PG208
Programmation Orientée Objet / Langage C++
Projet

Y. Bornat A. Valade

April 17, 2024

Le projet de cette année est de faire un conteneur trié.

1 Comportement minmal

L'objectif est de créer la classe `Sorted` qui a la particularité d'être triée par nature. De nombreux éléments seront laissé à l'appréciation du binôme, selon leur ambition. Par contre les éléments suivants seront indispensables.

- Un constructeur à partir d'un `std::vector`.
- Une méthode `push` pour insérer un nouvel élément dans la structure. La complexité algorithmique de cette méthode doit être strictement inférieure à $O(n)$ où n est le nombre d'éléments déjà stockés.
- Une méthode `size` pour récupérer le nombre d'éléments dans le tableau.
- Une méthode `min` et une méthode `max` pour récupérer respectivement le l'élément minimal et l'élément maximal du tableau.
- Une surcharge pour l'opérateur de flux sortant (`<<`) qui permet l'affichage (et donc le debuggage).
- Une méthode `vect` qui construit un vecteur trié à partir de la structure.
- Un mécanisme qui empêche les fuites mémoires.

1.1 Structure

Pour que la structure soit toujours triée, et que l'insertion des éléments soit optimale en terme de performances, nous allons utiliser une structure chaînée en arbre binaire. C'est à dire que chaque élément de la structure (que nous nommerons *noeud*) a deux pointeurs vers l'élément suivant, nous les nomerons *left* et *right*. L'ensemble des noeuds constitue une arbre.

Chaque noeud contient un élement *elt*. S'il faut ajouter un nouvel élément *nw*, on le compare à *elt*. Si $nw < elt$, *nw* est ajouté à l'arbre de gauche, sinon, il est ajouté à l'arbre de droite. Si un arbre est vide (égal à `nullptr`), alors il suffit de créer un nouveau noeud dont les deux sous-arbres seront vides.

Pour lire les éléments dans l'ordre, il suffit de faire une lecture récursive :

- D'abord le sous-arbre de gauche
- Puis l'élément central
- Finalement le sous arbre de droite

1.2 Gestion de la mémoire

Pour éviter les fuites mémoires, il existe plusieurs solutions.

La plus simple est de mémoriser tous les noeuds de l'arbre dans un vecteur. Les deux pointeurs permettront de gérer la structure arborescente indépendamment de la gestion mémoire

Un peu plus évolué, il y a l'utilisation des pointeurs uniques vus au TP4.

Sinon, il y a la méthode *roots* qui consiste à faire appel à `new` et `delete`

1.3 Types de données

Le type de données stocké dans la structure est laissé ua ibre choix du binôme.

Concernant le type choisi pour l'arbre, le plus évident est d'utiliser une classe `noeud` et de considérer qu'un arbre (ou sous arbre) est un `noeud *`.

2 Version Avancée

Si vous avez peur de vous ennuyer, vous pouvez utiliser une ou plusieurs des variantes suivantes.

2.1 Container *template*

Pourquoi fixer le type de données que manipule le conteneur arbre ? Écrivez plutôt une version template qui permet de définir une structure générique.

2.2 Fonction de comparaison arbitraire

Il est également possible de définir votre propre fonction de comparaison dans le constructeur. L'intérêt est de garder une plus grande flexibilité au moment de l'éxécution. En effet, si vous définissez votre propre type de données, et que vous surchargez la comparaison, tout est figé au moment de la compilation.

Si vous fournissez la fonction de comparaison au moment de l'éxécution, le système devient plus souple. Notamment, pour représenter une scène 3D, il est souvent nécessaire de trier les triangles à tracer du plus lointain au plus proche de l'observateur. Le problème est donc que la relation d'ordre change en même temps que la position de l'observateur.

L'intéret de la relation d'ordre est donc de fournir un foncteur qui prend deux arguments a et b. Le foncteur renvoie -1 si $a < b$, 0 si $a == b$ et 1 si $a > b$. Selon vos préférences, vous pourrez proposer un méthode qui permet de retrier le tableau après un changement ou reparamétrisation du foncteur. Si vous refaites le tri du tableau, lire les éléments par ordre de tri (croissant ou décroissant) est à évite, car vous vous retrouveriez dans un cas particulièrement défavorable pour l'équilibrage de l'arbre, et donc la complexité du calcul.

3 Fonctions optionnelles

Dans le cas où vous auriez terminé un peu trop tôt. Voici de quoi occuper votre fin de projet. Vous pouvez implémenter, au choix, tout ou partie de ces variations.

- des méthodes `pop_min` et `pop_max` qui permettent de retirer et renvoyer respectievement l'élément le plus petit et le plus grand du tableau. Il est également possible de créer une unique méthode `pop` qui prend en argument une valeur spéciale (de type non assimilable à `int`) qui définit si on veut l'élément le plus petit ou le plus grand.
- Une surcharge pour la méthode `[]` qui permet de lire un élément comme dans un tableau. Selon le choix du binôme, cette méthode aura une complexité en $O(n)$ pour un encombrement minimal, ou en $O(log(n))$ au prix d'un ou deux attributs supplémentaires dans chaque noeud.
- Si les deux points précédents sont traités, il est possible modifier `pop` pour lire et supprimer un élément quelconque du tableau.

- Une méthode `balance` qui équilibre le nombre d'éléments dans les deux sous-tableaux d'un noeud. Cela permet d'augmenter les performances générales de la structure en limitant la longueur de la branche la plus longue.
- Le top du rafinement, vous pouvez faire un itérateur qui permet de lire successivement tous les éléments du tableau. (Cf conclusion du TP5 pour la définition d'un itérateur). Cet aspect sera très chronophage, en effet, il faut reproduire une pile de parcours des éléments du tableau.
- Un système permettant plusieurs tris sur les mêmes éléments (chaque noeud fait alors partie de plusieurs arborescences différentes, chacune ayant sa propre fonction d'ordre).

4 Modalités de restitution

- Par e-mail, en Pièce jointe
  - Le rapport est au format PDF, et fait moins de 1Mo
  - Le code est fourni soit en vrac (fichier `.cpp`) soit en `.zip`. Le rapport n'est pas dans le fichier compressé.
  - Le binôme est en copie.
- Date à définir lors de la première séance