

**Solution 1 :** Files de priorités et tas

**1.a]** L'insertion et la suppression ont une complexité en  $\Theta(h)$ , la recherche du maximum en  $\Theta(1)$ .

**1.b]** Comme dans le TD précédent, nous avons  $h \geq \lfloor \log_2(n) \rfloor$ . Cependant, un tas est un arbre complet donc nous avons toujours  $h = \lfloor \log_2(n) \rfloor$ , donc les opérations d'insertion et de suppression ont une complexité  $\Theta(\log n)$ .

**1.c]** Voici le code obtenu :

---

```
1 typedef struct {
2     int max;
3     int n;
4     int* tab;
5 } heap;
6
7 /* Création d'un tas vide de taille m (et allocation de la mémoire) */
8 heap* init_heap (int m) {
9     heap* H = (heap*) malloc(sizeof(heap));
10    H->max = m;
11    H->n = 0;
12    H->tab = (int*) malloc(m * sizeof(int));
13    return H;
14 }
15
16 /* Impression (en largeur) du contenu d'un tas */
17 void print_heap (heap* H) {
18     for (int i=0; i<H->n; i++) {
19         printf("%d ", H->tab[i]);
20     }
21     printf("\n");
22 }
23
24 /* maximum d'un tas */
25 int heap_maximum (heap* H) {
26     if (H->n == 0) {
27         printf("Erreur : tas vide.\n");
28         exit(1);
29     }
30     return H->tab[0];
31 }
32
33 /* fonction pour échanger deux cases du tableau */
34 void swap(int* a, int* b) {
```

```

35  int tmp = (*a);
36  (*a)=(*b);
37  (*b)=tmp;
38  }
39
40  /* insertion de la clef k dans un tas */
41  void heap_insert (int k, heap* H) {
42  if (H->n == H->max) {
43  printf("Erreur : tas plein.\n");
44  exit(2);
45  }
46  int i = H->n;
47  H->tab[i] = k;
48  while ((i>0) && (H->tab[i] > H->tab[(i-1)/2])) {
49  swap(&(H->tab[i]), &(H->tab[(i-1)/2]));
50  i = (i-1)/2;
51  }
52  H->n++;
53  }
54
55  int main (int argc, char* argv[]) {
56  heap* H = init_heap(31);
57  heap_insert(55, H);
58  heap_insert(24, H);
59  ...
60  heap_insert(7, H);
61  print_heap(H);
62  heap_insert(43, H);
63  print_heap(H);
64  }

```

---

**1.d]** La fonction de suppression :

---

```

1  int heap_delete(heap* H) {
2  if (H->n == 0) {
3  printf("Erreur : tas vide.\n");
4  exit(3);
5  }
6  int max = H->tab[0];
7  int i = 0;
8  bool cont = 1;
9  H->n--;
10 /* on met la clef du dernier noeud à la racine */
11 H->tab[0] = H->tab[H->n];
12 while (cont) {
13 if (2*i+2 > H->n) {
14 /* si le noeud i n'a pas de fils */
15 cont = 0;
16 } else if (2*i+2 == H->n) {
17 /* si le noeud i a un seul fils (gauche)
18 on inverse les deux si nécessaire */
19 if (H->tab[i] < H->tab[2*i+1]) {
20 swap(&(H->tab[i]), &(H->tab[2*i+1]));
21 }

```

```

22     cont = 0;
23 } else {
24     /* si le noeud i a deux fils
25        on regarde si l'un des deux est plus grand */
26     if ((H->tab[i] < H->tab[2*i+1]) ||
27         (H->tab[i] < H->tab[2*i+2])) {
28         /* on cherche le fils le plus grand */
29         int greatest;
30         if (H->tab[2*i+1] > H->tab[2*i+2]) {
31             greatest = 2*i+1;
32         } else {
33             greatest = 2*i+2;
34         }
35         /* on inverse et on continue la boucle */
36         swap(&(H->tab[i]),&(H->tab[greatest]));
37         i = greatest;
38     } else {
39         cont = 0;
40     }
41 }
42 }
43 return max;
44 }

```

---

## Solution 2 : Tri par tas

**2.a]** L'insertion des  $n$  éléments :  $n \times \Theta(\log n)$ , puis répéter  $n$  fois le calcul du maximum, en  $\Theta(1)$ , et sa suppression, en  $\Theta(\log n)$ . Soit au total une complexité de  $\Theta(n \times \log n)$ , en moyenne ainsi que dans le pire des cas.

**2.b]** Voici le code du tri par tas :

---

```

1 void heapsort(int* tab, int n) {
2     int i;
3     heap* H = (heap*) malloc(sizeof(heap));
4     H->max = n;
5     H->n = 0;
6     H->tab = tab;
7     for (i=0; i<n; i++) {
8         heap_insert(tab[i],H);
9     }
10    for (i=n-1; i>=0; i--) {
11        tab[i] = heap_delete(H);
12    }
13    free(H);
14 }

```

---

## Solution 3 : Implémentation d'un arbre AVL

**3.a]** Voici la structure à utiliser :

---

```

1 typedef struct st_node{
2     int v;

```

```

3  struct st_node* fg;
4  struct st_node* fd;
5  int h;
6  int eq;
7  } node;

```

---

**3.b]** Voici la fonction d'insertion qui recalcule les coefficients d'équilibrage. Pour simplifier, on utilise une fonction `int height(node* A)` qui retourne la hauteur d'un arbre, y compris l'arbre vide.

```

1  int height(node* A) {
2  if (A == NULL) {
3      return -1;
4  }
5  return A->h;
6  }
7
8
9  void insert(node** A, int v) {
10 if ((*A) == NULL) {
11     /* on crée le nouveau noeud */
12     (*A) = (node*) malloc(sizeof(node));
13     (*A)->v = v;
14     (*A)->fg = NULL;
15     (*A)->fd = NULL;
16     (*A)->h = 0;
17     (*A)->eq = 0;
18 } else {
19     /* on insère à la position définie par la propriété d'ABR */
20     if (v < (*A)->v) {
21         insert(&((*A)->fg), v);
22     } else {
23         insert(&((*A)->fd), v);
24     }
25     /* on met à jour la hauteur et l'équilibrage */
26     if (height((*A)->fg) > height((*A)->fd)) {
27         (*A)->h = 1 + height((*A)->fg);
28     } else {
29         (*A)->h = 1 + height((*A)->fd);
30     }
31     (*A)->eq = height((*A)->fd) - height((*A)->fg);
32 }
33 }

```

---

**3.c]** Voici la fonction de rotation à gauche (rot\_D est presque identique) :

```

1  void rot_G(node** A) {
2  node* tmp;
3  if ((*A) != NULL && ((*A)->fd != NULL)) {
4      /* on modifie les pointeurs pour effectuer la rotation */
5      tmp = (*A)->fd;
6      (*A)->fd = tmp->fg;

```

```

7     tmp->fg = (*A);
8     (*A) = tmp;
9     /* on recalcule l'équilibrage de (*A)->fg, l'ancienne racine */
10    if (height((*A)->fg->fg) > height((*A)->fg->fd)) {
11        (*A)->fg->h = 1 + height((*A)->fg->fg);
12    } else {
13        (*A)->fg->h = 1 + height((*A)->fg->fd);
14    }
15    (*A)->fg->eq = height((*A)->fg->fd) - height((*A)->fg->fg);
16    /* on recalcule l'équilibrage de (*A), l'ancien fils droit */
17    if (height((*A)->fg) > height((*A)->fd)) {
18        (*A)->h = 1 + height((*A)->fg);
19    } else {
20        (*A)->h = 1 + height((*A)->fd);
21    }
22    (*A)->eq = height((*A)->fd) - height((*A)->fg);
23 }
24 }

```

---

**3.d]** Voici la nouvelle fonction d'insertion avec rotation (si besoin). Il suffit de vérifier la valeur du coefficient d'équilibrage après l'avoir mise à jour et de faire les rotations adaptées :

```

1 void insert(node** A, int v) {
2     if ((*A) == NULL) {
3         /* on crée le nouveau noeud */
4         (*A) = (node*) malloc(sizeof(node));
5         (*A)->v = v;
6         (*A)->fg = NULL;
7         (*A)->fd = NULL;
8         (*A)->h = 0;
9         (*A)->eq = 0;
10    } else {
11        /* on insère à la position définie par la propriété d'ABR */
12        if (v < (*A)->v) {
13            insert(&((*A)->fg), v);
14        } else {
15            insert(&((*A)->fd), v);
16        }
17        /* on met à jour la hauteur et l'équilibrage */
18        if (height((*A)->fg) > height((*A)->fd)) {
19            (*A)->h = 1 + height((*A)->fg);
20        } else {
21            (*A)->h = 1 + height((*A)->fd);
22        }
23        (*A)->eq = height((*A)->fd) - height((*A)->fg);
24        if ((*A)->eq == -2) {
25            /* l'arbre est trop haut à gauche */
26            if (height((*A)->fg->fd) > height((*A)->fg->fg)) {
27                /* l'arbre "du milieu" est le plus gros -> double rotation */
28                rot_G(&((*A)->fg));
29            }
30            /* dans tous les cas on finit par une rotation à droite */
31            rot_D(A);
32        } else if ((*A)->eq == 2) {

```

```
33     /* l'arbre est trop haut à droite */
34     if (height((*A)->fd->fg) > height((*A)->fd->fd)) {
35         rot_D(&((*A)->fd));
36     }
37     rot_G(A);
38 }
39 }
40 }
```

---