Informatique *TP*
Structures de données hiérarchiques Ipesup

1 Arbres binaires

1.1 Ecercice

Dessiner tous les arbres binaires ayant respectivement 3 et 4 nœuds.

1.2 Ecercice

Montrer, sans les construire tous, qu'il y a 42 arbres binaires possédant 5 nœuds. De manière générale, donner une définition récursive pour le nombre $C(n)$ d'arbres binaires possédant n nœuds.

1.3 Ecercice

Écrire une fonction qui calcule la hauteur d'un arbre binaire (au choix en OCaml ou en C).

1.4 Ecercice

Écrire une fonction `deepest : 'a bintree -> 'a` qui reçoit un arbre binaire en argument et renvoie l'étiquette d'un nœud de profondeur maximale dans cet arbre. Si l'arbre est vide, on échoue avec une exception. Si plusieurs nœuds sont à la profondeur maximale, on choisit arbitrairement. La complexité doit être linéaire en la taille de l'arbre.

1.5 Ecercice

Écrire une fonction (au choix en OCaml ou en C) qui prend en argument un entier $n \geq 0$ et renvoie un arbre binaire aléatoire contenant exactement n nœuds, par exemple étiquetés avec des caractères pris au hasard dans `'a', ..., 'z'`. On ne cherchera pas à assurer un tirage équiprobable parmi tous les arbres binaires possédant n nœuds, mais on s'assurera que tout arbre binaire possédant n nœuds peut être renvoyé.

1.6 Ecercice

Reprendre l'exercice précédent, mais en assurant cette fois un tirage équiprobable parmi tous les arbres binaires contenant n nœuds.

1.7 Ecercice

Écrire une fonction OCaml `preorder : string bintree -> unit` qui prend en argument un arbre binaire et affiche ses étiquettes (avec `print_string`) selon un parcours préfixe, en utilisant une pile et une boucle `while` plutôt qu'une fonction récursive. On rappelle l'existence du module `Stack` de la bibliothèque standard d'OCaml.

Est-il possible de faire la même chose pour le parcours infixe et le parcours postfixe ?

1.8 Ecercice

Dans cet exercice, on se propose d'écrire une fonction OCaml pour déterminer le i-ème élément, pour l'ordre infixe, dans un arbre binaire. Pour le faire efficacement, on va utiliser des arbres binaires où chaque nœud stocke, en plus de son étiquette, le nombre de nœuds de son sous-arbre. Appelons cela un arbre avec cardinaux.

1. Écrire une fonction `count : 'a bintree -> ('a * int) bintree` qui reçoit en argument un arbre binaire et renvoie un arbre binaire avec cardinaux ayant la même structure et les mêmes étiquettes. La complexité doit être linéaire en la taille de l'arbre.
2. Écrire une fonction `nth : ('a * int) bintree -> int -> 'a` qui reçoit en arguments un arbre binaire avec cardinaux t et un entier i et renvoie le i-ième élément de t pour l'ordre infixe. On suppose les éléments indexés à partir de 0.
3. Discuter la complexité de la fonction `nth`.

2 Arbres binaires de recherche

2.1 Ecercice

Écrire une fonction qui détermine si un arbre binaire est un arbre binaire de recherche. On attend une complexité linéaire en la taille de l'arbre.

2.2 Ecercice

Écrire une fonction qui renvoie la plus petite entrée dans un arbre binaire de recherche. En OCaml, cela peut prendre la forme d'une fonction `min_elt : ('k, 'v) bst -> 'k * 'v` qui lève l'exception `Not_found` si l'arbre est vide ; en C, cela peut prendre la forme d'une fonction `char *bst_min_elt(bst *t)` qui renvoie `NULL` lorsque l'arbre est vide.

2.3 Ecercice

On souhaite écrire une fonction qui supprime une entrée dans un arbre binaire de recherche, pour une clé donnée. Pour cela, on va procéder en deux temps :

1. Écrire une fonction qui supprime la plus petite entrée dans un arbre binaire de recherche non vide.
2. Écrire une fonction qui supprime une clé donnée dans un arbre binaire de recherche. Lorsque la clé à supprimer est la racine, on la remplace par la plus petite entrée du

sous-arbre droit (voir l'exercice précédent), que l'on supprime du sous-arbre droit avec la fonction précédente.

3 Arbres rouge-noir

3.1 Ecercice

Ajouter aux arbres rouge-noir une fonction `min_binding : ('k, 'v) t -> 'k * 'v` qui renvoie l'entrée pour la plus petite clé, si l'arbre est non vide, et lève l'exception `Not_found` sinon.

3.2 Ecercice

On considère la fonction OCaml suivante qui construit et renvoie un tableau contenant n arbres rouge-noir :

```
let all n =
  let a = Array.make n Rbt.empty in
  for i = 1 to n - 1 do a.(i) <- Rbt.add i i a.(i-1) done;
  a
```

Montrer que cette fonction a une complexité $O(n \log n)$ en temps et en espace. Comment expliquer alors que l'on ait pu ainsi construire n structures de données, contenant respectivement $0, 1, \ldots, n-1$ éléments, et n'occupant pour autant pas un espace quadratique au total ?

4 Tas

4.1 Ecercice

Donner une séquence $x_1, x_2, \ldots, x_n$ de valeurs entières telle que l'insertion successive de ces éléments dans un tas initialement vide donne un arbre final de hauteur au moins $n/2$.

4.2 Ecercice

Dans cet exercice, nous étudions une alternative aux files de priorité, où les tas restent cette fois équilibrés après chaque opération. Cette solution repose sur un arbre binaire appelé *arbre de Braun*, où, pour chaque nœud, le sous-arbre gauche possède soit le même nombre d'éléments que le sous-arbre droit, soit un élément de plus.

1. Montrer que pour tout arbre de Braun t non vide, on a $2^{h(t)} \leq n(t) < 2^{h(t)+1}$.
2. Écrire une fonction `insert : 'a -> 'a heap -> 'a heap` qui ajoute un nouvel élément dans le tas. La structure d'arbre de Braun guide naturellement vers la solution.
3. Écrire une fonction `extract : 'a heap -> 'a * 'a heap` qui extrait un élément arbitraire du tas passé en argument (supposé non vide). Là encore, il faut se laisser guider par la structure d'arbre de Braun.

4. Écrire une fonction `replace_min : 'a -> 'a heap -> 'a heap` qui prend en arguments une valeur x et un tas t non vide et renvoie un tas contenant x et tous les éléments de t sauf le plus petit.
5. Enfin, en utilisant les deux fonctions précédentes, écrire une fonction `merge : 'a heap -> 'a heap -> 'a heap` qui effectue la fusion de deux tas ℓ et r, sous l'hypothèse que $n(r) \leq n(\ell) \leq n(r) + 1$.

4.3 Ecercice

Écrire une fonction C `void hamming(int n)` qui affiche les n premiers nombres de Hamming dans l'ordre croissant. Il s'agit des nombres de la forme $2^i 3^j 5^k$ pour i, j, k des entiers naturels (voir page 670). On propose deux approches :

— en utilisant trois files ;
— en utilisant une file de priorité.

4.4 Ecercice

On souhaite ajouter au programme une fonction qui construit une file de priorité contenant tous les éléments d'un tableau a de taille n. On propose l'implémentation suivante :

```
pqueue *pqueue_of_array(int a[], int n) {
  pqueue *q = pqueue_create(n);
  q->size = n;
  for (int i = n / 2; i < n; i++) {
    q->data[i] = a[i];
  }
  for (int i = n / 2 - 1; i >= 0; i--) {
    move_down(q->data, i, a[i], n);
  }
  return q;
}
```

L'idée est ici de construire le tas de bas en haut. Les éléments d'indices $\lfloor n/2 \rfloor, \ldots, n-1$ forment déjà des tas réduits à un unique élément. On se contente de les copier depuis le tableau a. Pour les autres éléments, on utilise `move_down` pour les faire descendre à leur place, en procédant de la droite vers la gauche.

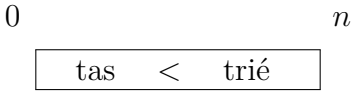
Par un argument simple, borner la complexité de cette fonction par $O(n \log n)$. En détaillant plus finement le coût des différents appels à `move_down`, montrer que cette fonction est en réalité de complexité $O(n)$.

4.5 Ecercice

Écrire une fonction C `void heapsort(int a[], int n)` qui trie un tableau a de n entiers en utilisant une file de priorité. On appelle cela le tri par tas. Donner sa complexité.

4.6 Ecercice

Le tri par tas proposé dans l'exercice précédent a le défaut de devoir utiliser un espace externe aussi grand que le tableau. Dans cet exercice, nous adaptons le tri par tas pour le réaliser *in place*, c'est-à-dire à l'intérieur même du tableau que l'on est en train de trier. On commence par réordonner les éléments du tableau pour qu'ils forment un tas où le plus grand élément se situe à la racine, c'est-à-dire que la relation d'ordre est inversée par rapport au programme vu en cours. Puis on retire les éléments un par un de ce tas, pour les placer dans la partie droite du tableau, du plus grand au plus petit. On a donc pendant cette seconde phase la situation suivante :



où la partie gauche contient un tas formé des éléments non encore triés, tous plus petits que les éléments déjà triés dans la partie droite. Il se trouve que les deux phases de cet algorithme peuvent être écrites avec une unique opération.

1. Écrire une fonction C `void move_down(int a[], int i, int x, int n)` qui écrit la valeur x à la place $a[i]$ puis la fait descendre à sa place dans le tas formé par les éléments $a[0..n[$, les valeurs les plus grandes étant en haut du tas.
2. En déduire un code qui réorganise les éléments d'un tableau a de taille n, en place, pour qu'ils forment un tas. Indication : parcourir le tableau de la droite vers la gauche.
3. En déduire enfin une fonction C `void heapsort(int a[], int n)` qui trie en place les éléments du tableau a et donner sa complexité.

5 Arbres

5.1 Ecercice

Réécrire la fonction `size` du programme vu en cours en utilisant la fonction `List.fold_left` plutôt que la fonction auxiliaire `size_forest`.

5.2 Ecercice

Écrire deux fonctions C `bintree *bintree_of_tree(tree *t)` et `tree *tree_of_bintree(bintree *b)` réalisant l'isomorphisme entre les arbres binaires et les arbres.

6 Arbres préfixes

6.1 Ecercice

Ajouter au programme une fonction `remove` qui supprime l'entrée correspondant à la clé s, si elle existe, et ne fait rien sinon.

6.2 Ecercice

La fonction `remove` de l'exercice précédent peut conduire à des branches vides, i.e. ne contenant plus aucun mot, ce qui dégrade les performances de la recherche. Modifier la fonction `remove` pour qu'elle supprime les branches devenues vides. Il s'agit donc de maintenir l'invariant qu'un champ `branches` ne contient jamais une entrée vers un arbre ne contenant aucun mot. Indication : on pourra se servir avantageusement de la fonction suivante :

```
let is_empty t =
  t.value = None && Hashtbl.length t.branches = 0
```

qui teste si un arbre ne contient aucun mot — à supposer que l'invariant ci-dessus est effectivement maintenu, bien entendu.

6.3 Ecercice

Ajouter au programme une fonction `prefix : 'a trie -> string -> int` qui détermine la longueur du plus grand préfixe d'une chaîne donnée qui est une clé dans un arbre préfixe, et lève l'exception `Not_found` si aucun préfixe n'est une clé dans l'arbre.