TD : Evaluation de performances - Liste

Dans le cours S1+S2 Algorithmique et structures de données, vous avez étudié les deux implémentations standard d'une liste : à l'aide d'un tableau au S1, à l'aide d'éléments chainés au S2. Vous avez également étudié comment construire très simplement une file (FIFO) et une pile (LIFO) à partir de l'interface d'une liste, indépendamment de son implémentation.

Le but de ce TD est double :

1/ Développer un service de gestion de listes, files et piles opérationnel, complet et robuste, à partir des corrigés du cours Algorithmique et structures de données. Le service permet de créer dynamiquement plusieurs de ces structures, de les manipuler et de les supprimer quand elles ne servent plus.

2/ Analyser les performances cpu et mémoire des deux implémentations d'une liste, dans le cadre des deux utilisations file et pile. Cela fait un total 2 implémentations x 2 utilisations = 4 configurations différentes à analyser.

Table des matières

1 Développement

Pour simplifier, on ne considère dans la suite que des listes, files et piles de *chaines de caractères*. On ne cherche pas à développer des structures de données génériques, permettant de stocker des éléments de type quelconque.

1.1 Service de gestion de listes

Point de vocabulaire : Une liste est une collection ordonnée d'*élément*s, chacun étant repéré par sa *position* (ou numéro d'ordre : 0, 1, 2, ...) dans la liste. La *taille* de la liste est le nombre d'éléments qu'elle contient. Sa *capacité* est le nombre maximal d'éléments qu'elle peut contenir. On peut définir une liste de capacité infinie, i.e. dont le nombre d'éléments n'est pas limité (autrement que par les capacités mémoire de la machine hôte).

L'interface du service est définie par la liste des fonctions de manipulation de listes, leur prototypes et les types associés. Elle est donnée dans le fichier `liste.h` fourni sur Moodle, et ne doit pas être modifiée. Le seul élément à compléter dans ce fichier est la définition du type opaque `liste`, en fonction des choix d'implémentation.

Un programme de test fonctionnel simple est également fourni dans le fichier `testListe.c`. Il exerce les fonctions indispensables du service ; vous devez le compléter pour qu'il les exerce toutes.

1.1.1 Implémentation tableau

Questions préalables :

a/ L'interface permet de créer des listes de capacités différentes, la capacité d'une liste étant spécifiée au moment de sa création. Qu'est-ce que cela implique concernant la nature du tableau que vous allez utiliser pour stocker ses éléments ?

b/ Le programme utilisateur peut demander à créer une liste de capacité infinie. Comment allez-vous gérer ce cas dans l'implémentation tableau ? Remarque : vous pouvez ne pas implémenter ce cas dans un premier temps.

En reprenant le corrigé en C des exercices du cours S1 Algorithmique et structures de données, développez dans le fichier `liste.c` l'implémentation tableau du service. Prenez soin de traiter correctement tous les cas particuliers et les cas d'erreur.

On remarque que certaines des fonctions de l'interface peuvent s'écrire en s'appuyant entièrement sur d'autres fonctions de plus bas niveau de l'interface, sans faire référence à la structure de données sous-jacente : par exemple, `position()` peut s'écrire à l'aide de `taille()` et `lire()` uniquement, sans référencer les champs de la structure `liste`. Cela vous paraît-il un bon choix ? Discutez les avantages et les inconvénients.

Testez votre implémentation en compilant et en exécutant le programme de test fonctionnel fourni.

1.1.2 Implémentation éléments chainés

Remarque préalable : un des intérêts de cette implémentation est de permettre la création de listes de capacité infinie de façon naturelle. Notez cependant que limiter la capacité d'une liste est une fonctionnalité *souhaitée* dans certains contextes – par exemple le schéma de communication producteur / consommateur. En conséquence, même dans l'implémentation éléments chainés, vous devez correctement prendre en compte le paramètre capacité.

En reprenant le corrigé en C des exercices du cours S2 Algorithmique et structures de données, développez maintenant l'implémentation éléments chainés. Utilisez la compilation conditionnelle pour placer cette seconde implémentation dans le fichier `liste.c`, en regard de la première. Cela permet de (i) factoriser les portions de code communes aux deux implémentations, (ii) renforcer la cohérence des parties spécifiques à chaque implémentation, notamment dans les cas particuliers et les cas d'erreur et (iii) faciliter en conséquence la maintenance de votre code.

Testez cette seconde implémentation avec le même programme de test que la première.

1.2 Service de gestion de files et piles

Comme étudié dans le cours Algorithmique et structures de données, on peut construire très simplement une file et une pile à l'aide d'une liste. Pour une file : on ajoute (enfile) à une extrémité de la liste (tête ou queue) et on retire (défile) de l'*autre*. Pour une pile : on ajoute (empile) à une extrémité de la liste, et on retire (dépile) de la *même*.

En respectant les interfaces fournies dans `file.h` et `pile.h`, implémentez un service de gestion de file dans `file.c` et un service de gestion de pile dans `pile.c`.

Sur le modèle de celui fourni pour les files, développez un programme de test fonctionnel pour les files et les piles. Exécutez-le pour vérifier vos implémentations.

On remarque que les fonctions de gestion de files et de piles s'écrivent de façon très simple, en une seule ligne, à partir de leur alter ego de gestion de listes. Développez en conséquence une seconde version de ces services dans laquelle les fonctions sont des macros.

2 Analyse des performances

Nous analysons maintenant les performances des files et des piles que nous avons développées. Chacune des deux structures peut reposer soit sur un tableau, soit sur des éléments chainés, ce qui fait 4 configurations distinctes à analyser.

2.1 Conception d'un benchmark

Pour qu'elle soit pertinente, notre analyse doit se faire selon un scenario d'utilisation réaliste, proche de l'utilisation classique d'une file ou d'une pile dans une application réelle. Ce scenario s'appelle un benchmark.

On peut prendre l'exemple d'une application de manipulation de graphe, et l'algorithme classique de parcours de graphe, qui s'appuie sur une file pour effectuer le parcours d'un graphe en largeur d'abord, ou sur une pile pour effectuer un parcours en profondeur d'abord.

Un autre point à prendre en compte est le temps d'exécution très court des fonctions. Il faut les exécuter un très grand nombre de fois pour s'affranchir des biais statistiques et de la faible résolution de la mesure du temps, et obtenir des mesures pertinentes.

Concevez le benchmark qui vous semble pertinent. Quels sont ses paramètres ?

2.2 Développement du programme de test

Développez un programme de test de performances correspondant au benchmark que vous avez conçu. Les paramètres du benchmark sont passés en arguments de lancement du programme de façon à enchainer une batterie de tests différents sans recompiler le programme.

2.3 Mesures de performances cpu

Une première mesure des performances consiste à mesurer le temps d'exécution du programme de test au moyen de la commande `time`. Le temps `real` dépendant de la charge de la machine au moment de l'exécution, nous nous focalisons sur le temps `user`.

L'utilisation de `gprof` permet d'aller plus loin en comprenant la répartition du temps d'exécution entre les différentes fonctions appelées. Le *flat profile* permet d'identifier les fonctions les plus consommatrices, le *call graph* permet de préciser dans quelles séquences elles ont été appelées.

Mesurez les performances des 4 configurations et interprétez les résultats.

2.4 Recommandations

Pour quel usage les implémentations tableau et éléments chainés sont-elles recommandées ?

Au vu des résultats, voyez-vous des optimisations à apporter au code ?

2.5 Analyse de la consommation mémoire

Nous utilisons valgrind, un logiciel de débogage et de profilage de programmes. Il est livré avec un ensemble d'outils standard, dont massif, qui analyse la consommation mémoire, notamment dans le tas. Le mode opératoire consiste dans un premier temps à exécuter le programme à analyser sous le contrôle de valgrind massif :

```
$ valgrind -tool=massif <programme> <paramètres>
```

et d'afficher ensuite avec la commande `ms_print` le fichier de traces généré par l'exécution :

```
$ ms_print massif.out.<pid>
```

Utilisez valgrind massif pour analyser et comparer la consommation mémoire des deux implémentations tableau et éléments chainés, pour une liste de taille donnée.

Quel est le programme de test existant le plus approprié pour faire cette analyse ?

Quel est le ratio asymptotique de consommation mémoire entre les deux implémentations ?

Comparez les résultats obtenus par valgrind massif avec votre propre évaluation papier de la consommation mémoire. Y a-t-il une différence ?

2.6 Analyse des fuites mémoire

L'analyse des fuites mémoire potentielles d'un programme peut se faire selon deux méthodes :

1/ en instrumentant le code, grâce à l'option de gcc nommée `-fsanitize=address`, puis en exécutant le programme instrumenté.

2/ en exécutant sous le contrôle de valgrind et de l'outil memcheck le programme compilé normalement.

En utilisant ces deux méthodes, analyser les fuites mémoire – ou d'autres erreurs plus sévères – des programmes que vous avez écrits, ainsi que des corrigés fournis sur Moodle. Corrigez ensuite toutes les fuites que vous trouvez.

Dans le « LEAK SUMMARY » fourni par memchek, à quoi correspondent les blocs « indirectly lost » et les « still reachable » ? Développez un programme simple qui génère un exemple de chaque catégorie.