Travaux Pratiques n°5

Cours d'Informatique de Deuxième Année

—Licence MI L2.2—

Structure de tas

On se propose d'implanter une structure de tas (également appelée *file de priorité*) afin de pouvoir réaliser le tri d'éléments en temps $O(n \log n)$.

Définition du tas Soit un ordre total $\leq$ sur un ensemble E. Un arbre est un tas d'éléments de E si et seulement si :

- L'arbre est binaire complet (pour un arbre de hauteur n, toutes les feuilles sont à une profondeur n ou $n-1$).
- Pour tout nœud (interne ou feuille) a de l'arbre, nous avons $p(a) \leq a$ (où $p(a)$ est le nœud père de a).

Tri par tas De cette définition découle la propriété suivante : pour tout nøeud a du tas, nous avons $r \leq a$ où r est la racine du tas. Cette propriété peut être exploitée afin de réaliser un algorithme de tri :

1. Le tas est initialisé.
2. Les n éléments sont insérés un à un dans le tas.
3. On supprime un à un les éléments du tas en enlevant à chaque itération i la racine : nous avons $r_1 \leq r_2 \leq \cdots \leq r_n$.

Réprésentation du tas Un tas étant un arbre binaire complet, il est possible d'utiliser un tableau pour le représenter :

- La cellule 0 est la racine du tas.
- Si le nœud a est représenté par la cellule d'indice i, alors :
  - son père $p(a)$ ($p(a)$ n'est pas défini pour la racine) est représenté par la cellule d'indice $\lfloor \frac{i-1}{2} \rfloor$.
  - son fils gauche $f_g(a)$ (s'il existe) est représenté par la cellule d'indice $\lfloor 2i+1 \rfloor$.
  - son fils droit $f_d(a)$ (s'il existe) est représenté par la cellule d'indice $\lfloor 2i+2 \rfloor$.

Structure C proposée On propose la structure C générique suivante afin de représenter un tas :

```
/* Type pointeur de fonction comparatrice d'éléments retournant un entier :
= 0 si les deux éléments sont égaux
< 0 si le premier élément est strictement inférieur au deuxième
> 0 si le premier élément est strictement supérieur au deuxième
*/
typedef int (*comparateur)(const void *, const void *);
```

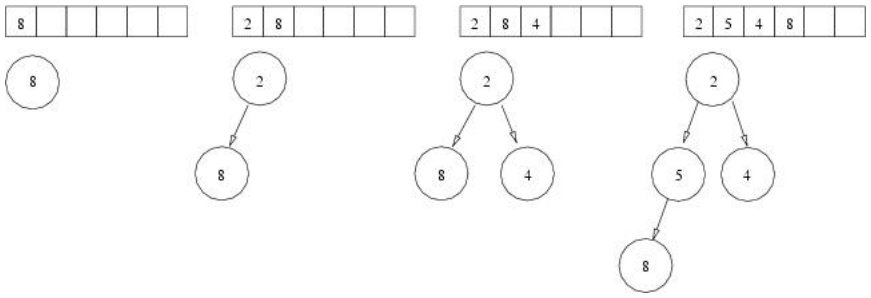

FIG. 1 – Exemple de tas avec insertion des éléments 8, 2, 4 et 5 en utilisant l'ordre naturel des entiers.

```
typedef struct _tas
{
    void * elements; /* Tableau d'éléments dans le tas */
    unsigned int quantum; /* Taille d'un élément en octets */
    unsigned int taille; /* Nombre d'éléments dans le tas */
    unsigned int capacite; /* Capacité du tas : taille <= capacite */
    comparateur cmp; /* Fonction comparatrice d'éléments */
} Tas;
```

▸ **Exercice 1.** ***Initialisation de tas*** *Écrire une fonction d'initialisation d'un tas vide : on spécifie en paramètres la taille de chaque élément (*quantum*), la capacité initiale du tas ainsi que la fonction de comparaison induisant l'ordre entre les éléments. On pourra considérer que la taille d'un élément (*quantum*) est bornée par une constante* MAX_QUANTUM.

▸ **Exercice 2.** ***Vérification de tas*** *Écrire une fonction prenant en argument un pointeur vers une structure de tas vérifiant si la propriété d'ordre est respectée. Cette fonction pourra être réutilisée par la suite comme assertion (macro* assert*) terminant les fonctions d'insertion et de suppression d'éléments.*

▸ **Exercice 3.** ***Insertion d'élément*** *Écrire une fonction permettant d'insérer un élément dans le tas. L'insertion peut être réalisée en deux temps :*

1. *On cherche à conserver la structure d'arbre binaire complet. À cet effet, on ajoute l'élément dans la première cellule inoccupée du tableau et on incrémente la taille du tas.*
2. *On remonte ensuite l'élément dans l'arbre jusqu'à ce que son père lui soit inférieur ou égal afin de conserver la propriété d'ordre.*

Quelle est la complexité temporelle de l'insertion d'un élément ?
On vérifiera avant toute insertion que la capacité du tas la permet : si ce n'est pas le cas, deux approches peuvent être envisagées :

- *Le retour d'un code d'erreur (*0 *si l'insertion échoue,* 1 *si elle est réalisée).*
- *Le remplacement du tableau actuel par un tableau de capacité plus importante (on pourra, par exemple, multiplier la capacité par un facteur* 2*) : il est alors nécessaire de recopier*

les cellules de l'ancien tableau vers le nouveau. Si l'allocation du nouveau tableau est impossible, on retourne une erreur.

▸ **Exercice 4.** ***Suppression de la racine*** *Écrire une fonction pour supprimer l'élément racine du tas (le plus petit élément présent dans le tas) et retournant cet élément. Là encore, cette opération peut être réalisée en deux temps :*

1. *La suppression de la racine et son remplacement par le dernier élément du tas (dernier élément du tableau) : on conserve la structure d'arbre binaire complet.*
2. *La descente du nouveau élément racine afin que ses enfants lui soient supérieurs.*

Quelle est la complexité de la suppression de la racine ?

▸ **Exercice 5.** ***Algorithme de tri*** *Rédiger la fonction de tri par tas de signature* `int Tas_tri(void * elements, unsigned int taille, unsigned int quantum, comparateur cmp)`. *Voici quelques explications sur les paramètres de cette fonction :*

- `void * elements` *: il s'agit du tableau d'éléments à trier : le résultat du tri est indiqué dans le même tableau.*
- `unsigned int quantum` *: ce paramètre désigne la taille en octets d'un élément du tableau.*
- `unsigned int taille` *: il s'agit de la taille en nombre d'éléments du tableau.*
- `comparateur cmp` *: ce paramètre est un pointeur vers une fonction comparant deux éléments.*

La fonction doit retourner 1 *si le tri a été réalisé correctement,* 0 *sinon (par exemple, si des opérations d'allocation de mémoire ont échoué).*
Écrire une fonction vérifiant si un tableau est trié : rajouter une assertion en fin de la fonction de tri afin de déterminer si le tri est correctement réalisé.
Quelle est la complexité temporelle globale, dans le pire des cas, de cet algorithme de tri ? Quelle est sa complexité en espace ? Discuter de ses avantages et inconvénients par rapport à d'autres algorithmes classiques de tri (tri rapide, tri par fusion, ...). On pourra (pour les courageux) comparer les performances pratiques de cet algorithme par rapport à la fonction de tri qsort *de la bibliothèque standard C sur un jeu de données aléatoire.*