UNIVERSITÉ DE SHERBROOKE
DÉPARTEMENT D'INFORMATIQUE

IFT 339

Travail pratique#2 : Implémentation d'une liste avec tableau circulaire

---

Dans ce travail, vous allez implémenter une liste avec un tableau circulaire, en imitant la classe deque de la Standard Library (SL). Le TP est une occasion de mettre en pratique l'utilisateur de pointeurs et de leur arithmétique. Quelques modalités :

- Ce travail compte pour **5.5 %** de la session.
- La date de remise de ce travail est le **lundi 30 septembre**. Notez que le devoir 3 vous sera donné quelques jours avant cette date, donc vous êtes invité(e)s à ne pas attendre à la dernière minute.
  Vous devez remettre sur `https://turnin.dinf.usherbrooke.ca/` les fichiers générés au cours de ce travail.
- Il n'y a **pas de script de correction automatique.** Vous êtes responsables de générer vos propres cas de test. Le fichier main.cpp donné contient quelques tests, mais il ne contient pas tous les cas possibles.

Partie 1 : liste via tableau circulaire

Vous devez implémenter le type de données abstrait *liste*, qui doit supporter les opérations suivantes :
- créer une nouvelle liste.
- obtenir/modifier le i-ème élément.
- ajouter/retirer un élément à la fin (push_back, pop_back).
- ajouter/retirer un élément au début (push_front, pop_front).
- obtenir la dimension (le nombre d'éléments de l'utilisateur stockés).

Votre implémentation stocke les éléments dans un tableau circulaire et utilise quatre pointeurs :
debut_cap : pointe vers le premier élément du tableau de stockage.
fin_cap : pointe vers le dernier élément du tableau de stockage.
debut_elem : pointe vers le premier élément de l'utilisateur.
fin_elem : pointe vers le dernier élément de l'utilisateur.

La classe s'appelle deque pour *doubly-ended queue*, voulant dire qu'elle est efficace pour l'ajout ou suppression autant au début qu'à la fin.

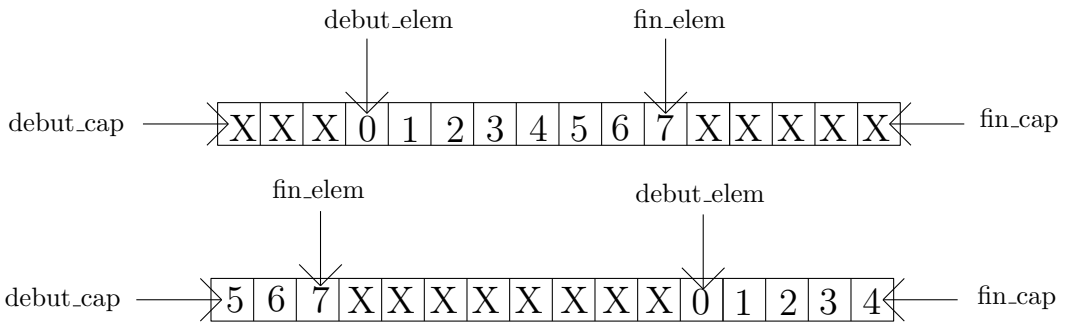

FIGURE 1 – Deux états possibles du tableau circulaire. Les nombres représentent l'ordre des éléments du point de vue utilisateur, les X représentent des cases réservées strictement pour des insertions futures et dont le contenu est indéterminé.

Dans un tableau circulaire, le premier élément de l'utilisateur peut se trouver n'importe où dabns le tableau de stockage, et il en est de même pour le dernier élément. Dépendamment de la séquence de push et pop effectuée, les deux cas de figure illustrés dans la figure sont possibles et doivent être gérés (voir le main.cpp fourni pour des exemples concrets).

Afin de réaliser la liste, Vous devez implémenter les fonctions C++ suivantes :

constructeurs : le constructeur par défaut créé une liste vide, mais avec une capacité initiale plus grande que 0. À vous de choisir comment représenter une liste vide. Un autre constructeur reçoit un entier n et créé une liste de n éléments. Vous devez aussi implémenter le constructeur par copie.

operator= : surcharge de l'affectateur, qui copie le deque passé et retourne une référence vers le deque courant.

destructeur : nettoie la mémoire allouée dynamiquement.

size : retourne le nombre d'éléments stockés par l'utilisateur. Demande u peu d'arithmétique de pointeurs.

reserve(n) : affecte la capacité à n. À l'interne, il est attendu que le tableau soit recréé peu importe n. Les éléments de 0 à $\min(n-1, size()-1)$ doivent rester présent. À vous de déterminer comment réaffecter les quatre pointeurs.

resize(n) : affecte le nombre d'éléments utilisateur à n. Les éléments de 0 à $\min(n-1, size()-1)$ doivent rester présent. Si n dépasse la capacité actuelle, vous devez appeler reserve(2 * n).

push_back(val) : ajoute val en fin. Si la capacité le permet, on utilise la prochaine case disponible après fin_elem, en revenant au début du tableau si nécessaire. Sinon, il faut appeler reserve avec le double de la capacité actuelle.

push_front(val) : ajoute val en début. Si la capacité le permet, on utilise la case disponible avant debut_elem. Sinon, il faut appeler reserve avec le double de la capacité actuelle.

pop_back : retire le dernier élément.

pop_front : retire le premier élément.

operator[](i) : retourne une référence vers le i-ème élément de l'utilisateur. Vous devez aussi implémenter la version const, et je vais tolérer un copier-coller ici (le at n'est pas demandé).

Des méthodes empty, front, back, swap sont déjà implémentée pour vous. Vous pouvez ajouter des méthodes privées aux classes fournies, au besoin.

Un fichier exemple main.cpp vous est aussi fourni, ainsi que la sortie attendue en l'exécutant une fois votre implémentation complétée. Notez que le fait que vous obteniez la sortie attendue n'implique pas une note de 100% : vous êtes responsable de fournir un code libre de bugs, même s'ils ne sont pas identifés par le main.cpp fourni.

Partie 2 : historique de requêtes

Vous êtes responsable d'un serveur qui gère des requêtes avec des priorités différentes. Les requêtes reçues sont ajoutées à une liste et, lorsque le serveur a le temps, il gère la prochaine requête ayant la priorité minimum (plus bas = plus prioriaire). On vous donne le journal des événements (souvent appelé le *log*) qui ont eu lieu dans l'ordre, et vous devez reconstruire l'historique des requêtes gérées. Le journal contient deux types d'événements :

add_request(id, p) : survient lorsque le serveur reçoit une requête, avec id l'identifiant de la requête et p la priorité, et l'ajoute à la liste des requêtes non-gérées. Ici, p est un entier entre 1 et 10 (et donc il y a $O(1)$ valeurs possibles pour p).

process_next() : survient lorsque le serveur gère la requête non-gérée avec la priorité minimum. Si plusieurs requêtes ont la priorité minimum, celle qui est arrivée le plus tôt est choisie. Cette requête est retirée de la liste.

Vous devez implémenter la fonction get_histo, qui reçoit une liste de n objets de type Evenement, et retourne la liste des identifiants de requêtes dans l'ordre dans lequel elles ont été gérées. Si des requêtes restent non-gérées à la fin du journal, elles ne seront pas dans votre sortie. Vous devriez viser un temps total de $O(n)$.

Vous trouverez en Annexe à la dernière page un exemple d'instance et de sortie attendue.

Évaluation

Respect des spécifications (45 points) : votre code doit effectuer les opérations mentionnées ci-haut de façon correcte. Ceci inclut l'obtention de la sortie attendue ainsi que l'absence d'autres possibles bugs.

Qualité du code (45 points) : assurez-vous de bien nettoyer la mémoire allouée dynamiquement, de ne pas avoir de code inutile, surtout d'éviter la répétition de code lorsque possible, et de respecter le principe d'encapsulation.

Respect des normes de programmation (10 points) : utilisez soit les normes affichées sur le site du cours, ou bien celles des notes de cours, mais soyez constant(e)s.

Remise du travail

Vous devez remettre les fichiers deque.h et main.cpp. Ne remettez pas d'autre fichier ni surtout d'exécutable. Il ne doit y avoir qu'une seule remise par équipe, à partir d'un seul CIP. Les noms des co-équipiers doivent être clairement indiqués en entête de chaque fichier soumis. Notez qu'il n'y a pas de script de correction automatique - votre code sera inspecté à la correction.

Annexe

Un exemple d'entrée pour la fonction get_histo. Colonne gauche : liste des évnements du journal, dans l'ordre de haut en bas. Colonne milieu : liste des requêtes qui n'ont pas été gérées, affichées sous la forme (id, p). Colone droite : affiche les requêtes au moment où elles sont gérées.

Note : même si la colonne milieu semble suggérer des stocker les requêtes dans l'ordre de réception, je recommande de stocker les requêtes différemment dans votre fonction.

Événement	**À gérer, paires de la forme (id, p)**	**Note**
add_request(1, 5)	(1, 5)	
add_request(2, 5)	(1, 5), (2, 5)	
add_request(3, 1)	(1, 5), (2, 5), (3, 1)	
process_next()	(1, 5), (2, 5)	(3, 1) gérée
add_request(4, 10)	(1, 5), (2, 5), (4, 10)	
process_next()	(2, 5), (4, 10)	(1, 5) gérée
add_request(5, 4)	(2, 5), (4, 10), (5, 4)	
add_request(6, 2)	(2, 5), (4, 10), (5, 4), (6, 2)	
add_request(7, 4)	(2, 5), (4, 10), (5, 4), (6, 2), (7, 4)	
process_next()	(2, 5), (4, 10), (5, 4), (7, 4)	(6, 2) gérée
process_next()	(2, 5), (4, 10), (7, 4)	(5, 4) gérée
process_next()	(2, 5), (4, 10)	(7, 4) gérée

La sortie de votre fonction devrait être 3, 1, 6, 5, 7. Puisque les requêtes 2 et 4 demeurent non-gérées, elles ne sont pas listées dans votre sortie.