TD-TP 4 : traduction sur exemples complexes

Le but de cette séance est de consolider les acquis des séquences précédentes.

Commencez par récupérer les sources pour cette séance sur l'Ensiwiki.

Les exercices sont organisés comme dans les TP précédents : un fichier `exo.c` qui contient le programme principal, un fichier `fct_exo.s` à remplir, et une règle de génération dans le Makefile (`make exo`) pour générer l'exécutable. Dans tous les exercices, il convient de vérifier l'exécution pas à pas du programme avec GDB et le simulateur.

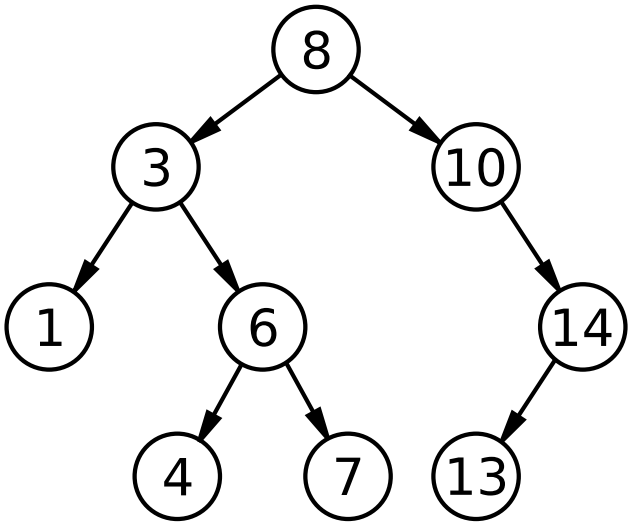
Ex. 1 : Parcours d'arbres binaires de recherche (annale juin 2013)

On va travailler dans cet exercice sur des arbres binaires de recherche (ABR) dont les nœuds contiennent une simple valeur entière naturelle. On rappelle les principales propriétés de cette structure de données :

- chaque nœud de l'arbre a au plus 2 fils ;
- le sous-arbre gauche d'un nœud N donné ne contient que des nœuds dont la valeur est strictement inférieure à la valeur de N ;
- le sous-arbre droit d'un nœud N donné ne contient que des nœuds dont la valeur est strictement supérieure à la valeur de N ;
- les sous-arbres gauche et droit d'un nœud donné sont aussi des arbres binaires de recherche.

Il vient naturellement de ses propriétés que chaque valeur est unique (*i.e.* il n'existe pas plusieurs nœuds de même valeur).

Le schéma ci-dessous est un exemple d'ABR qu'on va utiliser dans cet exercice :



On représente le type des nœuds d'un ABR par la structure suivante :

```
struct noeud_t {
    uint32_t val;       // valeur d'un noeud
    struct noeud_t *fg; // fils gauche du noeud
    struct noeud_t *fd; // fils droit du noeud
};
```

Un ABR est représenté simplement par un pointeur vers un nœud : `struct noeud_t *abr`. Si `abr == NULL` alors l'ABR est vide.

Le fichier `abr.c` contient le programme principal et quelques fonctions servant à tester le code écrit en assembleur.

Le fichier `fct_abr.s` contient le squelette des fonctions à écrire. Le code C à traduire est donné en commentaires avant chaque fonction.

Question 1 Compléter la fonction `est_present` : cette fonction prend en paramètre une valeur entière naturelle sur 32 bits et un ABR. Elle renvoie vrai *ssi* la valeur est présente dans l'ABR.
Le principe de la fonction à écrire est simple :

- si l'arbre est vide, alors la valeur n'est sûrement pas présente, on retourne *faux* :
- sinon, si le nœud courant contient la valeur recherchée, on renvoie *vrai* :
- sinon, si la valeur recherchée est plus petite que la valeur du nœud courant, on poursuit la recherche dans le fils gauche :
- sinon (c'est à dire si la valeur recherchée est plus grande que la valeur du nœud courant), on poursuit la recherche dans le fils droit.

Question 2 Compléter la fonction `abr_vers_tab` : cette fonction prend en paramètre un ABR (c'est à dire un pointeur vers la racine de l'arbre).
La fonction parcourt l'ABR et copie les valeurs des nœuds de l'arbre dans un tableau. Comme le parcours se fait en profondeur d'abord dans le sous-arbre gauche, puis dans celui de droite, le tableau final sera donc trié par ordre strictement croissant. Au passage, la fonction détruit les nœuds pour récupérer l'espace mémoire.

Le tableau est alloué dans le programme principal (`uint32_t tab[NBR_ELEM];`). On recopie ensuite son adresse dans une variable globale `uint32_t *ptr` de type « pointeur vers un entier ». Cette variable est physiquement localisée dans la zone `.data` du fichier `fct_abr.s`, et elle est rendue visible dans le fichier `abr.c` grâce au mot clé `extern`.

Le principe de la fonction de copie est simple, pour un arbre non-vide :

- on copie récursivement tous les éléments du fils gauche du nœud courant dans le tableau :
- on copie la valeur du nœud courant dans le tableau :
- on incrémente la variable `ptr` de façon à ce qu'elle pointe sur la case suivante du tableau :
- on détruit le nœud courant (avec la fonction `free`) : pour pouvoir continuer à parcourir le reste de l'arbre, on doit donc d'abord sauvegarder un pointeur vers le fils droit du nœud :
- on copie récursivement tous les éléments du fils droit dans le tableau.

La variable `ptr` sert donc à désigner la prochaine case libre dans le tableau, et on l'avance d'une case à chaque fois qu'on copie une valeur dans le tableau.

Pour aller plus loin... **Ex. 2 : Passage de tableaux en paramètre**

Dans cet exercice, on cherche à implanter l'algorithme du tri du nain de jardin, dont voici le principe.

Un nain de jardin souhaite trier des pots de fleurs par taille croissante en appliquant la stratégie suivante. Il regarde le pot devant lui :

— s'il est plus petit que le pot à sa droite, le nain avance d'un pas vers la droite (s'il n'est pas arrivé à la fin de la file de pots) ;

— si le pot devant lui est plus grand que le pot à sa droite, le nain échange les deux pots, et recule d'un pas vers la gauche (s'il n'est pas revenu au début de la file de pots).

Le fichier `fct_tri_nain.s` contient en commentaires une implantation en C de cet algorithme. On note que la fonction `tri_nain` prend en paramètre le tableau d'entiers à trier. On rappelle qu'en C, lorsqu'on passe un tableau en paramètre d'une fonction, c'est l'adresse du tableau qui est passée (le premier paramètre d'une fonction est stockée dans $a0 et on ne peut évidemment par recopier l'ensemble des éléments du tableau dans un seul registre !).

Pour tester votre programme, le fichier `tri_nain.c` fourni calcule les temps d'exécution de votre tri et d'un tri de référence, et vérifie que votre tri est correct (un message d'erreur sera affiché si le tableau résultat est faux).

Question 1 Traduisez en langage d'assemblage la fonction `tri_nain`.

On remarque qu'on fait beaucoup d'accès mémoire redondants dans cette traduction systématique, et qu'il semble facile d'optimiser le tri en utilisant intelligemment les registres pour stocker des valeurs.

Question 2 Implantez dans le fichier `tri_nain.s` une fonction `tri_nain_opt`, qui évite autant que possible les accès mémoires.

Notez bien que le tri de référence est un "quicksort", c'est à dire un tri en $O(n \times log(n))$ alors que le tri du nain est un tri en $O(n^2)$. Même en optimisant le code produit, cela ne compensera pas le fait que l'algorithme du nain est intrinsèquement peu efficace.