TP d'Algo n° 3

EPITA ING1 2013; E. CARLINET

Objectif : L'objectif de ce TP est d'étudier la structure d'un tas et ses algorithmes. Cette structure de donnée est largement utilisée, elle permet notamment d'implémenter des files de priorités, la recherche du plus court chemin en théorie des graphs, et est également au centre des algorithmes de tris *heapsort* (codé aujourd'hui) et *introsort* (codé vendredi).

1 Se former sur le tas

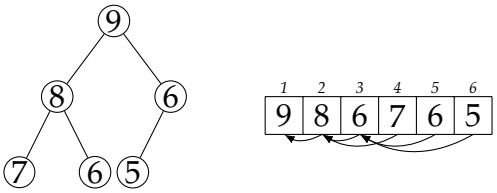

FIGURE 1 – Encodage d'un max-heap avec un vecteur.

encodage Pour implémenter un tas, nous utiliserons l'encodage des arbres parfaits sous forme de tableau. Si les indices commencent à 1 (ce n'est pas votre cas !) un nœud à la position i a sont fils gauche en position $2i$ et son fils droit en position $2i + 1$. Bien sûr les fils n'existent que si leur indices sont inférieurs à la taille du tableau.

structure La propriété de tas doit être assurée en tout nœud : la valeur d'un nœud d'un *max-heap* (resp. *min-heap*) ne peut être inférieure (resp. supérieure) à celles de ses fils.

On ne considère initialement que des *max-heap* dont les nœuds sont entiers.

Question 1 : Écrivez la fonction `void make_heap(int* vector, size_t n)` qui prend un vecteur d'entiers de taille n et le transforme pour qu'il vérifie la propriété de tas. La construction du tas se fera en place selon le principe vu en cours : une approche bottom-up où pour chaque nœud, le sous arbre formé vérifie la propriété de tas. Dans le cas contraire, le nœud courrant est permuté vers le bas jusqu'à atteindre sa bonne position. Vous êtes invités à écrire la fonction annexe `void heapify(int* heap, size_t n, size_t pos)` qui suppose que la structure de tas est vérifiée pour les nœuds de $pos + 1$ jusqu'à n et modifie le vecteur pour que cette propriété soit vraie au nœud pos (et donc pour tout nœud de pos à n).

Question 2 : Écrivez les fonctions `int check_heap(const int* heap, size_t n)` qui vérifie qu'un vecteur possède la propriété du tas, et `void prettyprint_heap(const int* heap, size_t n)` qui affiche le tas avec un parcours profondeur selon le format plus bas.

```
void heapify(int* vector, size_t n, size_t pos);
void make_heap(int* vector, size_t n);
```

```
int check_heap(const int* heap, size_t n);
void prettyprint_heap(const int* heap, size_t n);
```

Exemple :

```
int v[] = { 4, 7, 2, 1, 9, 7, 8, 7, 6 };
int size = sizeof(v) / sizeof(int);
make_heap(v, size);
prettyprint_heap(v, size);
assert(check_heap(v, size));
```

Et sa sortie :

```
9
|-- 7
  |-- 6
    |-- 1
    |-- 4
  |-- 7
|-- 8
  |-- 7
  |-- 2
```

2 Fonçons dans le tas

Question 3 : Nous allons maintenant implémenter les fonctionnalités de base des tas.

- `int top_heap(const int* heap, size_t n)` qui retourne le maximum du tas (supposé non vide).
- `void pop_heap(int* heap, size_t* n)` qui supprime la valeur maximal du tas. Elle décrémente la taille n du tas après suppression du nœud. L'utilisateur est responsable de passer un tas non-vide.
- `void push_heap(int* heap, size_t* n, int value)` qui insert une nouvelle valeur dans le tas. Pour ce faire, un nouveau nœud est ajouté en feuille à la position `head[n]` puis la valeur est remontée jusqu'à l'obtention de la propriété du tas. L'utilisateur est reponsable de l'allocation suffisante du tas de façon à l'écriture de `heap[n]` soit valide.
- `void update_heap(int* heap, size_t n, size_t pos, int value)` qui met à jour le nœud à l'indice *pos* avec la valeur *value*. La structure de tas doit être conservée après cette opération.

```
int v[100] = { 4, 7, 2, 1, 9, 7, 8, 7, 6 };
size_t size = 9;
make_heap(v, size);
printf("Insert␣11␣and␣7");
push_heap(v, &size, 11);
push_heap(v, &size, 7);
prettyprint_heap(v, size);
printf("Pop␣min␣and␣update␣pos␣3␣with␣0");
pop_heap(v, &size);
update_heap(v, size, 3, 0);
```

Sortie attendue :

```
Insert 11 and 7
11
|-- 9
  |-- 6
    |-- 1
    |-- 4
  |-- 7
    |-- 7
    |-- 7
|-- 8
  |-- 7
  |-- 2
Pop min and update pos 3 with 0
9
|-- 7
  |-- 4
    |-- 1
    |-- 0
  |-- 7
    |-- 7
|-- 8
  |-- 7
  |-- 2
```

3 Trié sur le tas

Question 4 : Ecrire la fonction `void sort_heap(int* heap, size_t n)` qui à partir d'un tableau ayant la structure de tas, modifie ce tableau pour le trier. Le tri doit être effectué en place.

Exemple :

```
int v[] = { 4, 7, 2, 1, 9, 7, 8, 7, 6 };
size_t size = 9;
make_heap(v, size);
sort_heap(v, size);
for (int i = 0; i < size - 1; ++i)
  printf("%i,", v[i]);
printf("%i\n", v[size-1]);
```

Sortie :

```
1,2,4,6,7,7,7,8,9
```

4 Généralisation

Question 5 : A la manière des TP précédents, modifiez l'ensemble des fonctions pour qu'elles puissent prendre une fonction de comparaison du style `int compare(void* a, void* b)` et qu'elle puissent travailler sur autre chose que des tableau d'entier.

Après ces modifications, le tas doit avoir cette interface :

```
typedef int (*cmp_fun_t)(void*, void*);

void heapify(void* tab, size_t n, size_t pos,
             size_t size, cmp_fun_t cmp);
void make_heap(void* tab, size_t n,
               size_t size, cmp_fun_t cmp);
int  check_heap(const void* heap, size_t n,
                size_t size, cmp_fun_t cmp);
void prettyprint_heap(const void* heap, size_t n,
                      size_t size, (const char*) (*printer)(void));
void* top_heap(const void* heap, size_t* n,
               size_t size); // pas très utile...
void pop_heap(void* heap, size_t* n,
              size_t size, cmp_fun_t cmp);
void push_heap(void* heap, size_t* n, void* value,
               size_t size, cmp_fun_t cmp);
void update_heap(void* heap, size_t n, size_t pos,
                 void* value, size_t size, cmp_fun_t cmp);

void sort_heap(void* heap, size_t n,
               size_t size, cmp_fun_t cmp);
```

5 Files de priorité

Question 6 : Les files de priorités sont implémentées efficacement en utilisant les tas. Nous considérons ici qu'une file de priorité est un tas qui gère lui-même sa mémoire, connaît sa taille et sa fonction de comparaison.

Implémentez l'interface suivante :

```
typedef int (*cmp_fun_t)(void*, void*);

typedef struct
{
  heap_data_t*  m_heap;
  size_t        m_heap_size;
  size_t        m_element_size;
  cmp_fun_t     m_cmp;
} pqueue;

pqueue make_pqueue(size_t element_size, heap_cmp_fun_t cmp);
void   free_pqueue(pqueue* q);
void   push_pqueue(pqueue* q, void* value);
void   pop_pqueue(pqueue* q);
void*  top_pqueue(const pqueue* q);
size_t size_pqueue(const pqueue* q);
```

où :

- `make_pqueue` crée une file de priorité vide,
- `make_pqueue` détruit une file de priorité,

- `push_pqueue` ajoute un élément (en réallouant `q->m_heap` si nécessaire)
- `pop_pqueue` supprime l'élément de priorité maximale.
- `top_pqueue` retourne l'élément de priorité maximale.
- `size_pqueue` retourne le nombre d'éléments dans la file.