Projet 1

I. Introduction

L'objectif de ce projet est d'étudier l'efficacité de deux implémentations différentes de la structure d'Anneau générique étudiée en travaux pratiques. Cette étude est principalement porté sur le coût spatial et temporel. Nous avons deux implémentations à étudier : un anneau implémenté avec une file et un autre avec deux piles. Nous ferons donc au cours de ce rapport une description détaillée des différentes méthodes ainsi que nos choix pour chacune des implémentations.

II. Implémentation des anneaux

a) Avec une file

Nous avons dans le dossier AnneauFile un fichier hpp et un fichier tpp. Le premier fichier comporte notre classe d'Anneau implémenté avec une file ainsi que la documentation des attributs et méthodes de l'objet. Nous avons donc comme attributs une file, avec le conteneur par défaut deque, et un entier nbElements représentant notre nombre d'éléments dans l'anneau. Nous avons choisis cette variable nbElements afin de se retrouver lors de nos méthodes ajoute() et supprime(), et de mieux visualiser les "transactions" d'éléments qui sont fait lors des ces deux méthodes. Nous aurions pu ne pas en créer et utiliser uniquement la méthode size() de la file pour la méthode taille(), c'est un choix que l'on espère judicieux.
Au niveau de la position courante, nous avons choisis que notre élément courant serait le premier élément de la file. Nous supprimons et ajoutons donc à cet endroit.

Notre constructeur Anneau(), qui créé notre objet, va initialiser la variable nbElements à 0 et rien d'autres. Le destructeur ~Anneau() lui est laissé par défaut. Notre courant étant définis comme le premier élément de la file notre méthode courant() va donc simplement renvoyer laFile.front(), le premier élément donc. Pour la suppression nous utilisons simplement la méthode pop() de la file, puisque c'est le courant qu'on supprime. contrairement à la méthode d'ajout qui elle a était un peu plus ardu. Pour ajoute() lorsque la taille de l'anneau est supérieur à 0 on utilise une file temporaire à laquelle on ajoute directement l'élément passé en paramètre via push(), ensuite dans une boucle for on ajoute à la file temporaire le courant de notre laFile et on défile notre laFile pour qu'à chaque fois nous y ajoutons la tête. Ensuite nous échangeons le contenu de notre file temporaire avec notre attribut laFile pour que laFile contienne les bon éléments via la méthode swap() de la file. Les autres méthodes étant identiques à l'autre implémentation nous les décrirons qu'une seul fois ci-dessous.
Nous n'avons rencontré aucun problème particulier lors de l'implémentation avec une file contrairement à celle avec les deux piles. Visualiser son fonctionnement a était plus long

que prévu. Nous avons finalement réussis à implémenter toutes ses méthodes, les problèmes majeurs rencontrés ont été énoncé ci-dessous.

Complexité temporel de l'anneau implémenté avec une file

Méthodes	**Ordre de grandeur**
estVide()	O(1) constant
ajoute(T t)	O(n)
supprime()	O(1) constant
courant()	O(1) constant
avance()	O(1) constant
recule()	O(n)
affiche()	O(n)
taille()	O(1) constant

b) Avec deux piles

Notre classe a pour attributs deux piles, ici laPileA et laPileB avec pour chacune un entier associé représentant le nombre d'éléments: nbElementsA et nbElementsB. La classe a, au passage, exactement les mêmes méthodes que l'implémentation avec une file.

Dans un premier temps nous avons le constructeur Anneau() qui va créer l'objet et simplement initialiser le nombre d'éléments de chaque pile à 0. Nous avons aussi le destructeur ~Anneau() qui est celui générer par le compilateur C++ par défaut.

Ensuite, on trouve la méthode estVide(), qui nous permet de savoir si l'anneau est vide. On a donc choisi de seulement vérifier si nbElementsA est égale à 0 car nous avons défini notre structure de tel sorte que notre élément courant se retrouve être le sommet de laPileA et que notre pile B permette juste de gérer les méthodes avance() et recule().
Nous faisons une explication détaillé de ce choix lors des deux méthodes annoncées ci-dessous.

Nous avons maintenant les méthodes ajoute() et supprime( ) qui vont juste avoir une incidence sur le courant. Pour ajouter un élément, comme nous utilisons l'objet stack de la STL on a aussi accès à ses méthodes, nous avons donc push() qui permet d'ajouter l'élément passé en paramètre au sommet de la pile et nous incrémentons le nombre d'élément de laPileA. Pour supprimer un élément, on vérifie d'abord que l'anneau est pas vide à l'aide de assert(!estVide()), si l'anneau comporte au moins un élément on utilise pop( ) qui permet de supprimer le sommet.

On a ensuite un problème qui pourrait survenir si il nous reste seulement un élément dans laPileA et plusieurs éléments dans laPileB car si l'on supprime le dernier élément de notre PileA on se retrouverait dans un cas où notre courant serait vide et notre anneau serait considéré comme vide alors que laPileB ne l'est pas. On fait donc une condition if qui va vérifier ce cas pour ainsi transférer toute laPileB dans laPileA et ainsi garder l'ordre des éléments de notre anneau. A la fin de cette méthode on décrémente le nombre d'éléments de A.

Après nous avons la méthode courant() qui va simplement retourner notre élément courant qui est celui placé au sommet de laPileA a l'aide de top(). On arrive enfin aux deux méthodes les plus importantes de notre anneau et sur lesquelles nous avons eu le plus besoin de réflexion.

Pour la méthode avance() nous avons décidé de simplement prendre le sommet de laPileA pour le mettre au sommet de laPileB ainsi on décale le courant de laPileA tout en gardant l'ordre des nos éléments en le mettant dans laPileB. On a dû répondre au problème posé par le fait qu'il reste seulement un élément dans notre pileA et donc qu'il faudrait transférer laPileB dans laPileA. On fait donc une vérification de ce cas ainsi qu'une boucle while qui va permettre de transférer un à un les éléments d'une pile a l'autre tout en conservant l'ordre. On incrémente et décrémente logiquement pour chaque utilisation de push() ou de pop() qui va avoir une influence sur le nombre d'éléments présent dans chacune des piles.

Pour la méthode recule(), on retrouve quasiment la méthode avance() mais en miroir. On va simplement prendre le sommet de laPileB et le mettre au sommet de laPileA, cependant comme juste avant on a un cas limite lorsque laPileB est vide, il faut alors mettre tous les éléments de laPileA dans laPileB sauf le dernier qui devient donc notre courant.

Enfin il nous reste les méthodes taille( ) et affiche( ) qui sont extrêmement simple. Pour la taille de l'anneau, on calcule juste la somme du nombre d'élément dans laPileA et dans laPileB. Contrairement à la méthode estVide() on a besoin de connaître le nombre d'éléments présent dans les deux piles. Cela est dû au fait que les méthodes avance() et recule() vont transférer les différents éléments d'une pile à l'autre ainsi il est possible qu'ils soient stockés séparément.

Pourquoi avoir fait ce choix? Nous avons pensé à une autre possibilité qui serait de se servir de laPileB comme une pile tampon qui aurait été utilisé pour transférer tous les éléments à chaque fois que l'on utilise avance() ou recule() puis tout remettre dans laPileA, mais il était logique que ce serait beaucoup plus coûteux à chaque opération.

Pour finir nous avons donc la méthode d'affichage qui va nous afficher le courant a l'aide d'une boucle qui permettra de faire le tour de l'anneau avec la méthode avance().

Complexité temporel de l'anneau implémenté avec deux piles

Méthodes	**Ordre de grandeur**
estVide()	O(1) constant
ajoute(T t)	O(1) constant
supprime()	O(n)
courant()	O(1) constant
avance()	O(n)
recule()	O(n)
affiche()	O(n)
taille()	O(1) constant

III. Jeu de test

Nous avons un fichier test où l'on test les différentes méthodes de l'anneau afin de s'assurer du bon fonctionnement de ces dernières. Le code a était commenté en adéquation aux tests pour plus de clarté. Nous pouvons tester les deux implémentations avec le même fichier, il y'a juste à changer le fichier inclus, cela est précisé dans le code. Nous avons donc choisis de tester l'anneau avec des 'char', on créé un anneau représentant le mot "bonjour" et on y essaye différentes méthodes. On test les méthodes taille(), supprime(), et affiche() et on fait ensuite pivoter l'anneau avec les méthodes avance() et recule(). Nous avons réunis tous les test dans une seul fonction afin de laisser la fonction main() vide pour vous laisser faire vos propres test en cas de besoin. La suite du projet, notamment la partie avec dedoublonne() est principalement assuré avec des anneaux d'Int.

IV. Dedoublonne

Nous utilisons un fichier à part pour toutes les fonctions relatives à dedoublonne(). Nous avons notre fonction dedoublonne() qui prends un anneau en paramètre et qui en renvoie un sans les doublons. Nous avons choisis de créer une fonction contient(), qu'on appelle dans dedoublonne() pour tester la présence d'un élément dans l'anneau passé en

paramètre. Ce choix a été fait pour une meilleur clarté et une meilleur réutilisation de ce principe, en effet si on aurait était amené à retester l'appartenance il nous aurait fallu faire la fonction. Nous pouvons également l'utiliser dans des futurs jeu de test afin de rendre nos implémentations d'anneau optimales. Revenons à notre fonction, on instancie un anneau temporaire, celui que la fonction renverra, et par la suite on parcours l'anneau passé en paramètre. On utilise donc une boucle for et à chaque itération on regarde si l'anneau temporaire ne contient pas l'élément courant de l'anneau passé en paramètre avec la fonction contient(), si l'élément n'y est pas, on l'ajoute donc à l'anneau temporaire. En dehors de notre condition if on fait pivoter l'anneau avec la méthode avance() pour que le courant change à chaque itération quelque soit le résultat du if. L'anneau temporaire est donc renvoyé par la fonction, sans les doublons de l'anneau passé en paramètre. L'ordre de grandeur de la fonction est $O(n^2)$, nous faisons deux boucles for allant de 1 à n (nb est la taille de l'anneau), une des boucles est explicite et l'autre se fait dans la fonction contient()(contient est donc en O(n)). Une fonction quadratique n'est donc pas optimal, mais c'est la plus rapide que nous ayons su faire.

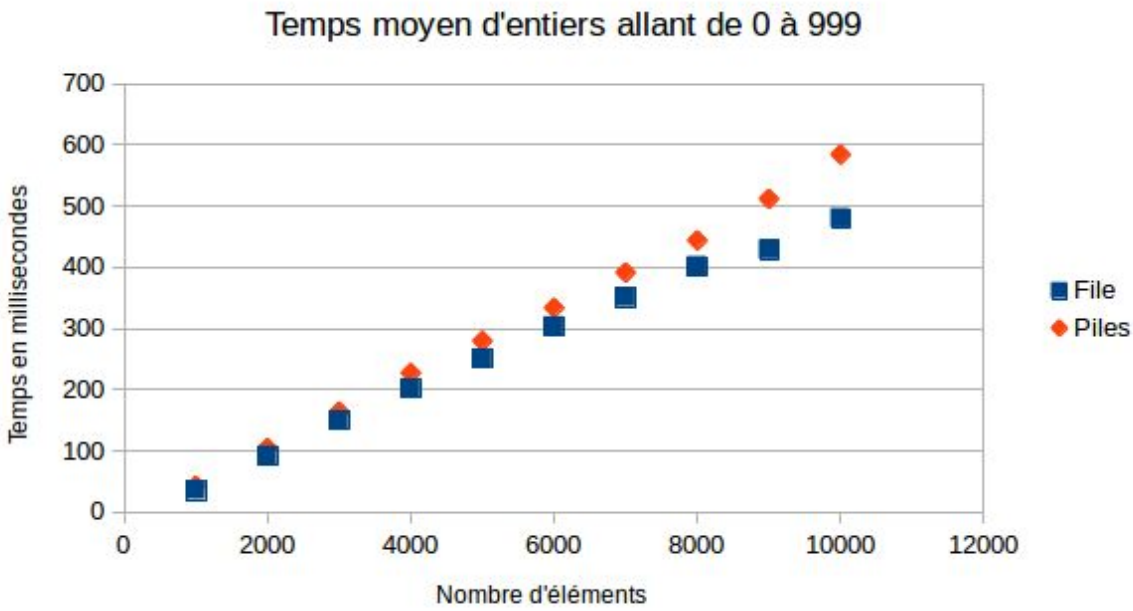
Il fallu ensuite faire une fonction pour créer un anneau remplis aléatoirement, malheureusement cette fonction n'a pu être faite en générique, remplir l'anneau aléatoirement aurait était impossible vu qu'on ne peut gérer l'aléatoire quelque soit le type d'objet. Nous avons donc choisis d'en faire deux, une pour un anneau d'entiers, une pour un anneau de caractères. Nous avons ensuite une fonction testDedoublonne() qui a pour but de tester les deux type d'anneaux remplis aléatoirement. On affiche l'anneau de base et ensuite un autre anneau sans les doublons. Juste pour s'assurer que la fonction dedoublonne() marche correctement.

Pour terminer nous avons notre dernière fonction, la plus importante, calculTempsMoyen() qui calcule le temps moyen d'exécution de la fonction dedoublonne() passé sur des anneaux d'entiers d'une taille variable. On fait 100 tests pour un nombre d'éléments donné pour ensuite faire la moyenne. Nous pensons que 100 test est assez correct ce qui nous permet d'avoir une moyenne la plus représentative possible pour notre anneau. On fait également varier le nombre d'éléments de 1000 à 10 000 avec un pas de 1000. Un autre problème se pose, si on a un anneau de 10 000 entiers allant de 0 à 5 aura-t-on un temps d'exécution de dedoublonne() identique à un anneau de 10 000 entiers allant de 0 à 10 000? On fera donc également varier les bornes des entiers aléatoires afin de constater si le nombre de doublons influence le temps d'exécution de la méthode dedoublonne() ou non.

V. Interprétation des résultats

a) Premier test

Nous calculons le temps d'exécution de dedoublonne() sur nos deux implémentations avec des anneaux d'entiers allant de 0 à 999. On remarque tout d'abord que pour des petites tailles d'anneaux les 2 implémentations ont un temps moyen quasi identiques, et plus la taille grandit plus la différence se creuse. Malgré ça on peut observer une certaine proportionnalité entre la taille de l'anneau et le temps moyen, les 2 nuages de points forment quasiment une droite, sauf peut-être celui pour la File où à partir de 9000 éléments la droite ralentit un peu son ascension. Nous n'avons pu malheureusement essayer des valeurs plus grandes que celles-ci car les PC mettaient vraiment trop de temps pour le faire. On suppose à ce moment précis des tests que les 2piles auraient eu une droite parfaite pour un nombre d'éléments plus grand, et une sorte de courbe qui s'écraserait un peu comme une courbe logarithmique pour la file. L'imprécision des mesures fausse peut-être notre avis pour la file. Et c'est peut-être finalement une droite qui est censé être représentée par les points.



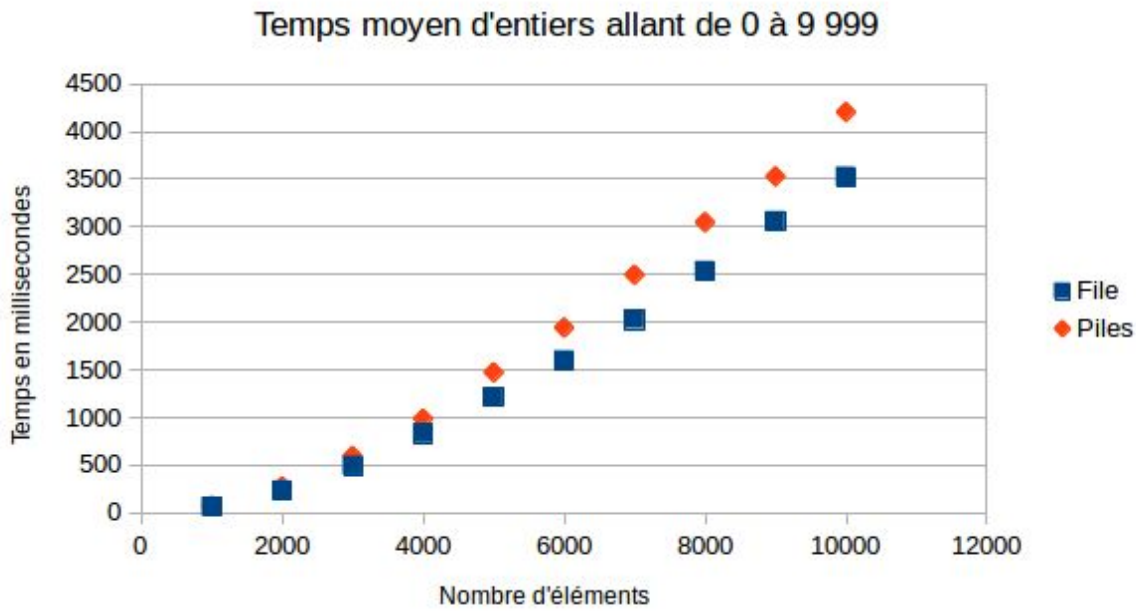
- *Valeurs associées au graphique ci-dessus*

Nombre d'éléments	*Avec une file*	*Avec 2 piles*
1000	*35.6523*	*42.8344*
2000	*91.6673*	*104.364*
3000	*149.58*	*163.935*
4000	*201.979*	*227.031*
5000	*250.859*	*279.523*
6000	*303.165*	*333.654*

7000	*350.771*	*391.418*
8000	*401.637*	*443.872*
9000	*428.444*	*511.744*
10000	*480.064*	*584.261*

b) Second test

Nous essayons maintenant avec des entiers allant de 0 à 9 999. Nous observons directement que les valeurs sont nettement plus grandes que le précédent test. La file est également plus rapide que les 2 piles. Cette fois-ci les points ne forment plus une droite mais une courbe. On peut supposer que plus le nombre d'éléments augmenterait plus l'allure des 2 courbes resteraient pareil.



- *Valeurs associées au graphique ci-dessus*

Nombre d'éléments	*Avec une file*	*Avec 2 piles*
1000	*62.3712*	*74.2415*

2000	*232.824*	*275.17*
3000	*491.218*	*594.714*
4000	*829.53*	*986.429*
5000	*1213.61*	*1473.29*
6000	*1597.75*	*1941.75*
7000	*2026.75*	*2497.33*
8000	*2536.25*	*3049.1*
9000	*3059.2*	*3529.41*
10000	*3525.15*	*4204.26*

VI. Conclusion

Finalement la file est plus rapide que les deux piles, pour le besoin d'une structure de donnée se comportant comme l'anneau étudié on peut donc dire que le choix d'une seul file serait plus judicieux. On souligne également que le temps moyen d'exécution de dedoublonne() pour des anneaux d'entiers varie selons deux facteurs : le nombre d'éléments dans l'anneau et et la borne supérieur pour les entiers de l'anneau. Nous n'avons malheureusement pas un temps constant, plus le nombre d'éléments est grands, plus les entiers sont grands et plus le temps d'exécution est lent. Le choix de la structure de données a donc son importance et il faut donc être apte à choisir celle qui correspond le plus aux besoins d'un projet.

Un des but du projet a aussi était de souligner l'importance de l'efficacité, la robustesse et la rapidité d'un algorithme. Ce sont des choses que nous devons prendre en compte lors de la réalisation d'un projet.

VII. Annexes

Lors de ce rapport nous avons utiliser un code couleur, le vert représente les fonctions/méthodes de classe et le bleu les variables/attributs de classe. Pour plus de clarté nous avons délibérément écris les fonctions sans leurs arguments afin de faciliter la lecture. Par faute de temps nous n'avons pu calculer le coût spatial et les test pour des valeurs plus grandes(anneau d'un million d'éléments etc..).

D'après http://www.cplusplus.com/reference/stack/stack/ et
http://www.cplusplus.com/reference/queue/queue/

Les fonctions push(), pop(), empty() et swap() sont en temps constants. Nous avons pris cela en compte dans nos calculs d'ordre de grandeurs.