CRO TD5

Arbres binaires

Outils de débogage : `gdb, ddd, nemiver`

1 séance sur machine, 17 juillet 2024

en cas de confinement : tout le monde peut faire les exercices sur les arbres binaires. En revanche, les exercices utilisants `gdb`, `ddd` *ou* `valgrind` *dépendent de votre installation, sur les distribution linux, pas de problèmes mais pour les autres je ne sais pas.*
Si vous arrivez à faire au moins la section 2 (utilisation de `gdb` *en ligne de commande), en connexion* `ssh` *sur les machines du départements c'est bien.*
Sinon, gardez vous un TODO pour la fin du confinement : `gdb` *est un outil important et à connaître. Si vous avez terminé et que vous êtes coinçé pour* `gdb`*, vous pouvez continuer avec les question falcultatives en fin de TD.*

1 Arbres binaires en C

Les arbres binaires que vous avez vu dans le cours ALG, sont implémentés en C par une combinaison de pointeur et de structure, comme le sont les listes. On commencera avec la définition de type suivante pour un noeud d'un arbre binaire d'entier (un arbre étant simplement un pointeur vers un noeud). Ici encore l'absence de fils est encodée par la valeur 0 pour le pointeur (on utilise plutôt la macro NULL, définie dans `<stdlib.h>`).

```
struct model_noeud
{
  int val;
  struct model_noeud *filsGauche ;
  struct model_noeud *filsDroit ;
} ;

typedef struct model_noeud NOEUD;

typedef NOEUD *ARBRE;
```

QUESTION 1 ▶ Créez les fichiers `arbre_type.h`, `arbre.c`, et `arbre.h` ainsi que le fichier `Makefile` permettant de compiler le fichier `arbre.c` (même si il est vide initialement). A ce stade, le fichier `Makefile` ne sert qu'à générer le fichier `arbre.o`

QUESTION 2 ▶ Définir la fonction `creerArbre`, donc le prototype est :
`ARBRE creerArbre(int val, ARBRE filsGauche, ARBRE filsDroit)`
Créer un fichier `main.c` qui appelle cette fonction pour créer un arbre avec (au moins) trois noeuds. Attention, vous allez avoir besoin d'utiliser la fonction `malloc` et donc d'inclure les headers de la librairie standard standard : `<stdlib.h>`.

QUESTION 3 ▶ Définir la fonction *récursive* `afficheArbre` qui utilise, par exemple, un parcours infixe pour afficher tous les noeuds de l'arbre. Tester cette fonction sur votre arbre.

QUESTION 4 ▶ Définir la fonction `hauteurArbre` qui calcule la hauteur d'un arbre. Tester cette fonction sur un arbre ayant des branches de longueurs différentes.

2 Outil d'aide pour les fuites mémoire : Valgrind

`valgrind` est un outil de vérification de code à l'exécution. Le code est exécuté dans une machine virtuelle qui vérifie les appels, adresses mémoires, etc. `valgrind` est essentiellement utilisé pour découvrir tous les problèmes mémoires : "fuites mémoire" (memory leak : mémoire allouée mais

non libérée) et accès en dehors des bornes des tableaux. Les fuites mémoires ont peu d'importance pour les programmes qui terminent car l'espace alloué est libéré par le système d'exploitation, mais lorsque le programme est utilisé comme librairie dynamique, les fuites répétées peuvent rapidement saturer la mémoire.
`valgrind` est extrêmement simple d'utilisation, si votre programme se lance avec la commande :
`monProg arg1 arg2`
il suffit d'exécuter :
`valgrind [option] monProg arg1 arg2`
ou `[option]` indique ce que l'on désire mesurer avec valgrind. L'option par défaut (pas d'option) exécute `memcheck` qui vérifie l'essentiel des problèmes mémoire comme l'accès en dehors des bornes des tableaux, l'utilisation de mémoire après libération ou de mémoire non initialisée et les fuites mémoire. Le rapport donnée par `valgrind` n'est pas très convivial.

QUESTION 5 ▶ Téléchargez le le programme `valgrindEx.c` sur Moodle et visualisez le, repérez les erreurs de la fonction `f`.

— Compiler ce fichier pour valgrind :
```
gcc valgrindEx.c -o valgrindEx
```
— Exécutez ce programme : `>./valgrindEx`
— Exécutez ce programme avec `valgrind` : `>valgrind valgrindEx`

Repérez dans le rapport de `valgrind` les erreurs que vous avez notés. Exécutez ce programme avec `valgrind` en mettant l'option permettant de visualiser ou la fuite mémoire a lieu.

3 Outils de débugage : GDB

Le débogage de programme C est un art en soi. Le travail le plus important se fait lors de la conception par la structuration en modules et fonctions ainsi que lors du choix de la structure de donnée. La programmation et le test de chaque module indépendamment permet aussi de gagner beaucoup de temps sur l'ensemble du projet. Malgré cela, il reste souvent des bugs difficiles à trouver rapidement. Nous allons étudier un outil fondamental du développement sous unix : `gdb` (et une interface associée : `nemiver` ou `ddd`), debuggeur indispensable pour comprendre certaine erreurs.

QUESTION 6 ▶ Rajouter l'option `-g` à vos règles de Makefile qui produisent les fichiers `.o`. par exemple :

```
arbre.o: arbre.c arbre.h arbre_type.h
gcc -c -g arbre.c -o arbre.o
```

Recompilez votre programme et lancez la commande (en supposant que votre binaire exécutable s'appelle `main`) :
`gdb main`
Vous devez avoir une invite `gdb`.

— Tapez la commande `break main`, puis `run`.
— Tapez alors la commande `layout src` qui vous permet de voir le code source en même temps.
— La commande `backtrace` permet à tout moment de connaître l'enchaînement d'appels de fonctions successifs qui ont amené à cet instruction.
— La commande `step` avance d'une instruction C, c'est à dire qu'à l'appel d'une fonction, elle entre dans le corps du code de la fonction.
— La commande `next` a le même comportement mais elle n'entre pas dans les fonctions appelées, elle les exécute d'un seul coup.

GDB

`gdb` est un *debugger symbolique*, c'est-à-dire un utilitaire Unix permettant de contrôler le déroulement de programmes C. Il permet (entre autres) de mettre des points d'arrêt dans un programme, de visualiser l'état de ses variables, de calculer des expressions, d'appeler interactivement des fonctions, etc.
`nemiver` ou `ddd` sont des interfaces graphiques qui facilitent l'utilisation de `gdb` sous X-Window, mais `gdb` ne nécessite aucun système de fenêtrage, il peut s'exécuter sur un simple terminal shell (mode *console*). Il est indispensable de comprendre le fonctionnement en mode `gdb` pour pouvoir utiliser les `ddd`.
Lorsque l'on lance `gdb` avec la commande `gdb unexecutable.exe`, `gdb` a chargé l'exécutable, il attend alors une commande `gdb`, comme par exemple `run` (pour exécuter le programme), `break` (pour mettre un point d'arrêt dans le programme), `step` (pour avancer d'une instruction dans le programme), etc. Il y a donc une *invite* `gdb`.
Les points d'arrêt peuvent se positionner au début des fonctions (`break main`, par exemple), ou à une certaine ligne (`break 6`, par exemple), ou lorsqu'une variable change de valeur (`watch i`, par exemple).
Les commandes importantes (et leur raccourcit) à connaître pour utiliser `gdb` sont les suivantes :

- `r` est équivalent à `run`
- `b main` est équivalent à `break main`
- `p var` est équivalent à `print var`
- `d var` est équivalent à `display var`
- `s` est équivalent à `step`
- `n` est équivalent à `next`
- `c` est équivalent à `continue`
- La commande `run` peut prendre des arguments, le programme est alors exécuté avec les arguments donnés
- la commande `info` permet d'afficher des informations sur l'état du programme dans le débugger. Par exemple `info b` liste les points d'arrêt.
- La commande `help` est l'aide en ligne de `gdb`

QUESTION 7 ▶ Mettez un point d'arrêt à la fonction `main`, entrez la commande `run`. Lorsque `gdb` vous rend la main, avancez pas à pas en tapant la commande `step` (raccourcit `s`). Assez rapidement vous devez avoir un message d'erreur comme quoi `gdb` ne trouve pas le code source, pourquoi ?

QUESTION 8 ▶ relancer le débuggage (commande `run`), ajouter un point d'arrêt à la fonction `afficher_arbre`, continuer l'exécution avec la commande `continue` (raccourcit : `c`). Afficher la valeur de l'arbre, par exemple si la variable arbre s'appelle `a` :

```
print a
```

A quoi correspond cette valeur ?
Affichez maintenant *l'objet pointé par* `a`, c'est à dire le noeud racine de votre arbre :

```
print *a
```

Affichez le fils gauche de la racine :

```
print *(a->filsGauche)
```

Expérimentez l'option `/x` de `print` :

```
print/x *(a->filsGauche)
```

Qu'est ce qui change ?

QUESTION 9 ▶ À quelle adresse est stockée votre arbre de référence ?

QUESTION 10 ▶ Affichez les 10 valeurs consécutives stockées en mémoire à partir de l'adresse de votre arbre en utilisant l'opérateur `@` de `gdb` (voir `help print` dans `gdb`). Par exemple, si l'adresse à laquelle est stockée votre arbre est `0x602050`, tapez :
`print/x *( 0x602050)@10`

Est ce que vous retrouvez vos éléments ?

4 Pour aller plus loin (facultatif) : Interface graphique du débugger

Note : Depuis quelques années, l'arrivée du codage UTF-8 a rendu obsolète un certain nombre d'interface graphique de `gdb`, nous essayons cette année sans garantie, en cas d'incohérence de comportement, toujours préférer l'interface textuelle de `gdb`.
L'interface graphique du débugger permet de faire les mêmes manipulations, mais dans un environnement plus convivial. Il existe deux interfaces graphiques installées sur les machines du département : `ddd` et `nemiver`. `ddd` n'est plus maintenu, mais `nemiver` est moins intuitif, on commencera par utiliser `ddd`.
Lancer `ddd`, par exemple, si votre exécutable s'appelle `main`, lancez :
`ddd main`

L'interface graphique est notamment utilisée pour se balader dans facilement dans les structures.

- Apprenez à vous balader dans les sources, chargez `arbre.c`, mettez un point d'arrêt sur la fonction `afficher_arbre`.
- Après le premier arrêt, exécutez le programme pas à pas, lorsque vous êtes dans la fonction `afficher_arbre`, afficher votre arbre de manière permanente, par exemple, si votre variable locale à la fonction `afficher_arbre` s'appelle `a`, tapez la commande :
  `graph display a`
  Ou utilisez le menu de `ddd` pour cela.
- Déroulez la structure en cliquant sur les champs filsGauche, filsDroit

5 Pour aller plus loin (facultatif) : Arbres Binaires de recherche

Dans cette exercice facultatif on utilisera toujours le même type `ARBRE` donné en page **??**, mais pour stocker un arbre binaire de recherche (ABR). On rappelle que ce sont des arbres binaires de valeurs avec les propriétés :

- Tous les nœuds du sous-arbre gauche d'un nœud ont une valeur inférieure (ou égale) à la sienne
- Tous les nœuds du sous-arbre droit d'un nœud ont une valeur supérieure (ou égale) à la sienne

Pour chacune des fonctions suivantes, testez-les avec la fonctions `afficherArbre` programmée précédemment.

QUESTION 11 ▶ Écrire une fonction :
`ARBRE ajouterABR(ARBRE a, int val)`
qui prend en argument une valeur entière ainsi qu'un ABR et renvoie l'ABR `a` auquel on aura ajouté un noeud avec la valeur `val` (en respectant les règles `ABR` bien sûr).

QUESTION 12 ▶ Écrire une fonction :
`ARBRE suprimerABR(ARBRE a, int val)`
qui prend en argument une valeur entière ainsi qu'un ABR et renvoie l'ABR `a` auquel on aura *supprimé* la valeur val, l'arbre en résultant doit toujours respectant les règles `ABR` bien sûr.

QUESTION 13 ▶ La transformation de *rotationDroite* est illustrée sur la figure **??**, elle conserve le caractère ABR d'un arbre. Écrire une fonction
`ARBRE rotationDroite(ARBRE a)`
qui applique une rotation droite sur la racine de l'ABR `a` (si elle est possible). Ajoutez dans le `main` une utilisation de cette fonction.

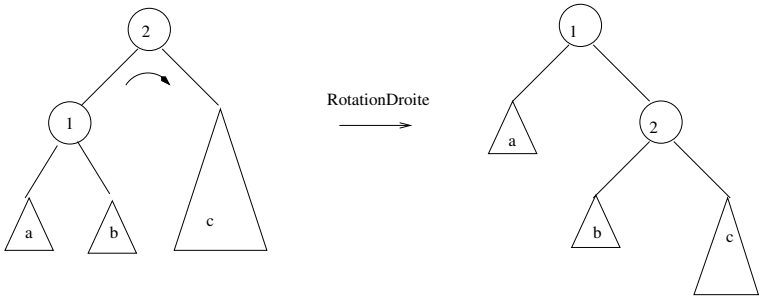

FIGURE 1 – Illustration de la transformation “rotation droite” sur un arbre