TP6 : Files et listes

MP2I Lycée Pierre de Fermat

Le but de cette séance est d'implémenter les deux versions des files que l'on a vu en cours, puis d'implémenter une structure proche des vecteur.

Créez pour ce TP deux dossiers : "file/" et "liste/". Pour chacune des deux SDA, tout le code écrit se trouvera dans le dossier correspondant.

Ce TP est assez long, on travaillera dessus pendant 2 séances. Il sera donc à rendre pour le ***dimanche 18 decembre à 22h*** . Les dernières questions portent sur de l'étude de performances, et peuvent être réalisées en ***binômes ou trinômes*** si vous le désirez. Dans ce cas, vous devez indiquer la composition de votre groupe dans votre rendu. De plus, les réponses aux questions concernées et les éventuels documents produits devront se trouver dans un seul des rendus du groupe (le reste du TP est individuel), et les autres membres indiqueront chez qui ces réponses se trouvent.

Comme d'habitude, vous devez commenter votre code : Un commentaire de documentation pour chaque fonction (pour les fonctions dont l'en-tête est dans un fichier .h, pas besoin de réécrire le commentaire dans le fichier .c), et des commentaires à l'intérieur du code lorsqu'une étape du code mérite une explication.

Vous trouverez sur Cahier de Prépa une archive à télécharger pour ce TP.

Files

On rappelle la spécification des files. Une file sert à stocker des éléments dans un ordre donné. On peut rajouter des éléments à la queue de la file (enfilage) et supprimer des éléments à la tête de la file (défilage), de telle sorte que les éléments sont défilés dans le même ordre qu'ils ont été enfilé, selon un schéma "premier arrivé premier sorti". Les 4 opérations sont :

— Créer d'une file vide

— Enfiler un élément à la queue de la file

— Défiler l'élément en tête de file

— Déterminer si une file est vide

Question 1. Ouvrez le fichier "file.h" de l'archive, qui donne les signatures des différentes opérations sur les files. Pour chaque fonction, écrivez un commentaire de documentation. Rappel : le nom de chaque paramètre doit apparaître dans le commentaire !

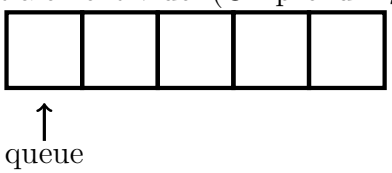
Comme pour les piles, on implémente, en plus des 4 opérations des fonctions de libération de mémoire et d'affichage.

Question 2. Écrivez dans un fichier "test.c" un programme permettant de tester les files, en créant et modifiant une file. Ce fichier devra commencer par inclure "file.h".

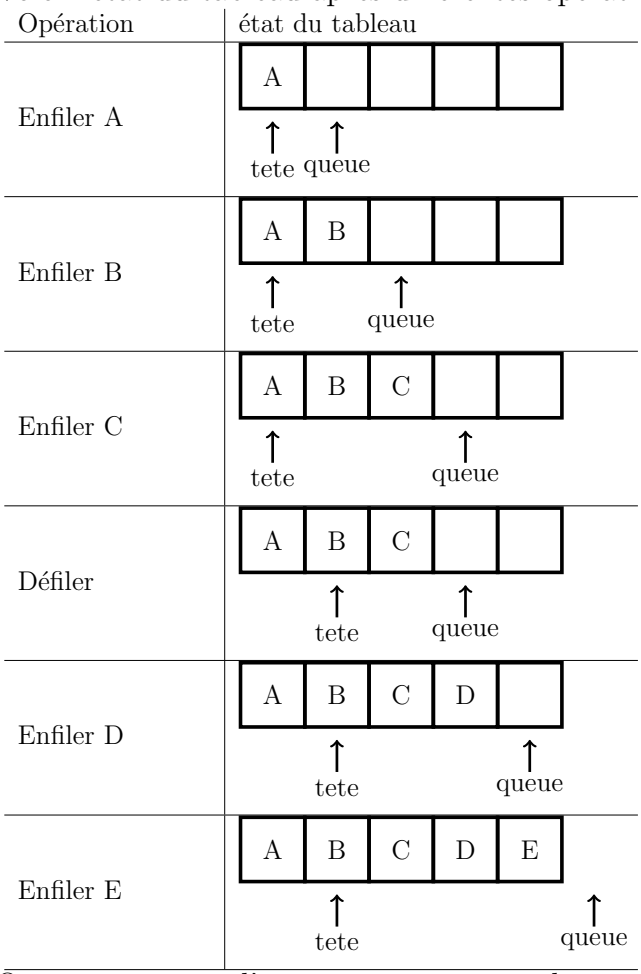
Implémentation par tableau

Si vous avez commencé cette implémentation la semaine dernière, vous pouvez reprendre votre code, il n'y aura que quelques lignes à changer.

L'idée est d'avoir un indice pour la queue, et de stocker le nombre d'éléments, ce qui permet de retrouver la position de la tête de file facilement. Par exemple, on considère une file F initialement vide. (On prend $N_{max} = 5$) :



Voici l'état du tableau après différentes opérations sur la file F :



On remarque que l'on peut se retrouver dans une situation où l'on ne peut plus enfiler parce que l'on a atteint la dernière case du tableau, même si la file contient moins de N_{max} éléments. Pour résoudre ce problème, on utilise un tableau ***circulaire***, c'est à dire que l'on manipule nos indices modulo la taille du tableau.

On utilise pour cette implémentation le type suivant :

```
1 #define Nmax 10000
2 typedef struct file_{
3   int queue; // prochaine case où enfiler un élément
4   int nb_elem; // nombre actuel d'éléments dans la file
5   T* tab; // tableau permettant de stocker les éléments.
6 } FIFO;
```

Voyons ce qu'implique le fait que le tableau est circulaire sur la fonction d'enfilage :

```
1 void enfiler(FIFO* f, T x){
2   assert(f->nb_elem < Nmax);
3
4   f->tab[f->queue] = x;
5   f->queue = (f->queue+1) % Nmax;
6   f->nb_elem++;
7 }
```

Il suffit donc juste de considérer les indices modulo N_{max}

Question 3. Dans un fichier "file_tab.c", recopiez la définition du type ci-dessus, et implémentez chacune des fonctions présentes dans l'en-tête "file.h".

Question 4. Compilez votre programme en utilisant main.c et file_tab.c pour tester votre implémentation. Vérifiez avec ***valgrind*** qu'il n'y a pas d'erreurs ou de fuites mémoire.

Implémentation par liste chaînée

Si vous avez fait l'implémentation avec les listes doublement chaînées la semaine dernière vous pouvez reprendre le même code, il n'y a que quelques lignes à enlever pour obtenir l'implémentation par listes simplement chaînées.

Comme pour les piles On utilise donc une liste ***doublement chaînée***, où chaque maillon connaît son successeur. Les flèches des successeurs vont dans le sens de la tête vers la queue. Autrement dit, le maillon de queue aura toujours son successeur égal à NULL. On utilise donc les types suivants :

```
1 typedef struct maillon {
2   T elem;
3   struct maillon* suivant;
4 } MAILLON;
5
6 typedef struct file_{
7   MAILLON* tete; //pointeur vers le prochain élément à défiler
8   MAILLON* queue; //pointeur vers le dernier élément ajouté
9 } FIFO;
```

Question 5. En vous inspirant de la fonction d'affichage d'une pile, écrivez la fonction affiche_file .

Question 6. Écrivez la fonction de libération de mémoire d'une file

Question 7. Écrivez la fonction de création d'une file vide

On considère maintenant l'opération d'enfilage. Les étapes sont :

1. Récupérer le maillon de queue actuel q
2. Créer un nouveau maillon m contenant l'élément à enfiler
3. Raccorder m à q
4. Indiquer que la queue de la file est maintenant m.
5. Si la file était vide, indiquer également que la tête de file est maintenant m.

Question 8. Implémentez la fonction enfiler , et testez là.

Question 9. Une fois que cette fonction marche comme vous voulez, implémentez les opérations restantes.

Question 10. Compilez à nouveau votre programme, cette fois ci en utilisant main.c et file_chaine.c. N'oubliez pas de vérifier votre programme avec valgrind !

Listes

Une liste est une structure de données proche du vecteur, mais ne permettant pas un accès à une case d'indice i donné. Les opérations sont :

Créer une liste vide	LISTE* creer_liste()
Déterminer le nombre d'éléments d'une liste	int taille_liste (LISTE* l)
Renvoyer la place en tête de liste	PLACE* tete(LISTE* l)
Renvoyer la place en queue de liste	PLACE* queue(LISTE* l)
Renvoyer l'élément écrit à une place	T contenu(PLACE* p)
Modifier l'élément écrit à une place	void modifier(PLACE* p, T x)
Accéder à la place suivante d'une place	PLACE* suivant(PLACE* p)
Accéder à la place précédente d'une place	PLACE* precedent(PLACE* p)
Insérer un élément avant une place	void inserer (LISTE* l, PLACE* p, T x)
Supprimer une place	void supprimer(LISTE* l, PLACE* p)
Ajouter un élément à la fin d'une liste	void ajouter(LISTE* l, T x)

Comme il n'y a pas d'opération permettant d'accéder à un indice particulier, on implémente directement les listes par listes doublement chaînées.

On utilise la structure suivante :

```
1  typedef struct place {
2    T elem;
3    struct place* suiv;
4    struct place* prec;
5  } PLACE;
6
7  typedef struct liste{
8    PLACE* tete; //pointeur vers le premier élément
9    PLACE* queue; //pointeur vers le dernier élément de la liste
10   int taille;
11 } LISTE;
```

L'opération la plus complexe est l'insertion avant une place p_1. Les étapes de cette opérations sont :

1. Créer une nouvelle place p pour stocker l'élément à insérer.
2. Récupérer le prédecesseur de p_1, notons le p_0. Attention, p_0 peut être NULL si p_1 est la tête de liste.
3. Mettre à jour le successeur de p_0 et le prédecesseur de p_1.
4. Mettre à jour le successeur et le prédecesseur de p.
5. Si p_1 était la tête de liste, indiquer que p est maintenant la tête de liste.

Question 1. Téléchargez le fichier "liste.h" sur CdP et commentez les différentes fonctions.

Question 2. Modifiez le fichier pour que les éléments des listes soient des int.

Question 3. Implémentez les fonctions dans un fichier "liste.c".

Question 4. Testez toutes vos fonctions dans un fichier "test_liste.c". Vérifiez avec valgrind qu'il n'y a pas de problème d'implémentation

On implémente maintenant quelques fonctions sur les listes. Créez un nouveau fichier "manip_liste.c". Pour chacune des fonctions suivantes, implémentez les dans ce fichier, et ajoutez dans le main plusieurs tests pour ces fonctions.

Question 5. Écrire une fonction PLACE* recherche(LISTE* l, int x) renvoyant la première place dans l contenant x, ou NULL si x n'apparaît pas dans l.

Question 6. Écrire une fonction void supprimer_occs(LISTE* l, int x) supprimant ***toutes*** les occurences de x dans l. Cette fonction doit être en $\mathcal{O}(n)$ où n est la taille de la liste, donc vous ne pouvez pas réutiliser la fonction de recherche précédente.

Dans les questions suivantes, on implémente un nouvel algorithme de tri : le ***tri rapide***. Créez un nouveau fichier "tri_rapide.c".

Question 7. Écrire une fonction void concatenation(LISTE** ls, int n) qui prend en entrée un tableau ls de n listes, et qui renvoie une liste contenant la concaténation de toutes les listes. Par exemple, la concaténation de $[1, 2, 4], [4, 8, 9], [22, 15]$ sera $[1, 2, 4, 4, 8, 9, 22, 15]$.

Question 8. Écrire une fonction LISTE** separer(LISTE* l, int x) qui renvoie un tableau de trois listes, $[l_<, l_=, l_>]$, avec :

- $l_<$ contient les éléments de l étant strictement inférieurs à x
- $l_=$ contient les éléments de l étant égaux à x
- $l_>$ contient les éléments de l étant strictement supérieurs à x

Ainsi, la somme des tailles des trois listes doit être la taille de la l.

Ces deux fonctions permettent de réaliser une étape du tri rapide :

Algorithme 1 : Tri rapide

Entrée(s) : L une liste
Sortie(s) : L' une copie triée de L

```
1 si L non vide alors
2     x ← L.tête();
3     L_<, L_=, L_> ← séparer(L);
4     Trier récursivement L_< et L_>;
5     retourner concatenation(L_<, L_=, L_>)
6 sinon
7     retourner liste_vide()
```

Lors de l'implémentation, il faudra faire très attention aux fuites mémoires : toute mémoire qui est réservée doit être libérée si elle n'est pas renvoyée. De plus, il faut bien prendre garde à ne pas écraser le contenu d'un pointeur si celui-ci pointe vers une zone non-libérée du tas.

Question 9. Implémentez une fonction qui génère des listes aléatoires de taille n donnée, à valeurs dans $[\![0, 100]\!]$.

Question 10. Implémentez une fonction qui vérifie si une liste est triée.

Question 11. Implémentez le tri rapide, et testez le avec les fonctions précédentes. Utilisez valgrind pour détecter d'éventuelles erreurs.

Question 12. Implémentez une fonction void insertion_triee (LISTE* l, int x) qui insère x dans l en supposant que l soit triée dans l'ordre croissant. L'insertion sera plus simple en partant de la tête.

Question 13. A l'aide de cette fonction, écrivez une fonction implémentant le tri par insertion LISTE* tri_insertion(LISTE* l) et vérifiez qu'il renvoie la même liste que le tri rapide.

C'est l'heure des ⋆***compétences numériques***⋆ ! On veut maintenant comparer les deux algorithmes en terme de performance. Le reste de cette partie peut être traité en ***groupes*** de 2 ou 3. Cependant, même si chaque membre ne code pas à part égale, vous devez vous assurer que tout le groupe comprend ce qui est fait et participe à la réflexion. Vous rendrez un petit rapport répondant aux questions de cette partie (soit dans le fichier de réponse soit dans un document à part) ainsi que le code (C, python, bash, autre) que vous aurez écrit pour obtenir et formatter vos résultats. Ce code devra être commenté ou expliqué rapidement dans votre rapport.

Dans un terminal, la commande ***time*** permet de chronométrer l'exécution d'un programme. Par exemple, pour un programme "compare_tris" :

time ./compare_tris

affiche le rapport d'information suivant :

```
real    0m1.052s
user    0m1.042s
sys     0m0.010s
```

Le temps qui nous intéresse est celui indiqué par "user". Pour automatiser la récolte d'information, le fichier "perf.py" contient une fonction temps_exec(cmd) qui renvoie le nombre de millisecondes nécessaire à l'exécution de la commande cmd.

Question 14. Écrivez un programme C prenant en argument deux entiers n et k, et un mot, qui peut être soit "RAPIDE" soit "INSERTION", et qui génère et trie une liste de taille n, k fois d'affilées, avec l'algorithme choisi. Vérifiez qu'il ne génère pas de fuite mémoire.

Le paramètre k va permettre de moyenner pour obtenir des valeurs plus précises. On veut obtenir des données pour $n \in \{200, 400, 600, \ldots, 10000\}$, en faisant 8 essais indépendants pour chaque n. Pour chaque essai, afin d'obtenir un temps plus précis, on prend k grand et on divise le temps mesuré par k. Afin d'avoir des mesures précise sans avoir des temps d'exécution trop longs, vous prendrez les valeurs de k suivantes :

- Pour le tri rapide : $k = 1000$ pour $n < 2000$, $k = 500$ pour $2000 \leq n < 5000$ et $k = 200$ sinon
- Pour le tri par insertion : $k = 500$ pour $n < 1000$, $k = 100$ pour $1000 \leq n < 2000$ et $k = 10$ sinon.

Question 1. Pour chaque algorithme et pour chaque $n \in \{200, 400, 600, \ldots, 10000\}$, avec le k correspondant, faire 8 essais indépendants et noter les temps obtenus. Le traitement total des données devrait prendre entre 30 minutes et une heure avec les paramètres donnés, vous avez le temps pour une sieste.

Question 2. Tracez les deux nuages de points correspondants, un pour chaque algorithme, montrant le temps d'exécution de chacun en fonction de la taille du tableau.

Question 3. Commentez rapidement les graphiques obtenus : quel tri semble meilleur ? Que semblent être les complexités asymptotiques ?