Rapport de projet

Membres du binôme :

FONTENEAU Félix Groupe : 3
FRATI—PERALDI Saveria Groupe : 3

Intitulé du projet : Arbre binaire ordonné.

Un arbre binaire ordonné est un arbre binaire muni d'une relation d'ordre qui permet d'ordonner les éléments de l'arbre selon leur clé.

Le projet consiste à visualiser graphiquement des arbres binaires ordonnés auxquels on applique les opérations de base: la recherche, l'ajout et la suppression d'un élément.

Choix de programmation.

Dans le cadre de notre projet, nous avons utilisé la PlancheADessin (http://users.polytech.unice.fr/~vg/peip2/s3/PlancheADessin/PaD-doc/PaD/PlancheADessin.html) au lieu de Swing. M. Granet n'y a vu aucun inconvénient puisqu'il s'agit d'une interface graphique étudiée en TD au début d'année. Si la fenêtre de la planche à dessin ne s'ouvre pas correctement, veuillez modifier les paramètres x et y correspondant aux dimensions de la planche à dessin en ligne 21 du fichier *Main.java*.

Côté programmation, elle est orientée objet en ce qui concerne la construction des arbres. En effet, nous avons compartimenté le code en divers objets fondamentaux sous forme de classes. Il y a les classes principales dont ***ArbreBinaire***, ***ArbreBinaireOrdonne***, ***ArbreBinaireDessine*** et ***Element***. Ces classes sont liées les unes aux autres afin de mieux organiser la structure de notre application. Nous n'avons pas omis les classes Exception ***NoeudPrincipalException***, ***ElementDejaExistantException***, ***CleNonTrouveeException***, ***ArbreVideException*** qui complètent notre code afin de mieux gérer les potentielles erreurs.

Concernant la gestion des interactions avec l'utilisateur, l'application est programmée en impératif avec une méthode statique.

Fonctionnement du programme.

I. Niveau programmation.

La classe ***ArbreBinaire*** constitue l'interface d'un arbre binaire et définit donc toutes les méthodes abstraites d'un arbre binaire général. Le but étant de créer un arbre binaire ordonné, on va se servir de cette interface dans le but d'implémenter un arbre binaire ordonné pour créer notre application.

ArbreBinaireDessine est une classe abstraite implémentant une interface graphique sous forme de Planche à Dessin pour dessiner un arbre binaire quelconque.

La classe ***ArbreBinaireOrdonne*** implémente l'interface ***ArbreBinaire*** pour récupérer ses caractéristiques. La classe est construite de telle sorte qu'on construit l'arbre du tronc aux feuilles. Elle définit trois méthodes principales que sont l'ajout, la suppression et la recherche. Finalement, elle hérite d'***ArbreBinaireDessine*** afin de récupérer l'interface graphique qu'elle propose.

Son constructeur principal est ***ArbreBinaireOrdonne***, il prend en paramètre un seul élément et créé un ***ArbreBinaire*** avec un noeud et deux branches vides.

II. Niveau utilisateur.

L'utilisateur compile le fichier *Main.java*, il l'exécute puis le terminal ouvre une nouvelle Planche à dessin sur laquelle est dessiné un arbre binaire ordonné comprenant des éléments par défaut.

Le terminal accueille l'utilisateur et lui récapitule les quatre actions qu'il est en mesure d'effectuer sur cet arbre, à savoir : « a » pour ajouter un élément, « s » pour en supprimer un, « r » pour en rechercher un, « n » pour faire un nouvel arbre, et enfin « stop » pour quitter l'application.

Si l'utilisateur choisit :

¤ « a » : Le terminal lui demande de préciser l'élément qu'il souhaite ajouter sous la forme *int, string*.

¤ « s » : Le terminal lui demande de préciser l'élément (appartenant à l'arbre) qu'il souhaite supprimer sous la forme *int.* Le cas échéant, le terminal informera que l'élément entré n'appartient pas à l'arbre, et il peut réitérer sa demande.

¤ « r » : Le terminal lui demande de préciser l'élément qu'il souhaite rechercher dans l'arbre sous la forme *int*. Si l'élément n'est pas présent, alors l'utilisateur en sera informé dans le terminal.

¤ « n » : Le terminal lui demande de préciser l'élément qui constituera le premier nœud du nouvel arbre sous la forme *int, string*.

¤ « stop » : L’application se ferme et la planche à dessin disparaît.

Les entrées « a », « s » et « n » provoquent un changement automatique de la forme de l’arbre, tout en conservant l’arbre ordonné.

Les algorithmes.

Ici nous allons détailler le fonctionnement des méthodes visant à simuler les opérations de base de tout arbre binaire ordonné : l’ajout, la suppression et la recherche d’un élément.

- Ajout

L'ajout d'un élément dans l'arbre binaire ordonné se fait de façon récursive. A chaque étape, on compare la clef de l'élément à insérer avec la clef de l'élément courant.

Si il est supérieur, alors on va l’insérer dans le sous arbre droit, si ce dernier est non nul, sinon on l'ajoute récursivement au sous-arbre droit. Si il est inférieur, on fait la même chose, mais avec le sous arbre gauche. Si il est égal à l'élément courant, on envoie une exception afin de ne pas avoir deux même valeurs dans un arbre.

La méthode d'ajout est donc une recherche du nœud d'insertion ainsi que de l'insertion d'un élément à ce nœud. Au terme d’1 ou 2 comparaisons par étapes, et d'au maximum log2(n)-1 étapes dans le pire des cas pour un arbre à n éléments, on évalue la complexité de l'algorithme en terme de comparaisons à O(log2(n)).

- Recherche

La recherche d'un élément par sa clef dans l'arbre se fait aussi de façon récursive. A chaque étape, on effectue 2 comparaisons de clefs. Une première pour vérifier si l'élément courant est celui recherché, dans ce cas précis, on retourne la valeur de l'élément.

Sinon on compare la clef pour savoir dans quel sous-arbre rechercher l'élément. Si le sous-arbre concerné est vide, on retourne une exception : ***ClefNonTrouveeException***. Comme la méthode de l'ajout, au termes d’1 à 2 comparaisons par étapes, et d'au maximum log2(n)-1 étapes dans le pire des cas pour un arbre à n éléments, on évalue la complexité de l'algorithme en terme de comparaisons à O(log2(n)).

- Suppression

La méthode de suppression est un peu plus subtile. Dans notre implémentation d'un arbre binaire ordonné, le tronc de l'arbre ne peut

être supprimé, i.e. il ne pas avoir l'attribut d'arbre vide, car arbre vide est une variable statique finale.

On considère ainsi que le nœud principal de l'arbre (le tronc) ne peut être supprimé. Ainsi, si on appelle la méthode supprimer avec pour paramètre la clef du nœud principal, la méthode enverra une exception : ***NoeudPrincipalException***. Sinon, l'algorithme va rechercher récursivement le nœud à supprimer comme pour les deux méthodes précédentes.

Si l'élément à supprimer n'existe pas, on envoie une exception : ***ClefNonTrouveeException***.

Si l'élément est trouvé, on distinguera trois cas.

Le premier est que l'élément n'a pas d'enfant, il suffit donc de supprimer l'élément en lui attribuant la valeur de l'arbre vide.
Si il a un seul sous-arbre, on supprime l'élément et on décale son sous-arbre au nœud actuel.
Si il en a deux, on va chercher dans le sous-arbre droit (on peut aussi le faire de la même manière dans le sous-arbre gauche) l'élément le plus proche possible du nœud à supprimer (à gauche). Dès qu'on a trouvé cet élément, on le décale vers le nœud de l’élément supprimé. Puis on supprime le doublon de la même façon.

La recherche de l'élément à supprimer est en O(log2(n)). La suppression fait intervenir d'autres suppressions pouvant à leur tour faire augmenter de façon considérable le nombre d’appels récursifs. On évalue la complexité de l'algorithme en terme de comparaisons à O(nlog2(n)) dans le pire des cas (peut être amélioré) et à O(log2(n)) dans le cas général.