IN 101 - TD 12
9 décembre 2011

Corrigé

ENSTA ParisTech

Solution 1 : Implémentation d'un tas

1.a] L'insertion et la suppression ont une complexité en $\Theta(h)$: il faut faire une opération par niveau du tas. La recherche du maximum en $\Theta(1)$ car il suffit de regarder la racine.

1.b] Comme dans le TD 10, 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    int 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 : Recherche des k plus grands éléments

3.a] Cet algorithme utilise un tas ayant une capacité de k éléments. Pour extraire les k plus grands éléments il faut utiliser un tas ayant l'ordre inverse : le plus petit élément à la racine. Ensuite l'algorithme fonctionne ainsi :

– les k premiers éléments lus sont ajoutés au tas,
– on compare chaque élément suivant à la racine du tas (le plus petit élément stocké),
  – s'il est plus petit, on ne fait rien avec,
  – s'il est plus grand, il fait partie des k plus grands éléments rencontrés jusqu'à présent et on extrait la racine du tas pour insérer ce nouvel élément.
– à la fin du flux, il suffit d'extraire les éléments du tas un par un et de les afficher.

La complexité temporelle de cet algorithme est bien $\Theta(n \log k)$ car on fait (dans le pire cas) n suppressions et n insertions dans un tas de taille k (et donc de hauteur $\log k$). On ne stocke que k éléments à la fois, donc on a bien une complexité spatiale de $\Theta(k)$.

3.b] Voici le code du `main` qui fait cela. On suppose que les inférieur/supérieur ont été inversés dans les fonctions `heap_insert` (ligne 48) et `heap_delete` (lignes 19, 26, 27 et 30).

```
1  int main(int argc, char* argv[]) {
2    int k=10;
3    int num;
4    heap* H = init_heap(k);
5    FILE* input = (argc == 2) ? fopen(argv[1], "r"): NULL;
6    if (input == NULL) {
7      fprintf(stderr, "Donnez le nom du fichier d'entrie.\n");
8      return 1;
9    }
10   while (fscanf(input, " %d ", &num) == 1) {
11     if (H->n < k) {
12       heap_insert(num, H);
13     } else if (H->tab[0] < num) {
14       heap_delete(H);
15       heap_insert(num, H);
16     }
17   }
18   while (H->n > 0) {
19     printf("%d ", heap_delete(H));
20   }
21   printf("\n");
22 }
```

Solution 4 : Implémentation d'un arbre AVL

4.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;
```

4.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.

```
int height(node* A) {
  if (A == NULL) {
    return -1;
  }
  return A->h;
}


void insert(node** A, int v) {
  if ((*A) == NULL) {
    /* on crée le nouveau noeud */
    (*A) = (node*) malloc(sizeof(node));
    (*A)->v = v;
    (*A)->fg = NULL;
    (*A)->fd = NULL;
    (*A)->h = 0;
    (*A)->eq = 0;
  } else {
    /* on insère à la position définie par la propriété d'ABR */
    if (v < (*A)->v) {
      insert(&((*A)->fg), v);
    } else {
      insert(&((*A)->fd), v);
    }
    /* on met à jour la hauteur et l'équilibrage */
    if (height((*A)->fg) > height((*A)->fd)) {
      (*A)->h = 1 + height((*A)->fg);
    } else {
      (*A)->h = 1 + height((*A)->fd);
    }
    (*A)->eq = height((*A)->fd) - height((*A)->fg);
  }
}
```

4.c] Voici la fonction de rotation à gauche (`rot_D` est presque identique) :

```
void rot_G(node** A) {
  node* tmp;
  if (((*A) != NULL) && ((*A)->fd != NULL)) {
    /* on modifie les pointeurs pour effectuer la rotation */
    tmp = (*A)->fd;
    (*A)->fd = tmp->fg;
    tmp->fg = (*A);
    (*A) = tmp;
    /* on recalcule l'équilibrage de (*A)->fg, l'ancienne racine */
    if (height((*A)->fg->fg) > height((*A)->fg->fd)) {
      (*A)->fg->h = 1 + height((*A)->fg->fg);
    } else {
      (*A)->fg->h = 1 + height((*A)->fg->fd);
    }
    (*A)->fg->eq = height((*A)->fg->fd) - height((*A)->fg->fg);
    /* on recalcule l'équilibrage de (*A), l'ancien fils droit */
    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  }
```

4.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 }
```