TP Algo2 – session 1 – Tas

23 mai 2025

Objectif d'apprentissage : cette première séance de TP vise à prendre en main des outils qui permettront d'implémenter proprement les algorithmes relatifs aux cours. Ces outils seront utilisés notamment pour garantir la qualité du code et serviront de base à la notation des évaluations prévues. Ce TP illustrera ces outils via l'implémentation des algorithmes liés aux tas.

On insistera sur quatre approches :

— L'analyse statique du code : on s'appuiera sur les avertissements renvoyés par le compilateur avec des options spécifiques. Il ne devra y en avoir aucun.
— L'analyse dynamique du programme : l'outil `valgrind` permettra d'analyser les accès et fuites mémoires. Tous les problèmes relevés devront être traités.
— Une validation par le test : des tests unitaires minimalistes seront à mettre en place et tous devront passer avec succès.
— La clarté du code : l'outil `indent` permettra d'avoir un rendu uniformisé avec les mêmes conventions de codage appliquées à l'ensemble du code produit.

L'archive fournie contient quatre fichiers :

`heap.h` Ce fichier contient les définitions des fonctions à implémenter. Le fichier `.h` sera toujours donné déjà complété dans les prochaines séances et ne doit jamais être modifié.

`heap.c` Les implémentations des algorithmes. Ce fichier sera à compléter.

`heap_tests.c` Les tests unitaires des algorithmes. Une version partielle est donnée. Il sera à compléter, mais sans modifier la fonction `main`.

`Makefile` L'automatisation de la mise en forme du code (avec `indent`), sa compilation (avec `gcc`), puis l'exécution des tests unitaires (avec `valgrind`). Ce fichier ne doit pas être modifié. Il est recommandé de l'utiliser juste avant la soumission (avec `make all`) pour s'assurer que tout est correct.

On va revoir toutes les étapes d'implémentation et de validation par un algorithme permettant de vérifier qu'un tableau représente un tas max valide.

1 Vérification tas max

1.1 Implémentation

On commence par implémenter la fonction `is_max_heap` dans le fichier `heap.c` en respectant la définition de la fonction donnée dans le fichier `heap.h`.

Toute fonction auxiliaire utile serait à déclarer et à implémenter dans le fichier `.c` car il ne faut pas modifier le fichier `.h`.

(Vous remarquerez peut-être que nous utilisons ici un simple tableau accompagné de son indice plutôt qu'une structure qui les regrouperait. Ce serait en effet une meilleure façon de faire et c'est ce qui sera fait par la suite, mais on se focalise sur cette structuration plus simple pour cette première séance.)

1.2 Compilation

La compilation devra se faire avec les options suivantes pour s'appuyer au maximum sur les capacités d'analyse statique. Cela permet de réduire en amont les bugs potentiels.

```
gcc -std=c99 -Wall -Wextra -pedantic -g -O2 heap.c heap_tests.c -o heap
```

Chaque avertissement doit être traité et il ne doit en rester aucun (on peut ignorer les avertissements qui concernent des parties qui seront faites par la suite).

1.3 Validation

On exécutera ensuite les tests unitaires pour tester la validité de ce qui a été codé. Pour ce premier algorithme, les tests sont déjà fournis dans `heap_tests.c`, mais il sera pertinent de compléter ces tests pour les prochains algorithmes.

On utilise une solution minimaliste inspirée de MinUnit qui consiste en deux macros au début du fichier. Chaque test unitaire consiste ensuite en une fonction spécifique qui est renseignée dans un tableau de fonctions juste avant le `main` qui les appelle.

Pour faciliter la mise en place de tests unitaires, il pourra être utile d'implémenter des fonctions auxiliaire dans le fichier `_tests.c`, mais il ne faut pas modifier la fonction `main`.

L'exécution finale doit produire le message (avec x le nombre de tests) :

```
All x tests passed
```

Le processus de validation est donc le suivant. Une fois une fonction implémentée (ou avant, ça marche aussi), il faut créer une fonction de test dans le fichier `_tests.c` et la renseigner dans le tableau `tests_functions`. On pourra ensuite compléter cette fonction avec ce qui irait plus classiquement dans une fonction `main` : manipuler des données et appeler la fonction à tester, afficher des informations avec `printf`, mettre en place des assertions avec `mu_assert` sur le modèle des exemples qui sont fournis, etc. Mais, au lieu de manipuler un `main`, on manipule ici une fonction de test, ce qui rend la vérification plus flexible.

1.4 Accès et fuites mémoires

On utilisera `valgrind` lors de l'exécution des tests unitaires pour vérifier dynamiquement les accès mémoires.

```
valgrind ./heap
```

L'analyse des accès mémoires ne doit rien détecter d'anormal. En, particulier, l'affichage doit contenir les lignes suivantes :

```
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Si l'outil détecte une anomalie, il faut utiliser le même méthode qu'avec les avertissements du compilateur : identifier le premier message d'erreur et la ligne concernée dans le code, interpréter le message d'erreur et déterminer l'erreur dans le code. Une fois le problème traité, il faut relancer l'outil pour vérifier l'impact et recommencer avec le premier message d'erreur.

(Plus d'informations sur l'usage de `valgrind` sont disponibles dans le document "Apprendre le langage C et son bon usage" disponible dans le cours Moodle.)

1.5 Conventions de codage

Enfin, on s'assurera de la clarté du code en garantissant le minimum : l'application d'un style d'indentation uniforme pour l'ensemble du code et pour tous les étudiants. Il suffira d'exécuter la commande suivante sur le fichier source :

```
indent -npsl -nut heap.c
```

2 Construire un tas

Appliquer la même approche pour les autres fonctions définies dans `heap.h`.

Compléter la stratégie de tests unitaires avec des tests basiques, puis, s'il vous reste du temps, l'ensemble des cas pathologiques que vous arrivez à trouver (sur l'exemple des tests fournis pour la première fonction).

Un `Makefile` automatise les opération d'indendation, de compilation et de test :

`make indent` pour indenter le code ;

`make build` pour compiler le code (ou simplement `make`) ;

`make test` pour lancer la stratégie de tests unitaires ;

`make all` pour l'ensemble de ces étapes ;

`make clean` pour nettoyer les fichiers générés.

Vous pouvez échanger votre stratégie de tests unitaires avec une autre personne pour voir si vous détectez un bug dans son implémentation et si la sienne détecte un bug dans la votre.

3 Soumission

Soumettre les fichiers `heap.c` et `heap_tests.c` **tels quels** (fichiers sources non compressés, non archivés, non renommés) sur le dépôt de devoir Moodle associé pour une évaluation automatique du travail réalisé (échéance fixée au dimanche soir de la semaine en cours).

Une note sera générée en prenant en compte :

— l'absence d'avertissements à la compilation avec `gcc` ;
— l'absence de problèmes détectés par `valgrind` ;
— le succès de la stratégie de tests unitaires enseignantes avec `MinUnit` ;
— le respect des conventions de codage avec `indent`.

La note se base essentiellement sur la proportion de tests enseignants qui réussissent (sans accès mémoire illégal) et est ensuite pondérée en fonction des problèmes d'indentation, des avertissements du compilateur et des fuites mémoires. La stratégie de tests unitaires des étudiants n'est pas directement évaluée mais une bonne stratégie de tests permet d'identifier des bugs et d'augmenter le nombre de tests qui réussit. (Il est demandé de la soumettre pour une évaluation éventuelle, mais peu probable cette année.)

La note ne comptera pas dans l'évaluation du module et cette étape est facultative, mais indicative du respect des consignes donc fortement recommandé car cela permet de se préparer aux soumissions qui compteront dans la note finale (mini-projet, projet-tournoi et épreuve de TP). Comme cette note ne comptera pas, il n'y aura pas de détection de plagiat.

Pour rappel, seuls les fichiers `heap.c` et `heap_tests.c` (sauf la fonction `main`) doivent être modifiés.