TD XXVI

ARBRES BINAIRES DE RECHERCHE EN C

Le but de cette séance est de programmer les opérations élémentaires sur les arbres binaires de recherche en C. Pour la grande majorité, nous les avons déjà vues en cours, mais l'idée est de :

- commencer par essayer d'écrire la fonction sans consulter le cours ;
- au besoin, consulter la description de l'algorithme dans le cours ;
- en désespoir de cause, consulter le code.

Nous allons utiliser le même type qu'en cours, mais avec un emballage supplémentaire :

```c
typedef int item;

struct Node {
  item key;
  struct Node *left;
  struct Node *right;
};

typedef struct Node node;

struct BST {
    node *root;
};

typedef struct BST bst;
```

- Le type `node` correspond à un nœud (et donc à un sous-arbre), et il s'agit d'un type « interne » : les utilisateurs de la structure de donnée n'y auraient pas accès. Les fonctions qui manipulent des nœuds seront préfixées par `node_`.
- Le type `bst` correspond à un arbre binaire de recherche entier, sous forme d'un pointeur vers sa racine. Toutes les fonctions destinées à être fournies aux utilisateurs manipulent uniquement des `bst` et sont préfixées par `bst_`.
- Un nœud vide est représenté par un pointeur nul de type `node*` ; un *arbre* vide est un `bst*` dans lequel le champ `root` contient un pointeur nul.

Exercice XXVI.1 — Fonctions utilitaires p. 4

1. Écrire la fonction `new_node` créant un nouveau nœud, avec la clé fournie.
2. Écrire la fonction `bst_make_empty` renvoyant un nouvel arbre, vide.
3. Écrire la fonction `node_free` libérant la mémoire utilisée par un nœud ainsi que par tous ses descendants.
4. Écrire la fonction `bst_free` libérant la mémoire utilisée par un `bst`.

```c
node *new_node(item x);
bst *bst_make_empty(void);
void node_free(node *n);
void bst_free(bst *t);
```

Exercice XXVI.2 – Insertion et construction

p. 5

1. Écrire la fonction `node_insert` qui ajoute un élément à un sous-arbre. Cette fonction renverra la racine du sous-arbre modifié, et n'aura aucun effet si l'élément à ajouter est déjà présent dans le sous-arbre.
2. Écrire la fonction `bst_insert` qui ajoute un élément à un arbre binaire de recherche.
3. Écrire la fonction `bst_from_array` qui construit un arbre binaire de recherche en insérant un par un tous les éléments d'un tableau, dans l'ordre.

```
node *node_insert(node *t, item x);
void bst_insert(bst *t, item x);
bst *bst_from_array(item arr[], int len);
```

Exercice XXVI.3 – Parcours

p. 5

1. Écrire deux fonctions `node_min` et `bst_min` permettant de déterminer le minimum d'un arbre binaire de recherche. On utiliser une assertion pour arrêter l'exécution de manière déterministe si l'arbre est vide.
2. Écrire deux fonctions `node_member` et `bst_member` testant si une certaine clé est présente dans un arbre.
3. Écrire deux fonctions `node_size` et `bst_size` renvoyant le nombre d'étiquettes d'un arbre.
4. Écrire deux fonctions `node_height` et `bst_height` renvoyant la hauteur. La hauteur d'un arbre vide vaut -1.
5. Écrire une fonction `node_write_to_array` qui écrit les étiquettes présentes dans un sous-arbre dans le tableau fourni.
   - Les étiquettes seront écrites en ordre croissant.
   - La valeur initiale de `*offset_ptr` indique l'indice du tableau dans lequel il faudra écrire la première (la plus petite) étiquette.
   - La fonction devra avoir l'effet secondaire suivant : après l'appel, `*offset_ptr` indique l'indice immédiatement après celui de la dernière case dans laquelle on a écrit au cours de l'appel.
6. Écrire la fonction `bst_to_array` qui renvoie un tableau contenant les étiquettes d'un arbre binaire de recherche en ordre croissant. Cette fonction modifiera la valeur pointée par `nb_elts` pour qu'elle soit égale au nombre d'étiquettes (et donc à la taille du tableau renvoyé).

   Remarque

   `nb_elts` est purement un « argument de sortie » : sa valeur au début de l'appel n'a aucune importance.

```
item node_min(node *n);
item bst_min(bst *t);
bool node_member(node *n, item x);
bool bst_member(bst *t, item x);
int node_size(node *n);
int bst_size(bst *t);
int node_height(node *n);
int bst_height(bst *t);
void node_write_to_array(node *n, item arr[], int *offset_ptr);
item *bst_to_array(bst *t, int *nb_elts);
```

Exercice XXVI.4 — Suppression

1. Écrire la fonction `node_extract_min`. Cette fonction supprime le minimum du sous-arbre passé en argument, et renvoie la nouvelle racine. De plus, elle a un effet secondaire : `*min_ptr` doit être égal à l'étiquette du minimum après l'appel.
2. Écrire les fonctions `node_delete` et `bst_delete`.

```
node *node_extract_min(node *n, int *min_ptr);
node *node_delete(node *n, item x);
void bst_delete(bst *t, item x);
```

Exercice XXVI.5 — Détermination expérimentale de la hauteur moyenne

À l'aide des deux fonctions fournies dans le squelette, écrire un programme ayant le comportement suivant :

- il attend deux arguments entiers en ligne de commande, `max_power` et `rep_count` ;
- pour chaque entier k entre 4 et `max_power`, il génère `rep_count` arbres binaires de recherche de taille 2^k en insérant les éléments de $[0 \dots 2^k - 1]$ dans un ordre aléatoire ;
- pour chacun de ces arbres, il calcule la hauteur ;
- pour chacun des k, il affiche une ligne contenant la valeur de 2^k et la valeur moyenne de la hauteur, séparées par une espace.

On pourra ensuite générer un graphique semi-log à l'aide du script Python fourni, et conjecturer un équivalent de l'espérance de la hauteur en fonction de la taille n de l'arbre.

Solutions

Correction de l'exercice XXVI.1 page I

```c
node* new_node(item x){
  node* n = malloc(sizeof(node));
  n->key = x;
  n->left = NULL;
  n->right = NULL;
  return n;
}

bst *bst_make_empty(void){
    bst *t = malloc(sizeof(bst));
    t->root = NULL;
    return t;
}

void node_free(node *n){
  if (n == NULL) return;
  node_free(n->left);
  node_free(n->right);
  free(n);
}

void bst_free(bst *t){
    node_free(t->root);
    free(t);
}
```

Correction de l'exercice XXVI.2 page 2

```c
node *node_insert(node *t, item x){
  if (t == NULL) {
    return new_node(x);
  }
  if (t->key < x) {
    t->right = node_insert(t->right, x);
  }
  else if (t->key > x) {
    t->left = node_insert(t->left, x);
  }
  return t;
}

void bst_insert(bst *t, item x){
    t->root = node_insert(t->root, x);
}

bst *bst_from_array(item arr[], int len){
  bst *t = bst_make_empty();
  for (int i = 0; i < len; i++){
    bst_insert(t, arr[i]);
  }
  return t;
}
```

Correction de l'exercice XXVI.3 page 2

1.

```c
item node_min(node *n){
  assert (n != NULL);
  while (n->left != NULL){
    n = n->left;
  }
  return n->key;
}

item bst_min(bst *t){
    return node_min(t->root);
}
```

2.

```c
bool node_member(node *n, item x){
    if (n == NULL) return false;
    item key = n->key;
    if (x == key) return true;
    if (x < key) return node_member(n->left, x);
    return node_member(n->right, x);
}
```

```c
bool bst_member(bst *t, item x){
    return node_member(t->root, x);
}
```

3.

```c
int node_size(node *n){
    if (n == NULL) return 0;
    return node_size(n->left) + node_size(n->right) + 1;
}

int bst_size(bst *t){
    return node_size(t->root);
}
```

4.

```c
int max(int x, int y){
    if (x <= y) return x;
    return y;
}

int node_height(node *n){
    if (n == NULL) return -1;
    return 1 + max(node_height(n->left), node_height(n->right));
}

int bst_height(bst *t){
    return node_height(t->root);
}
```

5.

```c
void node_write_to_array(node *n, item arr[], int *offset_ptr){
    if (n == NULL) return;
    node_write_to_array(n->left, arr, offset_ptr);
    arr[*offset_ptr] = n->key;
    *offset_ptr = *offset_ptr + 1;
    node_write_to_array(n->right, arr, offset_ptr);
}
```

6.

```c
item *bst_to_array(bst *t, int *nb_elts){
    int len = node_size(t->root);
    *nb_elts = len;
    item *arr = malloc(len * sizeof(item));
    int offset = 0;
    node_write_to_array(t->root, arr, &offset);
    return arr;
}
```

Correction de l'exercice XXVI.4 page 3

1.

```c
node *node_extract_min(node *n, int *min_ptr){
  assert(n != NULL);
  if (n->left == NULL){
    node *result = n->right;
    free(n);
    *min_ptr = n->key;
    return result;
  }
  return node_extract_min(n->left, min_ptr);
}
```

2.

```c
node *node_delete(node *n, item x){
    if (n == NULL) return n;
    if (x < n->key) {
        n->left = node_delete(n->left, x);
        return n;
    }
    if (x > n->key) {
        n->right = node_delete(n->right, x);
        return n;
    }
    if (n->left == NULL) {
        node *result = n->right;
        free(n);
        return result;
    }
    if (n->right == NULL) {
        node *result = n->left;
        free(n);
        return result;
    }
    item successor = 0;
    node_extract_min(n->right, &successor);
    n->key = successor;
    return n;
}

void bst_delete(bst *t, item x){
    t->root = node_delete(t->root, x);
}
```

Correction de l'exercice XXVI.5 page 3

On ne met que la partie à ajouter au squelette :

```c
bst *random_bst(item items[], int len){
    shuffle(items, len);
    return bst_from_array(items, len);
}

int average_height(int size, int rep_count){
    item *arr = malloc(size * sizeof(item));
    for (int i = 0; i < size; i++){
        arr[i] = i;
    }

    int sum = 0;
    for (int i = 0; i < rep_count; i++){
        bst *t = random_bst(arr, size);
        sum += bst_height(t);
        bst_free(t);
    }
    return (float)sum / rep_count;
}

int main(int argc, char* argv[]){
    assert(argc == 3);
    int pow_max = atoi(argv[1]);
    int rep_count = atoi(argv[2]);

    for (int k = 4; k <= pow_max; k++){
        int len = 1 << k;
        float avg = average_height(len, rep_count);
        printf("%d %f\n", len, avg);
    }

    return 0;
}
```