# LOG3430 – Méthodes de test et de validation du logiciel

---

**Laboratoire 2**

**Tests de partition de catégorie et de flot de données**

**Département de génie informatique et de génie logiciel**

**École Polytechnique de Montréal**


**Soumis par**

**Roman Zhornytskiy (1899786)**

**Hakim Payman (1938609)**

**Gabriel Tagliabracci (1935775)**

**Groupe : 02**

**Soumis à Noureddine Kerzazi**

**Hiver 2020**

## 4. Travail à effectuer

**4.1.** Pour afficher la liste des nœuds en ordre (inorder traversal), on exploite la propriété de base d'un BST pour lire les nœuds de l'arbre de gauche à droite. Utilisez l'approche de partition de catégories EC pour tester les opérations **d'insertion et d'affichage**. Astuce : considérez la propriété « **le BST est trié correctemen**t » comme une des catégories. Dans votre rapport, il faut aller jusqu'au niveau des cas de test, **pas** l'écriture des tests avec l'échafaudage unittest. [4 points]

Cas de test pour l'insertion
V : number. Cela représente la valeur du nœud.
O : any. Peut représenter un objet ou un type de base (int, char, etc.) autre que number.
N : nœud.
B : BST.

B1 = {B = not None} [propriétés : le BST est trié correctement]
B2 = {B = not None} [propriétés : le BST n'est pas trié correctement]
B3 = {B = None}

V1 = {valeur < 0} [propriétés : number]
V2 = {0 < valeur < ∞} [propriétés : number]
V3 = {valeur = O} [erreur]
V4 = {valeur = valeur} [la valeur existe déjà dans l'arbre, propriétés : number]

N1 = {N: None}
N2 = {N: not None}
N3 = {N: root} [Le nœud est la racine du BST]
N4 = {N: leaf} [Le nœud n'a pas de fils]

B1V1N1 -> t1 = < {B = 1,2,3, valeur = -420, N = None}, {-420 est insérée} >
B2V2N2 -> t2 = < {B = 3,2,1, valeur = 4, N = not None}, {BST n'est pas trié correctement} >
B3V3N3 -> t3 = < {B = None, valeur = "hello", N = root}, {Erreur, comparaison indéfinie} >
B1V4N4 -> t4 = < {B = 1,2,3, valeur = 1, N = leaf}, {Erreur, cette valeur existe déjà dans le BST} >

Cas de test pour l'affichage
B : BST.

B1 = {B: None}
B2 = {B: not None} [propriétés: le BST est trié correctement]
B3 = {B: not None} [propriétés: le BST n'est pas trié correctement]

B1 -> t1 = < {B = None}, {Le BST est vide} >
B2 -> t2 = < {B = 1,2,4}, {1,2,4} >
B3 -> t3 = < {B = 10,2,1}, {10,2,1} >

**Note** : pour le test 3 (t3) de l'affichage on assume que la méthode d'affichage ne vérifie pas la validation de l'arbre.

**4.2.** Utilisez de nouveau l'approche de partition de catégories, mais cette fois avec AC au lieu de EC, pour l'opération de suppression de nœuds (méthode delete_node()) dans le fichier BST.py. Allez jusqu'au niveau de l'écriture des tests avec unittest. [6 points]

Cas de test pour la suppression
B : BST.
N : Nœud.

B1 : {B : None}
B2 : {B : not None}

N1 : {N : number} [le nœud est dans l'arbre]
N2 : {N : number} [le nœud n'est pas dans l'arbre]

B1N1 -> t1 = < {B = None, N = None}, {Erreur, BST est None} >
B1N2 -> t2 = < {B = None, N = not None}, {Erreur, BST est None} >

B2N1 -> t3 = < {B = not None, N = None}, {Erreur, nœud est None} >
B2N2 -> t4 = < {B = not None, N = not None}, {Noeud supprimé avec succès}>

Il est à noter que le test t2 est impossible à tester, car il est contradictoire dans le sens où ajouter un nœud à B1 ne le rend plus « None ».

**4.3.** Utilisez maintenant l'approche boîte blanche all-P-uses/some-C-uses (flot de données) sur la même opération (méthode delete_node()) dans le fichier BST.py que 4.2 (suppression de nœuds) en regardant le code source joint à cet énoncé. Allez de nouveau jusqu'au niveau de l'écriture des tests avec unittest. Comparez vos résultats avec ceux de 4.2, qu'est-ce que vous remarquez? [6 points]

Le diagramme de flot de contrôle sur la prochaine page nous a permis d'établir le jeu de tests (voir test_BST.py) permettant de passer au moins une fois à travers tous les blocs conditionnels, c'est-à-dire les blocs employant un prédicat (P-Use) pour faire bifurquer le déroulement du programme dans une branche. Nous avons donc couvert les chemins suivants avec 10 tests (voir test_BST.py) :
Path1 = {A, B, W}
Path2 = {A, C, B, W}
Path3 = {A, C, D, E, F, J, K, U, W}
Path4 = {A, C, D, E, F, G, H, K, U, W}
Path5 = {A, C, D, E, F, G, I, K, U, W}
Path6 = {A, C, D, E, K, L, M, O, S, T, U, W}
Path7 = {A, C, D, E, K, L, M, O, P, R, T, U, W}
Path8 = {A, C, D, E, K, L, N, O, P, Q, T, U, W}
Path9 = {A, C, D, E, K, U, W}
Path10 = {A, C, D, E, K, U, V, W}

Nous remarquons que nous avons beaucoup plus de tests qu'à la question 4.2 car l'approche en 4.3 exige de couvrir beaucoup plus de cas de figures qu'en 4.2.

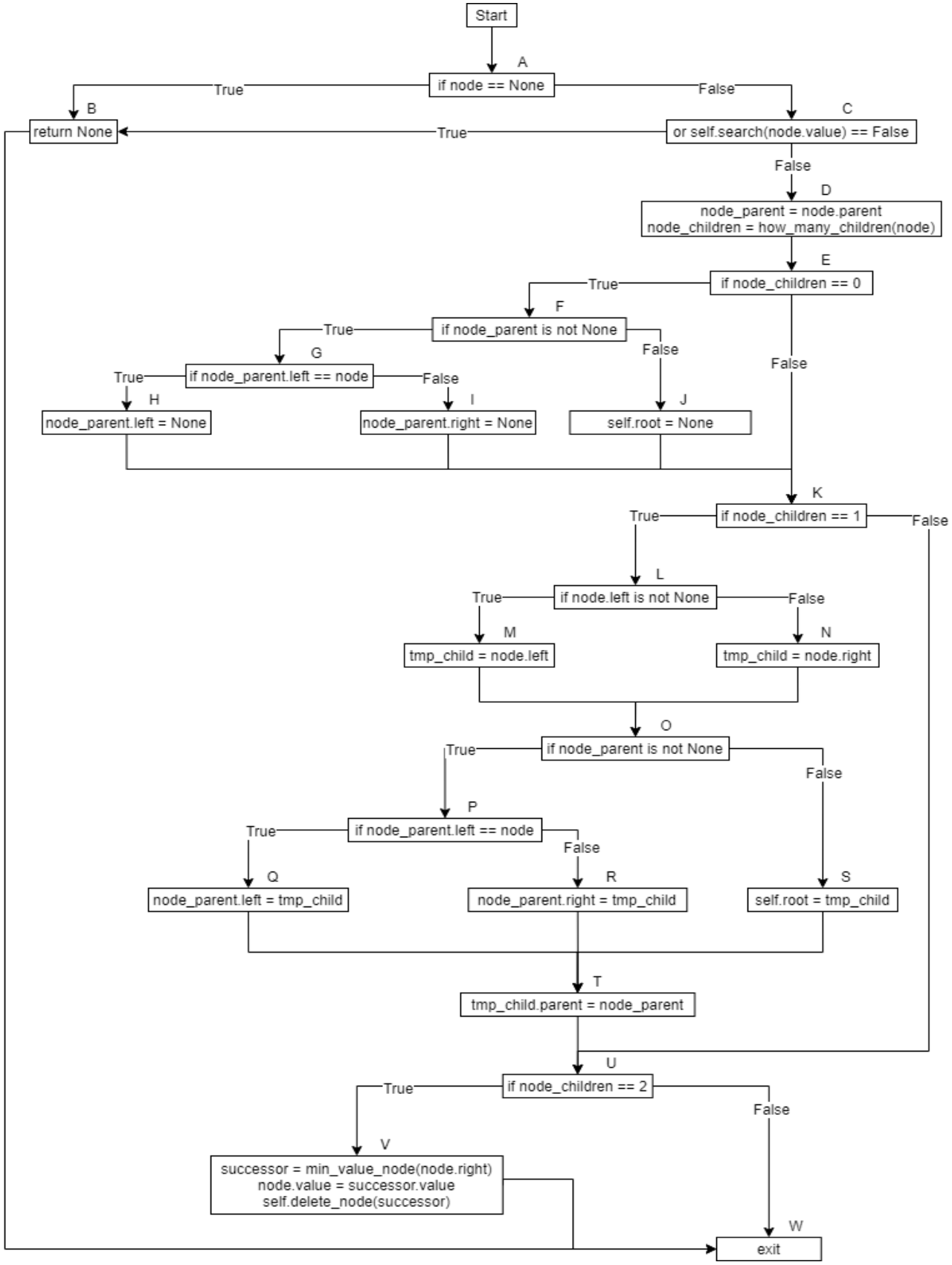


*Figure 1: Diagramme de flot de contrôle de la méthode delete_node*

**4.4.** Utilisez de nouveau l'approche boîte blanche all-P-uses/some-C-uses (flot de données), cette fois-ci sur la méthode invertTree() dans le fichier BST.py (inversion de l'arbre). Dans votre rapport, il faut aller jusqu'au niveau des cas de test, ****pas**** l'écriture des tests avec l'échafaudage unittest. [4 points]

Ici, nous considérons les méthodes _reversetree et reversetree séparèment. Elles ont donc un jeu de tests chacune.

Cas de test pour l'inversion (_reverseTree())
N : Nœud.

N1 = {N: None}
N2 = {N: number}

Path1 = {A, B, E}
Path2 = {A, C, D, E}

N1 -> t3 = < {N = None}, {return value = None} >
N2 -> t4 = < {N = 2}, {return value = 2} >

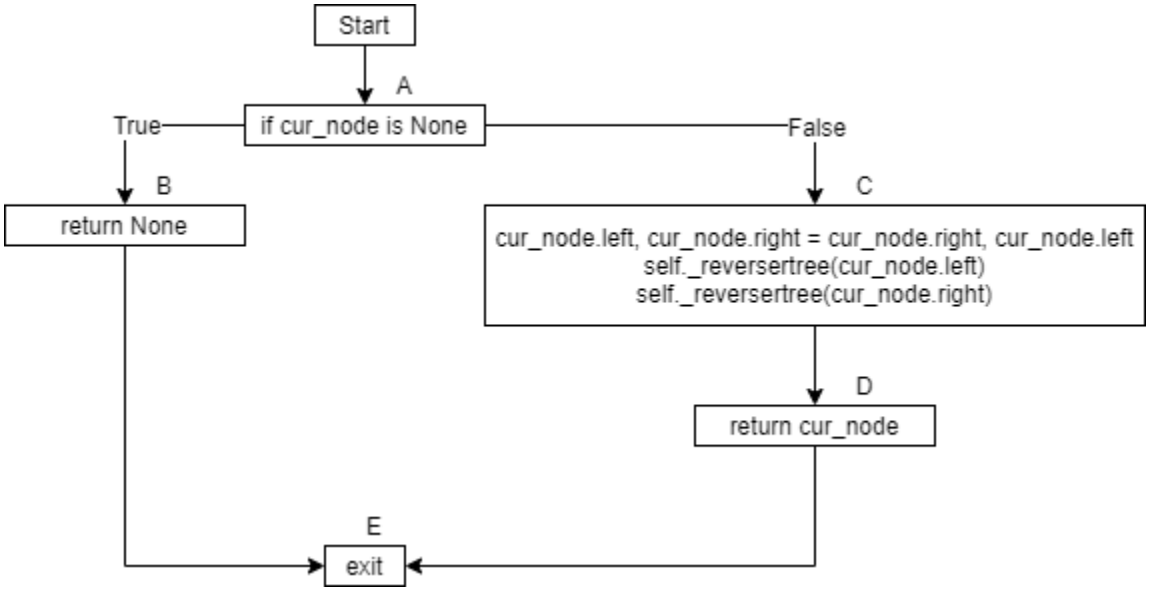


*Figure 2: Diagramme de flot de contrôle de la méthode _reverseTree*

Cas de test pour l'inversion (reverseTree())
B : BST.

B1 = {B: None}
B2 = {B: not None}

Path1 = {A, C, D}
Path2 = {A, B, D}

B1 -> t1 = < {B = None}, {return value = None} >
B2 -> t2 = < {B = 2}, {return value = 2} >

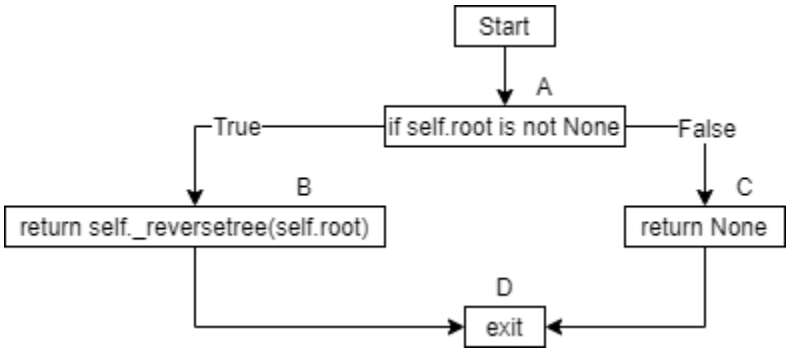


*Figure 3: Diagramme de flot de contrôle de la méthode reverseTree*