Algorithmique et Structure de Données 2
Rapport Projet 1

Valentin Hénique Corentin Chédotal

13 Mars 2016

1 Introduction

Dans le cadre de l'Unité d'Enseignement X4I0030 intitulée "Algorithmique et Structure de Données 2" nous avons été amené à produire un premier projet. Celui-ci consiste en la réalisation d'une structure de données abstraite vue en cours : l'Anneau. Le langage de programmation exigé était le C++. Le but du projet était donc d'implémenter l'Anneau mais d'au moins deux façons différentes afin d'observer les différences que cela engendrerait. Les deux implémentations minimales requises étaient par l'utilisation d'une File et d'une Pile, deux structures de données déja implémentées en C++ respectivement par la `queue` et la `stack`. Ce rapport détaillera comme demandé nos choix derrière les implémentations et fera apparaître les caractéristiques essentielles de chacune d'entre elle afin de faciliter la différenciation des deux méthodes.

2 Implémentations

2.1 L'Anneau par File (`queue`)

La première façon de réaliser la structure de données abstraite de l'Anneau de façon concrète que nous avons fait est celle employant la File ou `queue` en C++ car elle paraissait la plus facile à implémenter.

2.1.1 Fonctionnement

Cette méthode vient donc utiliser le principe de la File FIFO (*First In First Out*) comme container des données qui seront entrées par l'Utilisateur dans l'Anneau. Le fonctionnement comme un Anneau, c'est à dire sans notion de queue ou de tête (qui sont pourtant intrinsèques à la File), est rendu possible en ne permettant l'accès qu'à un point ce qui est déja le cas dans la File (la tête de la File) et la rotation de l'Anneau se fait en faisant en fait tourner l'intégralité des données autour du point d'accès à l'élément courant. Ceci va donner l'illusion de l'absence de tête ou de queue car à chaque fois qu'une donnée arrive à la tête

et doit "reculer" elle sera immédiatement remise en queue. Voici comment les méthodes de l'Anneau ont été implémentées avec la File :

- `Anneau()` : Le constructeur de Anneau fait simplement appel au constructeur de la File (`queue`) afin de construire la file qui servira à stocker les données
- `~Anneau()` : Le destructeur d'Anneau, il est vide car l'Anneau ne fait pas l'objet d'allocations dynamiques nécessitant d'être détruites à la destruction de l'Anneau
- `estVide()` : Cette méthode vérifiant si un Anneau est vide fait simplement appel à la méthode incluse dans la File qui fait déjà exactement ceci pour la `queue`
- `ajoute()` : Cette méthode permettant l'ajout d'un élément à l'Anneau utilise directement la méthode ajoutant un élément à la queue de la File
- `supprime()` : De la même façon que `ajoute()` ici la méthode supprimant un élément de l'Anneau fait appel à la méthode de la File supprimant l'élément en tête de celle-ci (ce qui s'avère d'ailleurs être l'élément courant de l'Anneau)
- `courant()` : Justement, cette méthode retournant l'élément courant de l'Anneau utilise simplement la méthode de `queue` retournant l'élément en tête de la File
- `avance()` : Cette méthode fait tourner l'Anneau en faisant tourner les éléments de la queue de la File vers la tête comme indiqué en 3
- `recule()` : À l'inverse de `avance()` cette méthode va faire tourner l'Anneau en déplaçant tous les éléments de celui-ci autant de fois qu'il y a d'éléments - 1 donnant ainsi l'impression que l'Anneau tournait de la tête vers la queue comme montré en 4

Enfin nous avons décidé de garder le container par défaut pour la `queue`, la `deque`. En effet celle-ci serait optimale lors d'ajout en fin ou en début de container alors que son homologue la `list` serait préférée lors d'ajout au milieu du container ce qui n'est pas notre cas ici.

2.1.2 Complexité Temporelle

Ci-après se trouve le tableau des ordres de grandeur des complexités temporelles des diverses méthodes de l'Anneau implémenté à l'aide de la File.

Méthodes	Complexité
`Anneau()`	$\Theta(1)$
`~Anneau()`	$\Theta(1)$
`estVide()`	$\Theta(1)$
`ajoute()`	$\Theta(1)$
`supprime()`	$\Theta(1)$
`courant()`	$\Theta(1)$
`avance()`	$\Theta(1)$
`recule()`	$\Theta(n)$

TABLE 1 – Complexité Temporelle par File

Après il est évident que les ordres de grandeurs des méthodes ne sont finalement que des indicateurs très abstraits et théoriques du fonctionnement d'une implémentation. Ainsi se trouve ci-dessous comme demandé une étude du temps d'exécution de la fonction `dedoublonne()` présentée en 2.3 en fonction du nombre de valeurs stockées dans l'Anneau.

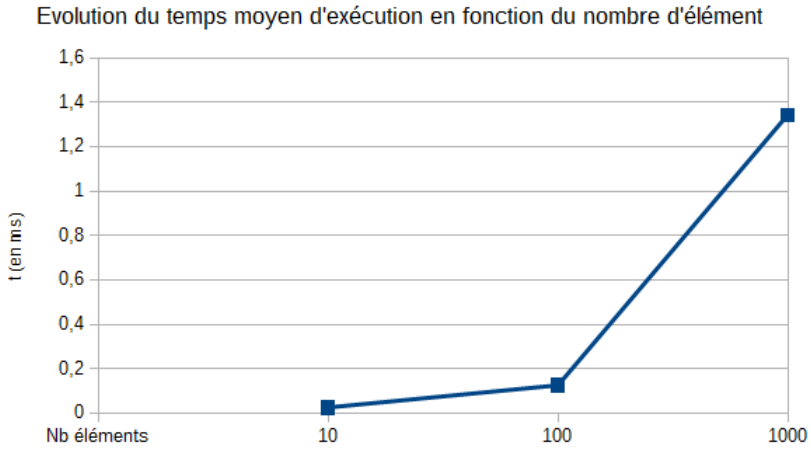

FIGURE 1 – Évolution du temps d'exécution par File

2.1.3 Encombrement Mémoire

Enfin l'encombrement mémoire de l'Anneau par File reste relativement limité. En effet le seul attribut de la classe Anneau consiste ici en une seule et unique File (`queue`). Or l'empreinte mémoire de celle-ci est plus ou moins proportionnelle au nombre d'éléments de la File, le facteur de proportionnalité dépendant du type de container employé. Ainsi il est tout à fait possible de faire un parallèle avec les complexités temporelles et donner un ordre de grandeur de l'encombrement mémoire de l'Anneau implémenté. Il est donc possible de

dire que l'encombrement mémoire est linéaire en fonction du nombre d'éléments dans l'Anneau.

2.2 L'Anneau par Pile (`stack`)

La deuxième façon d'implémenter de façon concrète la structure de donnée abstraite de l'Anneau que nous avons réalisé est celle employant la structure de Pile ou `stack` en C++. Légèrement moins facile à implémenter car le principe même de la Pile force à faire une petite gymnastique mentale afin de garder l'ordre intact lors des déplacement. Enfin comme demandé dans le sujet il est employé non pas une seule mais bien deux Piles dans l'implémentation.

2.2.1 Fonctionnement

Cette méthode vient donc employer deux Piles LIFO (*Last In First Out*) comme contenants des données placées par l'Utilisateur dans l'Anneau. En réalité une seule des deux Piles contiendra les données de manière générale mais de part le fonctionnement même de la Pile (l'ajout et suppression ne pouvant se faire qu'en tête) il était essentiel d'avoir une Pile de plus stockant les données de façon à pouvoir les faire "tourner" dans l'Anneau sans changer l'ordre de celles-ci. Le fonctionnement comme un Anneau est justement rendu possible en ayant qu'un seul point d'accès, l'élément courant (qui ici est la tête de la Pile) et la rotation de l'Anneau est rendue possible par une série d'empilement-dépilement de la Pile principale à celle temporaire. Cette série d'opérations est responsable de 'illusion d'une absence de tête et de queue pour l'Utilisateur qui à l'emploi de l'Anneau ne verra que ses données "tournant" par rapport au courant qui est "fixe".
Les méthodes de l'Anneau sont presque toute implémentées de la même façon que par File (2.1.1), il ne sera donc détaillé ci après que les méthodes dont l'implémentation diffère vraiment de part l'utilisation de Piles plutôt que d'une File.

- `ajoute()` : La méthode permettant d'ajouter un élément dans l'Anneau nécessite ici d'abord d'empiler tous les éléments de l'Anneau dans la Pile temporaire avant d'ajouter le nouvelle élément puis de tout réempiler sur la Pile principale par dessus le nouvel élément ; ceci afin de conserver l'ordre des données
- `avance()` : Cette méthode faisant avancer les données en tournant l'Anneau nécessite à nouveau des dépilement et empilement vers et depuis la Pile temporaire avec le cas particulier de l'élément courant qui doit être mis de la tête de la Pile principale à la base de celle-ci donnant l'impression d'absence de tête et queue. La rotation se fait dans ce cas de la base vers la tête comme montré en 5
- `recule()` : À l'inverse de `avance()` le principe reste le même avec dépilement et empilement seulement là c'est la tête de la Pile temporaire soit la base de la Pile standard que l'on essaye de déplacer en tête de la Pile

de stockage ; ainsi la rotation va se faire de la tête vers la base comme visible en 6

Enfin, suivant le même raisonnement que pour l'implémentation par File nous avons décidé de conserver la `deque` comme container des données de la Pile car elle est plus adaptée aux accès aux données que nous faisons que son alternative, la `list`.

2.2.2 Complexité Temporelle

Ci-dessous se trouve un tableau des ordres de grandeur des complexités temporelles des diverses méthodes de l'Anneau implémentés à l'aide de deux Piles.

Méthodes	Complexité
`Anneau()`	$\Theta(1)$
`~Anneau()`	$\Theta(1)$
`estVide()`	$\Theta(1)$
`ajoute()`	$\Theta(n)$
`supprime()`	$\Theta(1)$
`courant()`	$\Theta(1)$
`avance()`	$\Theta(n)$
`recule()`	$\Theta(n)$

TABLE 2 – Complexité Temporelle par Pile

De la même façon que pour la partie précédente sur l'implémentation par File il est évident que les ordres de grandeurs des méthodes ne sont que des représentations abstraites et théoriques du fonctionnement d'un algorithme. Il est toujours très intéressant d'évaluer le fonctionnement d'une implémentation dans un cas concret. Ce faisant vous trouverez ci-dessous, et comme demandé dans le sujet, une étude du temps d'exécution de la fonction `dedoublonne()` présentée en 2.3 en fonction du nombre de valeurs stockées dans l'Anneau.

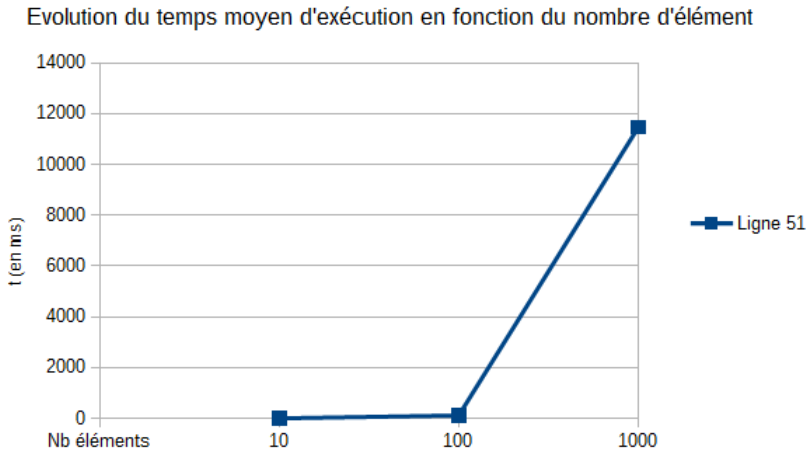

FIGURE 2 – Évolution du temps d'exécution par Pile

2.2.3 Encombrement Mémoire

Enfin de la même manière qu'il est possible de donner des ordres de grandeurs de complexités temporelles il est possible de donner des ordres de grandeur de l'encombrement mémoire d'une implémentation. Or ici les deux seuls attributs sont les deux Piles or l'encombrement mémoire d'une Pile est plus ou moins proportionnel au nombre d'éléments dans celle-ci. Le facteur de proportionnalité dépendant du container de données employé dans la Pile en question. Il se trouve que dans le cas général une seule des deux Piles n'est remplie cependant il n'en reste pas moins possible de s'arrêter là et dire que l'encombrement mémoire est purement et simplement linéaire dans ce cas. En effet lors d'ajout ou de rotation de l'Anneau les données de la Pile principale sont stockées dans la Pile temporaire puis supprimées de la Pile principale et puis vice-versa. Ainsi il y a deux moments pour chacune de ces opérations durant lesquels les deux Piles contiennent l'intégralité des données. L'ordre de grandeur ne change pas vraiment, il s'agit toujours d'une forme linéaire mais il s'agit à ce moment là d'un $2n$ plutôt que d'un n. Or si le temps est fondamentalement infini (bien qu'il est évident qu'on ne restera pas devant un programme un siècle durant à attendre un résultat) l'espace mémoire machine ne l'est pas. Il paraissait donc important de noter cette nuance dans le cas d'emploi d'Anneau stockant une très grande quantité de données lourdes.

2.3 La fonction `dedoublonne()`

Cette fonction, réalisée comme demandé dans le sujet, sert principalement de banc d'essai des implémentations de la structure de données Anneau. Prenant un Anneau rempli d'éléments en argument elle retournera un Anneau privé des

éventuels éléments présents en double ou plus dans la structure retirant donc les doublons d'où son nom. Elle fait cela en déterminant d'abord le compte des éléments à l'intérieur de l'Anneau puis à l'aide de deux boucles imbriquées elle parcourera l'Anneau, pour chaque élément de celui-ci, à la recherche de doublons de l'élément et le cas échéant les supprimera de la structure. Cependant il est très important de noter que pour que `dedoublonne()` fonctionne correctement il faut que le type stocké dans l'Anneau donné à la fonction ait un test d'égalité fonctionnel sinon la fonctio peut ne pas agir correctement.

3 Conclusion

Pour conclure sur ces deux implémentations nous nous rendons très vite compte que l'implémentation par File est bien plus adaptée pour la concrétisation de la structure de données abstraite Anneau que celle employant deux Piles. En effet d'une façon générale la complexité temporelle est bien meilleure par l'utilisation de File que par Piles. En plus de cela l'encombrement mémoire est stable dans le temps quelque soit les opérations faite sur l'Anneau par File alors que certaines opérations sur l'Anneau implémenté par deux Piles voient l'encombrement mémoire doubler pendant un certain temps. Ainsi en plus d'être fixe l'encombrement de la File est égal ou moindre à celui des deux Piles. Enfin et bien que ce ne soit pas une raison à elle seule de choisir l'une ou l'autre méthode il est appréciable de noter que l'implémentation par File en plus d'être la plus efficace était à nos yeux la plus facile à implémenter.

Finalement, pour conclure sur ce projet. Nous sommes plutôt heureux du résultat et ce projet nous a permis aussi de découvrir *Git* nous permettant une bien plus grande aisance de travail à distance. Cela s'est révélé particulièrement important car malgré nos efforts nous avons été poussé par le temps et bien que jamais en retard la date butoir a été à l'origine de quelques stress.

Annexe

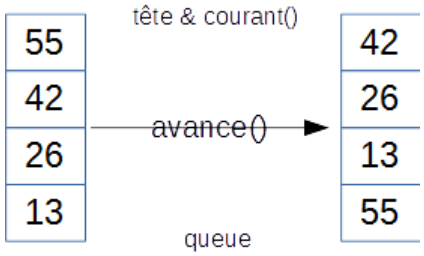

FIGURE 3 – Évolution d'une File donnée suite à `avance()`

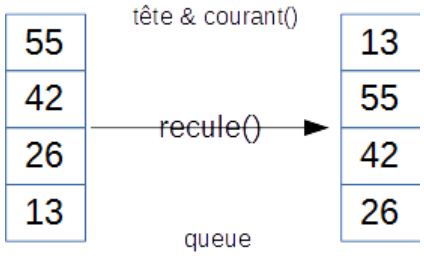

FIGURE 4 – Évolution d'une File donnée suite à `recule()`

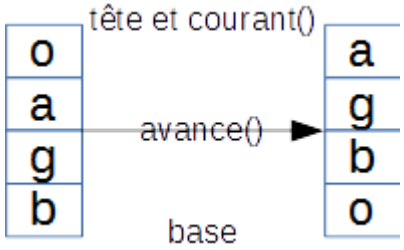

FIGURE 5 – Évolution d'une Pile donnée suite à `avance()`

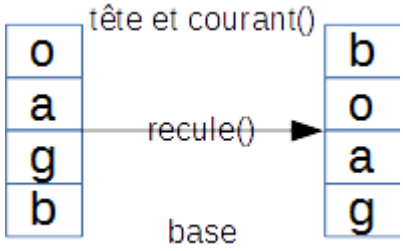

FIGURE 6 – Évolution d'une Pile donnée suite à `recule()`

Table des matières