INFO-F103 – Algorithmique I
Projet 2 - Analyse des transactions bancaires avec les Arbres Binaires de Recherche

Youcef Bouharaoua
youcef.bouharaoua@ulb.be

Année académique 2022-2023

Introduction

Au cours des derniers mois, nous avons été témoins de l'effondrement de plusieurs institutions financières traditionnelles, comme "le Crédit du Listenbourg", ainsi que de l'échec de certaines plateformes de cryptomonnaie telles que "FTY". Ces événements ont ébranlé la confiance du public envers les systèmes bancaires et financiers conventionnels. Face à ces défis, en tant que futurs informaticiens, vous avez une opportunité unique d'explorer et de développer des solutions innovantes pour rétablir la confiance dans le secteur financier.

Dans ce contexte, vous décidez d'utiliser vos connaissances en programmation et en algorithmique afin créer un système de transactions alternatif basé sur le concept des arbres.

Les arbres sont des structures de données hiérarchiques fréquemment utilisées en informatique pour organiser et manipuler des données. Ils peuvent offrir des avantages significatifs en termes d'efficacité des transactions car certains types d'arbres permettent notamment une recherche plus efficace.

Au cours de ce projet, notre attention se focalisera principalement sur les aspects algorithmiques du projet. Cependant, nous vous encourageons vivement à revisiter ce sujet ultérieurement, une fois que vous aurez acquis une base solide en informatique en général, ainsi que des compétences spécifiques en cryptographie et en sécurité.

Énoncé

Dans le cadre de ce projet, pour gérer les transactions du système en cours de développement, il est nécessaire d'adapter la classe *BinarySearchTree* étudiée en cours, afin qu'elle réponde aux spécifications du système de transactions basé

sur les arbres. Une transaction est représentée par un nœud dans l'arbre de recherche binaire, contenant un identifiant unique, un montant et des références aux autres nœuds. Vous pouvez ajouter d'autres informations si nécessaire. Pour ce faire, suivez les étapes ci-dessous :

- Commencez par implémenter la classe *Node* fournie ci-dessous, qui servira de base pour les nœuds de notre arbre de transactions :

```
class Node:
def __init__(self, transaction_id, transaction_amount):
    # Entier qui permet d'identifier de la transaction
    self.transaction_id = transaction_id

    self.left = None
    self.right = None
    self.father = None
    self.height = 1
    self.transaction_amount = transaction_amount
```

- Ensuite, modifiez la classe *BinarySearchTree* pour qu'elle utilise des objets de la classe *Node* au lieu d'utiliser directement les valeurs des éléments. Vous devrez également adapter les méthodes de la classe *BinarySearchTree* pour gérer les objets *Node*. Voici quelques exemples de modifications que vous devrez apporter :
  - Modifiez la méthode *init* pour initialiser l'arbre avec un objet *Node*.
  - Adaptez la méthode *insert* pour qu'elle insère un objet *Node* dans l'arbre.
  - Adaptez la méthode *find* pour qu'elle recherche un objet *Node* en fonction de sa clé.
  - Adaptez la méthode *remove* pour qu'elle supprime un objet *Node* de l'arbre.
- Une fois que vous avez adapté la classe *BinarySearchTree*, assurez-vous de tester son fonctionnement avec des exemples de transactions et de vérifier que les opérations d'insertion, de recherche et de suppression fonctionnent correctement.

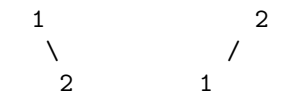
Exploration des structures d'arbres binaires de recherche

Dans le cadre de notre projet de système de transactions basé sur les arbres, il est essentiel de comprendre comment différentes structures d'arbres binaires de recherche peuvent impacter la performance et la sécurité de notre système. En résolvant cet exercice, vous allez explorer les diverses combinaisons possibles d'arbres pour gérer efficacement les transactions et vous préparer à concevoir un système capable de s'adapter aux différentes situations que vous pourriez

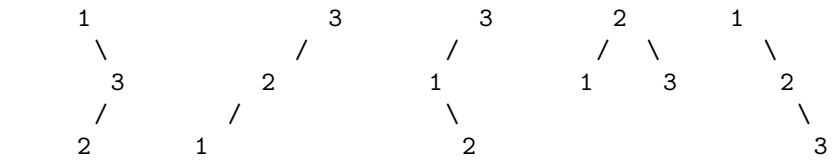
rencontrer.

Nous vous demandons de créer une fonction **récursive** permettant de déterminer combien d'arbres binaires de recherche structuralement uniques peuvent être construits à partir des identifiants allant de 1 à T, où T est un identifiant maximum donné en paramètre. Les arbres binaires de recherche doivent être construits sur la base de ces numéros d'identifiant.

Par exemple, pour T = 2, il y a 2 arbres uniques :



Pour T = 3, il y a 5 arbres possibles :



Bonus:dans le fichier PDF contenant vos réponses, proposez une méthode **non récursive** et **sans construction d'arbres** pour calculer efficacement le nombre d'arbres binaires de recherche uniques ayant n sommets. Expliquez comment vous êtes parvenu à cette solution. Assurez-vous de détailler clairement votre raisonnement.

Identification des transactions les plus importantes

Dans un système de transactions, il est essentiel de pouvoir repérer rapidement les transactions les plus significatives, telles que celles avec des montants élevés, afin de garantir un traitement prioritaire et/ou pour identifier des activités inhabituelles.

De ce fait, nous vous demandons d'implémenter une méthode qui étant donné un arbre binaire de recherche de transactions et un entier positif *j*, trouve la j-ième plus grande valeur de *transaction_amount* dans l'arbre binaire de recherche. Dans ce contexte, la j-ième transaction la plus importante est celle dont le montant est à la j-ième position si on devait trier les transactions par montants décroissants.

Nous vous demandons donc:

- Implémentez une version **récursive** de la fonction pour trouver la transaction dont le montant est à la j-ième position si on les trie par importance (montants décroissants). Dans cette version, vous devez implémenter la méthode *jemePlusImportante* qui prend en paramètre un entier *j* et appelle la méthode récursive *jemePlusImportanteUtil* avec la racine globale de l'arbre, *j* et une liste *tab* initialisée à [0].

- Implémentez une version itérative de la fonction pour trouver la transaction dont le montant est à la j-ième position si on les trie par importance (montants décroissants).
- Comparez la complexité temporelle et spatiale des deux méthodes (récursive et itérative). Y a-t-il une approche préférable à l'autre ? Justifiez votre choix.

Voici un exemple, Dans celui-ci nous avons décidé d'afficher un tuple (id,*transaction_amount*) pour chaque noeud, nous vous encourageons bien évidemment à faire un meilleur affichage :

```
       5,50
     /      \
    3,30     7,70
   /  \      /   \
 2,20 4,40 6,60  8,80

La 3eme plus grande valeur de transaction_amount
dans cet arbre est 60 qui a comme id 6.
```

Correction des transactions

Vous avez identifié une occasion d'acquérir une institution "FTZ" ayant une gestion désastreuse qui a subi une attaque informatique.

FTZ enregistre ses transactions dans un fichier *.txt*. Vous voulez maintenant convertir toutes leurs transactions afin qu'elles soient représentées dans votre nouveau système que vous êtes en train de développer.

Votre équipe de cybersécurité a identifié la faille : un "hackeur" s'est amusé à modifier les montants de certaines transactions. Pour résoudre ce problème, l'équipe vous suggère d'adopter la solution suivante :"Dans le fichier *transactions.txt*, remplacez chaque montant par le montant immédiatement supérieur dans les lignes suivantes du fichier. S'il n'y a pas de montants plus grands dans les lignes qui suivent, remplacez-le par -1". L'id de la transaction reste le même.

En d'autres termes, afin de résoudre le problème lié à la conversion des transactions enregistrées dans le fichier "transactions.txt", vous devez implémenter une solution basée sur les arbres de recherche binaires. Le but est de parcourir le fichier et de remplacer chaque montant par un montant supérieur dans les lignes suivantes, si un tel montant existe. Si aucun montant supérieur n'est trouvé, remplacez-le par -1. Votre défi est de trouver une approche optimale pour cette tâche en exploitant les propriétés des arbres de recherche binaires. Une fois les modifications effectuées, enregistrez les résultats dans un nouveau fichier nommé "transactions_X_corrected.txt".

Voici un exemple d'un fichier "transactions_1.txt"

```
(1,160)
(2,1160)
```

```
(3,1420)
(4,360)
(5,620)
(6,640)
(7,1260)
(8,1840)
(9,860)
(10,60)
(11,1820)
(12,1860)
(13,500)
(14,1600)
(15,560)
```

Et voici le fichier "transactions_1_corrected.txt" après avoir apporté la correction:

```
(1,360)
(2,1260)
(3,1600)
(4,500)
(5,640)
(6,860)
(7,1600)
(8,1860)
(9,1600)
(10,500)
(11,1860)
(12,-1)
(13,560)
(14,-1)
(15,-1)
```

Dans cette partie, nous vous demandons de discuter brièvement la complexité temporelle et spatiale de votre approche. De plus, veuillez fournir le pseudo-code (sans dévoiler le code complet) d'une autre méthode qui pourrait résoudre le problème en temps $\mathcal{O}(n \log n)$ sans nécessairement utiliser les ABR. Si vous optez pour ne pas utiliser les ABR, indiquez le nom de la structure de données abstraite (ADT) que vous auriez choisie et justifiez votre choix. Enfin, mettez en évidence les différences entre l'approche mise en œuvre et celle que vous avez proposée.

Fusion de systèmes

Dans le contexte de d'une fusion de deux institutions financières, il est nécessaire de regrouper les informations sur les transactions des deux systèmes en un seul

arbre. Le nouvel arbre contient les transactions des deux systèmes , facilitant ainsi la gestion des opérations. Nous vous demandons donc d'implémenter la méthode **fusion** qui prend en entrée deux arbres et qui retourne une liste contenant les éléments du nouvel arbre trié suivant un ordre "infixé" (comme vu au cours), tout en préservant l'ordre et les propriétés d'un arbre binaire de recherche, la complexité temporelle doit être aussi faible que possible.

```
# Exemple 1:
#Arbre 1:
        3
     /     \
    1       5

#Arbre 2:
     4
   /   \
2      6
Resultat de la fusion: 1 2 3 4 5 6
# Exemple 2:
#Arbre 1:
            16
          /  \
         4    20
       /
      2
#Arbre 2:
           15
         /
      9
    /
0
Resultat de la fusion: 0 2 4 9 15 16 20
```

Remise

Votre tache consiste à mettre en œuvre les méthodes et fonctions requises mentionnées ci-dessus et à analyser les complexités lorsque cela est nécessaire. Vous soumettrez votre projet sur l'Université Virtuelle.

Vous devez soumettre un fichier *.zip* qui contiendra un fichier *.py* contenant l'ensemble de votre code ainsi qu'un fichier pdf contenant l'ensemble de vos analyses. Nommez ces fichiers en utilisant le format suivant : *NomPrenomEtudiant_proj2algo.py* et *NomPrenomEtudiant_proj2algo.pdf* (en remplaçant par vos propres nom et prénom(s)). Par exemple, l'étudiant Cdeleau Ceprojet créera un fichier *.py* nommé *CeprojetCdeleau_proj2algo.py* et un fichier pdf nommé *CeprojetCdeleau_proj2algo.pdf*, puis les inclura dans une archive *.zip*.

Nous tenons à vous rappeler que la réalisation personnelle de vos projets est essentielle à votre apprentissage et à votre réussite universitaire. Si nous avons des raisons de croire que votre projet n'a pas été réalisé par vous-même, nous vous convoquerons pour une défense orale. Nous sommes conscients que certaines ressources peuvent être tentantes, mais nous insistons sur le fait qu'une utilisation inappropriée de ces ressources peut nuire à votre compréhension de la matière et à votre avenir professionnel. Nous sommes là pour vous aider et vous guider, alors n'hésitez pas à nous contacter si vous avez des questions ou des préoccupations.

Le projet est à remettre pour le 2 mai 2023 à 23h59 sur l'UV. Tout manquement aux consignes ou retard sera sanctionné directement d'un 0/10.

Evaluation

Seront évalués la qualité de votre code, la mise en pratique de la matière vue en cours, les optimisations mises en place et la qualité de vos tests. Veillez à ce que votre code soit commenté de manière concise pour mettre en valeur ses fonctionnalités et les optimisations appliquées. Dans le cas où l'exécution de votre code en ligne de commande produit une erreur, une note nulle sera reportée pour la partie exécution, veillez donc à bien tester votre code.