Licence Informatique **J1IN601 – Algorithmique 3**
Université Bordeaux 1

TP noté avril 2013

Modalités du TP

L'archive fournie contient l'ensemble des fichiers nécessaires pour le TP. Il est interdit d'ajouter un fichier et les seuls fichiers que vous pouvez modifier sont les fichiers `heap1.c`, `heap2_recursif.c` et `heap2_iteratif.c`. Il est notamment interdit de modifier les fichiers `.h`.

Chaque étudiant doit faire le TP seul. Vous pouvez vous inspirer des fichiers produits lors des différents TP ainsi que des corrections fournies. Toute autre source d'« inspiration » (voisins, internet...) et strictement interdite.

À la fin de l'épreuve vous enverrez un email à votre chargé de TP usuel, dont le sujet sera « TP noté d'algorithmique 3 » et dont le corps contiendra vos nom et prénom ainsi que les fichiers `heap1.c`, `heap2_recursif.c` et `heap2_iteratif.c` que vous aurez produit. **Tout écart dans le format de l'email et tout email reçu après l'heure de fin de l'épreuve seront sanctionnés.**

Pour avoir tous les points, il ne faut pas que votre implémentation génère des fuites de mémoire.

Présentation du sujet

Le but de ce TP est d'implémenter des variantes de deux `tas min` vus en cours. Outre les fonctions classiques `heap_insert`, `heap_get_min` et `heap_remove_min`, ces tas possèdent une fonction `heap_merge`. Les fonctions `heap_insert` et `heap_remove_min` doivent être implémentées en utilisant la fonction `heap_merge`.

```
/* create an empty heap tree */
heap heap_create(keyfunc f);

/* destroy the heap */
void heap_destroy(heap h);

/* insert an element in the heap or NULL if object is NULL */
heap heap_insert(heap h, void* object);

/* get the element of minimum weight of NULL in case of empty heap */
void* heap_get_min(heap h);

/* removes the element of minimum weight or does nothing in case
   of empty heap */
heap heap_remove_min(heap h);

/* inserts all the elements of B into A, empties B and returns A.  */
heap heap_merge(heap A, heap B);
```

Sujet

Dans ce sujet, on demande d'implémenter plusieurs fonctions sur ces tas. Pour chaque fonction `riri` à implémenter, les fichiers `prof_*.o` contiennent une fonction `prof_riri` équivalente. Si besoin, vous pourrez utiliser ces fonctions.

Pour les premières questions, on travaille dans le fichier `heap1.c`.

1. Implémenter `heap_create`.
2. Implémenter `heap_destroy`.
3. Implémenter `heap_get_min`.
4. Implémenter `heap_insert`.
   Indication : On peut commencer par créer un tas avec un nœud.
5. Implémenter `heap_remove_min`.
   Indication : Si on supprime la racine d'un tas, on obtient deux tas.
6. Implémenter `heap_merge`.
   Pour cela, à chaque nœud x d'un tas on associe une *S-valeur* qui est la longueur du plus long chemin partant de x et qui n'emprunte que des fils droits.
   Soient x_A et x_B les éléments minimaux respectifs de A et B. Supposons que $x_A \leq x_B$. On fusionne récursivement B et le sous-arbre droit de A. Une fois la fusion effectuée, pour équilibrer le tas et obtenir une bonne complexité, si la S-valeur de $A \to \text{left}$ est inférieure à celle de $A \to \text{right}$, alors on permute les fils gauches et droits de A. Bien évidemment, si $x_A > x_B$, on fusion récursivement A et le fils droit de B.
7. On implémente maintenant une stratégie différente pour fusionner deux tas. Au lieux de permuter les fils de A seulement quand la S-valeur de $A \to \text{left}$ est supérieure à celle de $A \to \text{right}$, on permute toujours les deux fils de A.
   Implémenter une version récursive de cette stratégie dans le fichier `heap2_recursif.c`.

   Notez que si on augmente trop la valeur de `M` dans le fichier `test_heap2_recursif.c`, on obtient un `segmentation fault` et `valgrind` précise que cette erreur est due à un débordement de pile.
8. Pour régler ce problème, implémenter une version itérative de `heap_merge` et de `heap_destroy` dans le fichier `heap2_iteratif.c`. Pour cela, vous pouvez utiliser une structure de pile (`stack.h` et `stack.c`).