MOUMOU Nassima

Compte Rendu du TP2 (Arbres Binaires de Recherche)

Présentations :

Le but de ce projet de Tp Algo et Prog 5, était d'implémenter le type abstrait dictionnaire à l'aide des arbres binaires de recherche en utilisant les mêmes éléments que pour le TP d'AP4 (clé entière + valeur réelle). Dans ce compte rendu, nous présenterons nos méthodes de recherche, ajout, supprimer et affichage, ainsi que les résultats de nos tests.

La classe Element :

La classe Element permet de représenter un élément d'une structure de données telle qu'un arbre binaire de recherche. Cette classe encapsule deux informations pour chaque élément : une clé (un entier) et un contenu (un double). Les principes de cette classe sont les suivants :

- Constructeurs : la classe Element possède deux constructeurs : un constructeur par défaut et un constructeur avec deux arguments. Le constructeur par défaut initialise la clé et le contenu avec des valeurs aléatoires. Le constructeur avec deux arguments permet de spécifier la clé et le contenu pour chaque élément.
- Accesseurs et modificateurs : la classe Element fournit des méthodes pour accéder et modifier les informations clé et contenu. Les méthodes getCle() et getContenu() permettent d'accéder respectivement à la clé et au contenu de chaque élément. Les méthodes setCle() et setContenu() permettent de modifier respectivement la clé et le contenu de chaque élément.
- Méthode toString() : la méthode toString() permet de représenter chaque élément sous forme d'une chaîne de caractères. Elle renvoie une chaîne qui contient la clé et le contenu de l'élément.
- Utilisation de la classe Math : la classe Element utilise la classe Math de Java pour générer des valeurs aléatoires dans le constructeur par défaut. La méthode random() de la classe Math permet de générer des nombres aléatoires compris entre 0 et 1.

Les résultats numériques attendus dépendent de l'utilisation de la classe Element dans le reste du code. Dans l'exemple de la classe Noeud, l'objet Element est utilisé pour encapsuler une clé et un contenu pour chaque nœud de l'arbre binaire de recherche. Les valeurs numériques attendues dépendent donc des valeurs spécifiées lors de la création des

objets Element. Si les valeurs sont aléatoires (comme dans le constructeur par défaut), les résultats numériques attendus seront différents à chaque exécution du programme.

La classe Noeud :

La classe Noeud permet de représenter un nœud dans une structure de données comme un arbre binaire de recherche. Chaque nœud est caractérisé par un objet Element qui encapsule une clé (un entier) et un contenu (un double), ainsi que deux pointeurs vers les fils gauche et droit du nœud.

La classe possède deux constructeurs : un constructeur par défaut qui crée un nœud sans contenu ni fils, et un constructeur avec deux arguments permettant de spécifier la clé et le contenu de l'objet Element encapsulé dans le nœud. Des accesseurs et des modificateurs sont également définis pour le contenu et les fils gauche et droit.

La méthode toString() permet de représenter le nœud sous forme d'une chaîne de caractères. Si l'objet Element encapsulé dans le nœud est null, la méthode retourne "nil".

➔ Interprétation des résultats numériques attendus:

- Lorsqu'on crée un nouvel élément sans préciser la clé et le contenu, les valeurs de la clé et du contenu seront initialisées au hasard grâce à l'utilisation de la fonction Math.random() qui génère un nombre aléatoire entre 0 et 1.
- Si on crée un nouvel élément en précisant la clé et le contenu, ces valeurs seront directement affectées à l'élément.
- Les méthodes getCle() et getContenu() permettent de récupérer respectivement la clé et le contenu de l'élément.
- Les méthodes setCle() et setContenu() permettent de modifier respectivement la clé et le contenu de l'élément.
- Enfin, la méthode toString() permet d'obtenir une représentation textuelle de l'élément sous la forme "Clé: x -> Contenu: y", où x et y représentent respectivement la clé et le contenu de l'élément.

La classe ArbreBinaire :

L'implémentation fournie est une implémentation d'un arbre binaire de recherche. Cette structure de données est utilisée pour stocker des éléments en maintenant une relation d'ordre sur les clés de ces éléments. L'arbre binaire de recherche est construit de manière à ce que les éléments ayant une clé inférieure soient situés dans le sous-arbre gauche et ceux ayant une clé supérieure soient situés dans le sous-arbre droit du nœud courant. L'arbre binaire de recherche est un arbre binaire ordonné, ce qui signifie que pour chaque nœud de l'arbre, toutes les clés des éléments de son sous-arbre gauche sont inférieures ou égales à la

clé de l'élément stocké dans le nœud, et toutes les clés des éléments de son sous-arbre droit sont supérieures ou égales à la clé de l'élément stocké dans le nœud.

- Voici une description plus détaillée des méthodes utilisées dans le code pour la mise en place d'un arbre binaire de recherche :

1) Méthode de recherche :

La méthode de recherche permet de retrouver un noeud dans l'arbre à partir de sa clé. Elle se fait de façon récursive. Elle prend en entrée la clé à rechercher et le noeud courant à partir duquel la recherche doit commencer.
La méthode compare la clé à rechercher avec la clé du noeud courant. Si la clé recherchée est inférieure à celle du noeud courant, on descend dans le sous-arbre gauche en appelant récursivement la méthode de recherche avec le sous-arbre gauche comme nouveau noeud courant. Si la clé recherchée est supérieure, on descend dans le sous-arbre droit en appelant récursivement la méthode de recherche avec le sous-arbre droit comme nouveau noeud courant. Si la clé recherchée est égale à la clé du noeud courant, on a trouvé le noeud recherché.

2) Méthode d'ajout :

La méthode d'ajout permet d'ajouter un noeud dans l'arbre. Elle prend en entrée la clé du nouveau noeud à ajouter. Si l'arbre est vide, le noeud à insérer devient la racine de l'arbre. Sinon, on recherche la position du noeud à insérer en comparant sa clé avec celle des noeuds courants, jusqu'à trouver une feuille où l'on peut insérer le nouveau noeud.

La méthode compare la clé du nouveau noeud avec la clé du noeud courant. Si la clé du nouveau noeud est inférieure à celle du noeud courant et que le sous-arbre gauche est vide, le nouveau noeud devient le fils gauche du noeud courant. Si la clé du nouveau noeud est supérieure à celle du noeud courant et que le sous-arbre droit est vide, le nouveau noeud devient le fils droit du noeud courant. Si la clé du nouveau noeud est inférieure à celle du noeud courant et que le sous-arbre gauche n'est pas vide, on continue la recherche en appelant récursivement la méthode d'ajout avec le sous-arbre gauche comme nouveau noeud courant. Si la clé du nouveau noeud est supérieure à celle du noeud courant et que le sous-arbre droit n'est pas vide, on continue la recherche en appelant récursivement la méthode d'ajout avec le sous-arbre droit comme nouveau noeud courant.

3) Méthode d'affichage :

La méthode d'affichage permet d'afficher les noeuds de l'arbre dans l'ordre infixe (parcours gauche-racine-droit) ainsi que leur profondeur. Elle utilise une méthode auxiliaire

qui prend en compte la profondeur du noeud courant. La méthode d'affichage prend en entrée le noeud courant à partir duquel l'affichage doit commencer.

La méthode d'affichage appelle récursivement la méthode d'affichage avec le sous-arbre gauche comme nouveau noeud courant, puis affiche la clé et la profondeur du noeud courant, puis appelle récursivement la méthode d'affichage avec le sous-arbre droit comme nouveau noeud courant.

4) La méthode de calcul de la hauteur :

La méthode de calcul de la hauteur de l'arbre utilise une méthode récursive qui calcule la hauteur de chaque sous-arbre en comptant le nombre de niveaux, puis renvoie le maximum entre les hauteurs des sous-arbres gauche et droit, auquel on ajoute 1 pour tenir compte du niveau de la racine.

5) Méthode de suppression :

La méthode de suppression permet de supprimer un noeud de l'arbre à partir de sa clé. Elle prend en entrée la clé du noeud à supprimer et le noeud courant à partir duquel la suppression doit commencer.

La méthode de suppression est un peu plus complexe que les autres méthodes. Elle doit gérer plusieurs cas :

- Si l'arbre est vide, il n'y a rien à supprimer.
- Si la clé du noeud à supprimer est inférieure à celle du noeud courant, on descend dans le sous-arbre gauche en appelant récursivement la méthode de suppression avec le sous-arbre gauche comme nouveau noeud courant.
- Si la clé du noeud à supprimer est supérieure à celle du noeud courant, on descend dans le sous-arbre droit en appelant récursivement la méthode de suppression avec le sous-arbre droit comme nouveau noeud courant.
- Si la clé du noeud à supprimer est égale à celle du noeud courant, on a trouvé le noeud à supprimer. On doit alors gérer plusieurs cas :
    - Si le noeud n'a pas d'enfants, on peut simplement le supprimer en le mettant à null dans son parent.
    - Si le noeud a un seul enfant, on le remplace par son unique enfant.
    - Si le noeud a deux enfants, on doit rechercher le plus grand noeud dans le sous-arbre gauche (qui sera le prédécesseur du noeud courant) et le remplacer par le noeud courant. On supprime ensuite le noeud courant.

MOUMOU Nassima

La classe TestMain :

La classe TestMain est une implémentation de l'arbre binaire de recherche (ABR) dans Java. Il teste principalement deux méthodes : ajout et suppression.

La méthode remplirAlea est une méthode utilitaire qui génère un tableau d'éléments aléatoires. La méthode main contient deux parties :

- Dans la première partie, la méthode teste quelques fonctionnalités de l'implémentation de l'ABR, telles que l'ajout, la recherche et la suppression de nœuds dans l'arbre. Ensuite, elle affiche l'arbre de recherche binaire résultant.
- Dans la deuxième partie, le programme teste l'efficacité pratique des méthodes ajout et suppression sur un grand nombre d'éléments en mesurant le temps d'exécution.

Le programme crée deux tableaux d'éléments : un tableau d'éléments tab de taille NB_ELEM et un tableau ASUPPRIMER de taille NB_ELEM2. Le tableau tab est rempli avec des éléments aléatoires en utilisant la méthode remplirAlea. Le tableau ASUPPRIMER est rempli avec des éléments aléatoires de tab en utilisant une probabilité de 0,15 pour chaque élément, et en arrêtant dès que ASUPPRIMER est rempli.

Dans la partie de test de l'efficacité, le programme ajoute les éléments de tab à l'ABR, en les insérant un par un à l'aide de la méthode ajout, tout en remplissant également ASUPPRIMER avec les mêmes éléments en utilisant la probabilité susmentionnée. Ensuite, le programme effectue une série de tests : il insère 1000 nouveaux éléments aléatoires, puis supprime les 1000 éléments de ASUPPRIMER de l'arbre.

Enfin, le programme affiche les temps d'exécution pour l'insertion initiale, les insertions supplémentaires et la suppression. Les résultats numériques attendus dépendent de l'efficacité de l'implémentation de l'ABR et de la puissance de la machine exécutant le programme. En général, on s'attend à ce que les temps d'exécution augmentent linéairement avec la taille des tableaux tab et ASUPPRIMER. La suppression devrait prendre un peu plus de temps que l'insertion. La durée d'exécution des tests est affichée en nanosecondes.

Comparaison des résultats obtenus :

En moyenne, l'opération d'ajout en tête dans une liste chaînée prend environ 450 ns, tandis que l'opération d'insertion au milieu prend plus de temps. La méthode de suppression, quant à elle, a un temps d'exécution moyen d'environ 30179 ns.

En moyenne, l'opération d'ajout d'un élément dans un arbre binaire prend environ 11820 ns. La méthode de suppression, quant à elle, a un temps d'exécution moyen d'environ 3017 ns.

En comparant les temps d'exécution moyens de l'ajout d'un élément dans une liste chaînée et dans un arbre binaire, on peut voir que l'ajout dans un arbre binaire est plus

rapide que dans une liste chaînée, avec un temps moyen de 11820 ns contre 450 ns pour la liste chaînée lors de l'ajout en tête. Cela peut être dû au fait que les arbres binaires sont souvent équilibrés, ce qui réduit le temps de recherche et d'insertion d'éléments. Cependant, le temps de suppression dans une liste chaînée est nettement plus rapide que celui dans un arbre binaire, avec un temps moyen de 3017 ns contre un temps indéterminé pour l'arbre binaire, ce qui peut être un inconvénient lors de la manipulation de grandes quantités de données dans les arbres binaires.

Complexité :

La complexité temporelle de l'ajout d'un élément dans une liste chaînée dépend de l'emplacement où l'élément doit être ajouté. Dans le cas de l'ajout en tête, le temps d'exécution est constant O(1) car l'élément est inséré au début de la liste. Cependant, pour l'ajout en milieu ou en fin de liste, le temps d'exécution est proportionnel à la taille de la liste et est de l'ordre de O(n), où n est le nombre d'éléments dans la liste.

En ce qui concerne l'ajout d'un élément dans un arbre binaire, la complexité temporelle dépend de la hauteur de l'arbre et de l'emplacement où l'élément doit être ajouté. Dans le pire des cas, où l'arbre est non-équilibré, la hauteur de l'arbre peut être de l'ordre de n, où n est le nombre d'éléments dans l'arbre, ce qui conduit à une complexité temporelle de O(n) pour l'ajout d'un élément. Cependant, dans le cas d'un arbre binaire équilibré, la hauteur de l'arbre est logarithmique par rapport au nombre d'éléments, ce qui donne une complexité temporelle de O(log n) pour l'ajout d'un élément.

En ce qui concerne la suppression d'un élément, la complexité temporelle dépend également de l'emplacement de l'élément dans la liste chaînée ou l'arbre binaire. Dans le cas d'une liste chaînée, la suppression d'un élément nécessite la recherche de l'élément dans la liste, qui a une complexité temporelle de O(n) dans le pire des cas. Ensuite, l'élément peut être supprimé en temps constant O(1) si on dispose d'un pointeur sur l'élément précédent, ou en temps linéaire O(n) si cela n'est pas le cas.

Dans le cas d'un arbre binaire, la suppression d'un élément peut avoir des implications sur la structure de l'arbre et peut nécessiter des opérations de rééquilibrage si l'arbre est équilibré. Dans le pire des cas, où l'arbre est non-équilibré, la suppression d'un élément peut avoir une complexité temporelle de O(n), tandis que dans le cas d'un arbre équilibré, la suppression a une complexité temporelle de O(log n).