TRAVAUX PRATIQUES XXIII

Tri par tas - sujet classique

L'objectif de cette section est de trier un tableau en C par la méthode du tri par tas.

A Représentation d'un tas

1. Rappeler comment représenter un arbre binaire complet à l'aide d'un tableau. Écrire l'arbre correspondant au tableau :

2	13	14	7	1	8

2. Est-ce un tas ? Proposer un tas-min contenant les mêmes éléments et l'écrire sous forme de tableau.

Dans ce TP, on suppose que nos tas ont une taille bornée par $N = 10000$ et on utilise un tableau statique de taille 10000 pour représenter un tas, en retenant sa taille "réelle" dans un entier. On représente nos tas en langage C en utilisant le type suivant :

C
```c
struct tas_s {
    int T[10000]; //tableau statique représentant le tas (pas besoin de l'allouer !)
    int n; //nombre de noeuds du tas (taille réelle du tableau)
};
typedef struct tas_s tas;
```

Dans l'idéal, on voudrait implémenter une file de priorité à l'aide d'un tas. Dans ce sujet, pour simplifier, on ne s'intéressera qu'à un tas d'entier et non à une file de priorité plus générale. Pour une version qui manipule des couples (valeur, priorité), se référer au sujet étoilé.

J'imagine que vous savez ce que je vais vous demander de faire en premier...

3. Écrire une fonction C `affiche_tas` qui prend en argument un tas et l'affiche. On ne demande pas d'effort particulier sur la mise en forme. Par exemple, le tout premier arbre donné dans ce TP pourrait être affiché en terminal de la manière suivante :

Terminal
```
2 13 14 7 1 8
```

Voilà, vous avez de quoi tester vos fonctions !

B Opérations sur les tas

Passons aux choses sérieuses.
On donne en C la signature impérative de tas suivante (extrait du fichier header `tas-classique.h`) :

C : tas-classique.h
```c
//Crée un tas vide.
tas tas_vide(void);

//Insère un élément entier prio dans le tas t.
void insere(tas* t, int prio);

//Extrait le minimum du tas t. MODIFIE le tas.
int extrait_min(tas* t);

//Renvoie le minimum du tas t, SANS modifier le tas.
int lit_min(tas t);

//Teste si le tas est vide.
bool est_vide(tas t);
```

Cette question vous laisse en autonomie sur le TP :

4. Implémenter en C chacune des fonctions de la signature (pour un tas contenant uniquement les priorités).

Pour les tests, vous pouvez partir du tas suivant (à écrire dans votre main) :

```c
tas t = {.T = {1, 8, 7, 14, 13, 20}, .n = 6};
```

Maintenant, trions !

On souhaite désormais trier un tableau d'entiers en C par la méthode du tri par tas.

5. Écrire une fonction `tas entasser(int* tab, int n_tab)` qui renvoie un tas contenant les éléments du tableau `tab`.
6. Écrire une fonction `void tri_par_tas(int* tab, int n_tab)` qui trie le tableau `tab` donné en argument en utilisant la méthode du tri par tas.

   Aide : on rappelle que les deux étapes du tri sont les suivantes :

   1) On crée un tas à partir des éléments du tableau en les insérant un par un dans un tas initialement vide (c'est votre fonction précédente).

   2) On extrait successivement le minimum du tas et on le place au bon endroit dans le tableau à trier, jusqu'à ce qu'il ne reste plus d'éléments dans le tas.

Pour occuper les plus rapides

Pour l'instant, pour implémenter notre tri par tas, nous avons créé à côté du tableau à trier un tas que nous avons rempli avec les éléments du tableau, puis nous avons reversé ce tas dans notre tableau. On peut faire mieux. On peut trier le tableau *en place* par la méthode du tri par tas.

7. Quelle est la complexité en espace obtenue par le tri par tas précédent ?
8. Proposer une modification de l'algorithme pour qu'il effectue le tri *en place*, en complexité spatiale $O(1)$.

   Aide : le tableau et le tas doivent être la même chose, et on doit travailler dessus petit à petit...
9. Écrire en C un tri par tas en place, en utilisant la méthode proposée précédemment. *N'hésitez pas à me faire vérifier votre algorithme avant de vous lancer dans le code ! Ce sera sans doute plus simple de ré-écrire vos fonctions d'insertion et d'extraction pour les adapter à cette situation que de généraliser.*