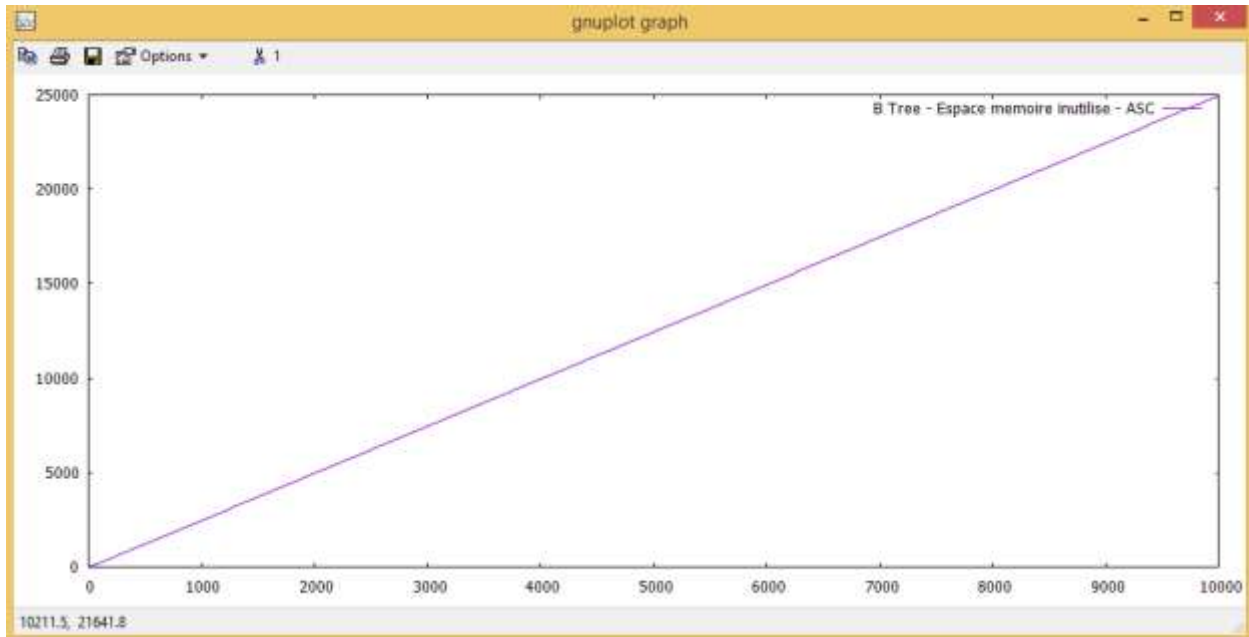
```

- Dans le cas où on ajoute des clés croissantes dans 10 mille opérations
Ce sont les résultats de B-arbre :
 - L'espace mémoire inutilisé égale suivant à 2.5 fois le nombre d'éléments.

Total cost: 25629800

Average cost: 2562.980

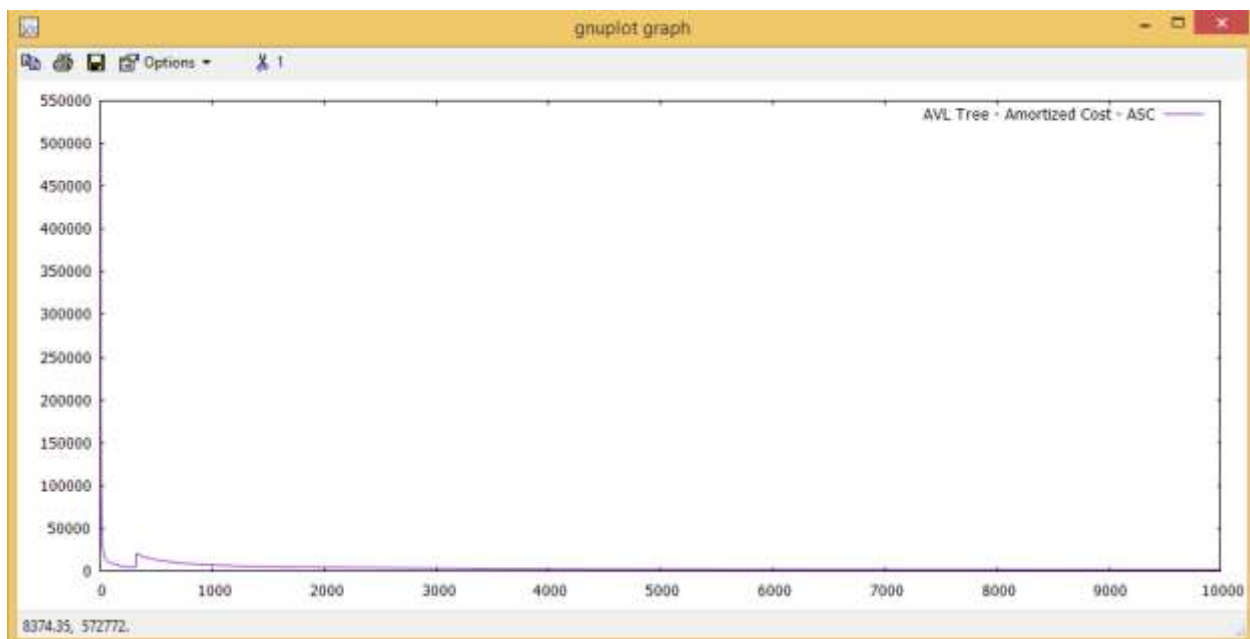
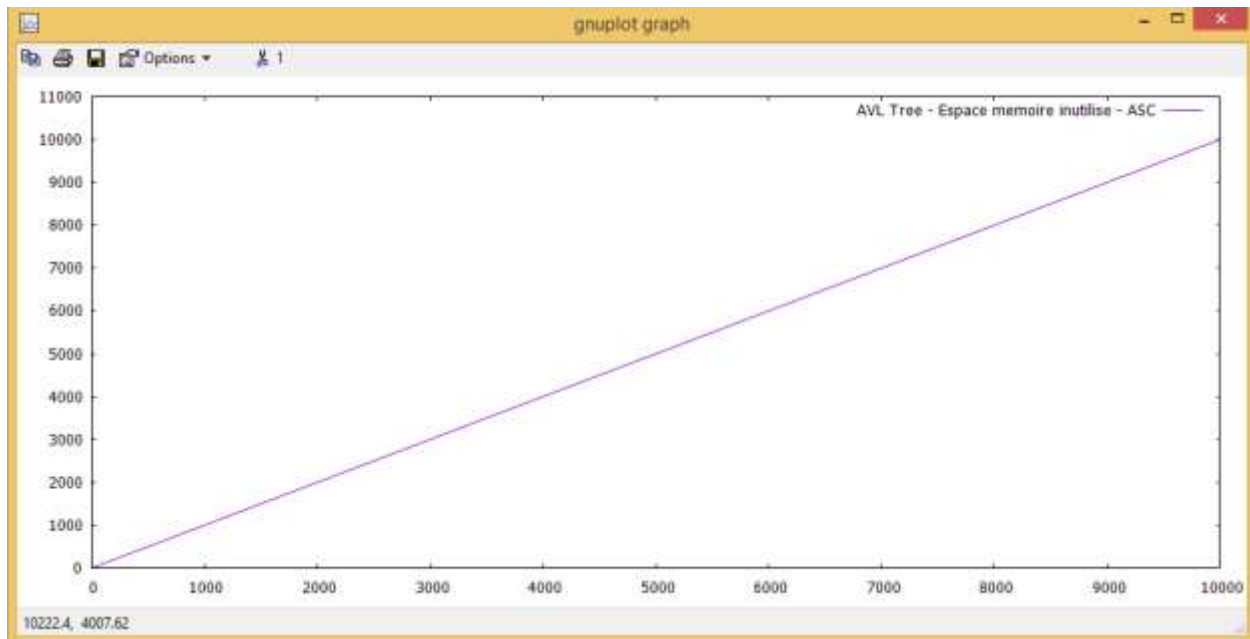




On crée une méthode et ajoute une propriété *size* pour compter l'espace mémoire inutilisé d'AVL-arbre. L'idée est à compter les références inutilisées de toutes les feuilles dans AVL-arbre. Parce que les références inutilisées peuvent provoquer l'erreur StackOverflow.

```
public int getUnusedMemory() {  
    if (root == null) return 0;  
    int height = height(root);  
    int sizeOfHeight = (int) Math.pow(2, height);  
    int sizeOfHeight2 = (int) Math.pow(2, height+1);  
  
    int count = sizeOfHeight2 - sizeOfHeight + size - sizeOfHeight - 1;  
  
    return count;  
}
```

- L'espace mémoire inutilisé égale au nombre d'éléments + 1 puisque les références inutilisées sont toujours. Par exemple : On a 7 éléments, alors le nombre d'feuilles est 4 et le nombre de références



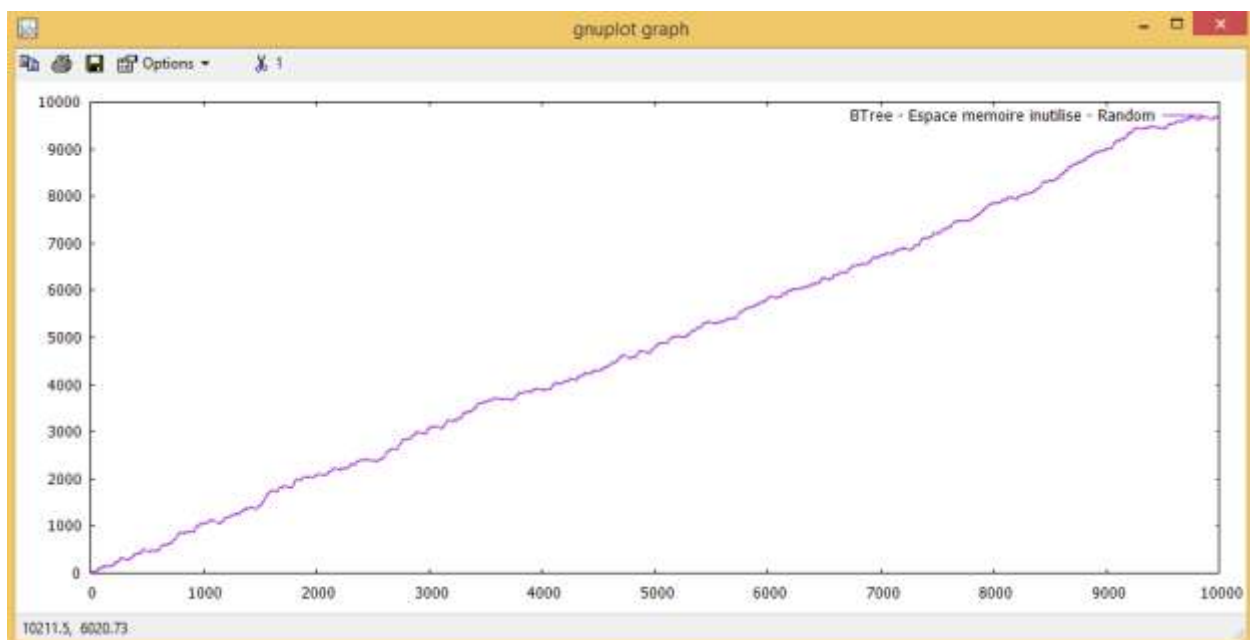
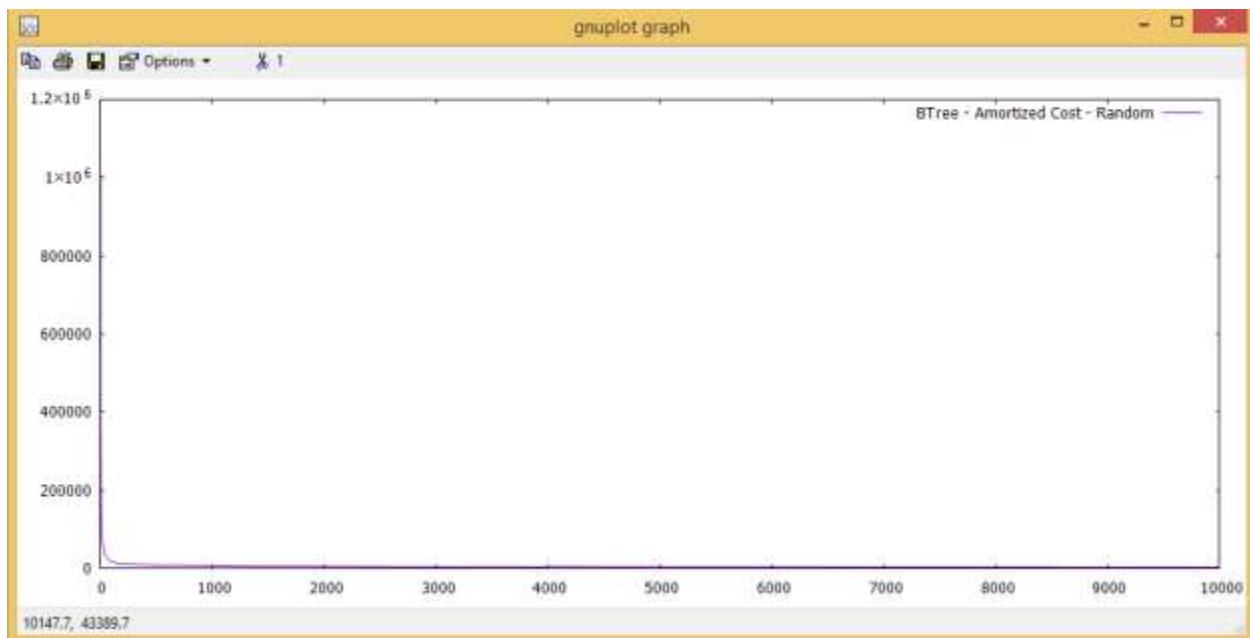
- Dans le cas où on ajoute des clés aléatoires

BTree :

- L'espace mémoire inutilisé de ce cas est plus petit que l'espace mémoire inutilisé de premier cas. De plus, l'espace mémoire inutilisé égale semble au le nombre d'éléments de B-arbre

Total cost : 31214732

Average cost : 3121.4732



AVL Tree :

Total cost : 9990869

Average cost : 999.0869

