DS4 : Arbres, Structures hiérarchiques

15 mars 2024

Vous pouvez réutiliser des résultats des questions précédentes pour une démonstration ou un code, même si la question n'a pas été traitée.

Vous pouvez introduire des fonctions, variables, types ou notations supplémentaires pour produire vos réponses.

Si, au cours de l'épreuve, vous repérez ce qui vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez votre composition en expliquant les raisons des initiatives que vous seriez amené·e à prendre.

Pour la pagination, vous numéroterez toutes les pages de votre composition en faisant figurer le numéro de la page actuelle à partir de 1, ainsi que le total de pages de votre composition. Vous ferez la pagination durant le temps de l'épreuve, et non après la fin de l'épreuve.

Votre nom devra figurer sur chacune de vos copies qui seront rendue imbriquées les unes dans les autres.

Les questions difficiles sont indiquées par une ou plusieurs étoiles (⋆). Cette difficultés est principalement indicative.

Prenez bien soin de lire les consignes et les indications fournies dans le sujet.

Bon courage, et bonne composition.

Problème : Annuaire en C

Ce problème doit être traité dans le langage C.

L'ordre *lexicographique* est l'ordre du dictionnaire sur les chaînes de caractère. On rappelle par ailleurs que dans l'ordre lexicographique, un préfixe est avant le mot dans le dictionnaire, et donc l'assertion `"a"<"ab"` est vraie, mais pas `"a">"ab"`.

Question 1. Proposer une fonction de prototype `int comparer_chaine(const char * a, const char * b)` qui renvoie un entier qui vaut :

— 0 si a et b sont égales ;
— -1 si a est plus grande strictement que b dans l'ordre lexicographique ;
— 1 sinon.

On n'utilisera pas de fonctions de la bibliothèque standard dans cette question.

Correction : On ne peut pas simplement comparer les chaînes de caractères en C, il va falloir utiliser le fait qu'il s'agit de tableaux de caractères de sentinelle nulle.

On parcours les élèments tant qu'il y a des éléments à parcourir dans les deux chaînes, en interrompant le calcul si on a un caractère différent. S'il n'y a plus de caractères dans l'une des chaîne, on vérifie là où on s'est arrêté en premier.

```c
int comparer_chaine(const char * a, const char * b){
    int indice = 0;
    while (a[indice]!='\0' && b[indice]!='\0'){
        if (a[indice]<b[indice]){
            return 1;
        }
        if (a[indice]>b[indice]){
            return -1;
        }
        indice++;
    }
    if (a[indice]==b[indice]){
        return 0;
    }
    if (a[indice]=='\0'){
        return -1;
    }
    return 1;
}
```

On aurait pu améliorer un peu le code en utilisant le fait que la sentinelle est nulle, et qu'elle est inférieur à tous les autres caractères en C.

On cherche à stocker des entiers à l'aide d'un table d'association indéxée par des chaînes de caractères munies de l'ordre lexicographique. Nous utilisons un *arbre binaire de recherche* pour implémenter cette structure.

On appelera *annuaire* une telle structure.

On se donne le type suivant pour stocker les éléments de l'annuaire :

```c
typedef struct _annuaire {
    char * clef;
    int valeur;
    struct _annuaire * fg;
    struct _annuaire * fd;
} annuaire_t;
```

Le champs `clef` contient un tableau de caractères qui correspondent à la clef, le champs `valeur` contient la valeur stockée pour cette clef, et les champs `fg` et `fd` contiennent respectivement un pointeur vers le fils gauche (ou `NULL` s'il n'y en a pas) et un pointeur vers le fils droit du nœud (ou `NULL` s'il n'y en a pas).

Les questions suivantes cherchent à implémenter les opérations d'ajout, de retrait et de recherche en conservant la strucutre d'arbre binaire de recherche pour les clefs dans l'annuaire.

Question 2. Proposer une fonction de signature `int rechercher(annuaire_t * a, const char * c)` qui recherche une clef `c` dans l'annuaire et renvoie la valeur associée.

Dans le cas où la clef n'est pas présente, le programme lèvera une erreur à l'aide d'un appel à `assert`.

Dans les questions suivantes, on prendra soin de remarquer que l'annuaire est fourni sous la forme d'un pointeur vers un pointeur vers l'annuaire. Cela nous permet de modifier l'annuaire dans le cas où on doit rajouter un élément dans l'annuaire vide, ou bien retirer le dernier élément d'un annuaire.

Par exemple, dans la fonction suivante, si le pointeur en entrée pointe vers le pointeur `NULL`, on s'attend à ce que la valeur pointée par le pointeur en entrée devienne un pointeur vers un nouvel annuaire alloué sur le tas.

Question 3. Proposer une fonction de signature `void modifier_annuaire(annuaire_t ** a, const char * c, int v)` qui ajoute dans a la valeur v avec la clef c si la clef n'est pas déjà présente, ou qui modifie la valeur v de la clef c si la clef est déjà présente.

Question 4. Proposer une fonction de prototype `void renvoyer_et_retirer_plus_petit(annuaire_t ** a, int * sortie_valeur, char ** sortie_clef)` qui modifie un annuaire non vide pour en retirer le nœud de plus petite clef et qui modifie les valeurs pointées par `sortie_valeur` et `sortie_clef` pour y mettre les valeur et clef associées au nœud retiré.

Question 5. Proposer une fonction de prototype `void retirer_annuaire(annuaire_t ** a, const char * c)` qui modifie l'annuaire pour en retirer le noeud dont la clef est `c`.

On supposera que la clef est présente dans l'annuaire.

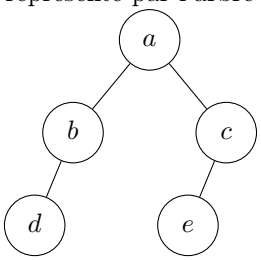
Problème : Tableaux flexibles

Ce problème doit être traité dans le langage OCaml.

Dans ce problème, on représente un tableau par un arbre binaire. Si le tableau est vide, il est représenté par l'arbre vide. Sinon, l'élément d'indice 0 du tableau est stocké à la racine de l'arbre, le sous-arbre gauche contient une représentation des éléments d'indices de la forme $2i+1$ et le sous arbre droit continent une représentation des éléments d'indices de la forme $2i+2$. Ainsi, le tableau de taille 5

a	b	c	d	e

est représenté par l'arbre binaire



avec l'élément a d'indice 0 à la racine, le sous-tableau $[b, d]$ des éléments d'indices impaires récursivement organisé pour former le sous-arbre gauche et le sous-tableau $[c, e]$ des éléments d'indices pairs récursivement organisé pour former le sous-arbre droit.

Question 6. Dessiner l'arbre binaire correspondant à un tableau de taille 12.

Pour la mise en œuvre en OCaml, on suppose que les éléments sont d'un type `elt` donné qui n'est pas précisé. Un tableau flexible est représenté par une valeur du type OCaml suivant :

```
type tree = E | N of tree * elt * tree
```

La valeur `E` représente l'arbre binaire vide. Une valeur $N(l, x, r)$ représente un arbre binaire non vide, avec la valeur x à la racine, un sous-arbre gauche l et un sous-arbre droit r. La taille d'un arbre t, notée

$|t|$, est son nombre de nœeuds. On a donc :

$$|E| = 0$$
$$|N(l, x, r)| = 1 + |l| + |r|$$

Question 7. Écrire une fonction `get: int ->tree ->elt` qui reçoit en arguments un entier i et un tableau flexible t, avec $0 \leqslant i < |t|$, et qui renvoie l'élément d'indice i du tableau représenté par t. La complexité doit être proportionnelle à la hauteur de t. On ne demande pas de la justifier.

On remarque l'invariant structurel suivant : dans un tableau flexible, le sous-arbre gauche a toujours au moins autant de nœuds que le sous-arbre droit, car il y a au moins autant d'indices de la forme $2i+1$ que d'indices de la forme $2i+2$. Plus précisément, pour tout sous-arbre $N(l, x, r)$ d'un tableau flexible, on a

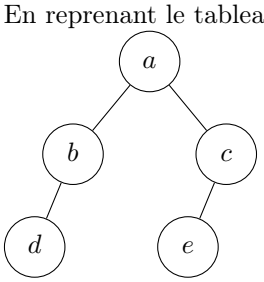
$$|r| \leqslant |l| \leqslant |r| + 1$$

Question 8. Montrer que la hauteur d'un tableau flexible de taille n est en $O(\log n)$.

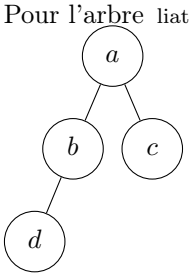
Notre objectif est de réaliser quatre opérations pour respectivement agrandir ou rétrécir le tableau d'une case sur l'une ou l'autre de ses extrémités.

Commençons par le côté droit, c'est-à-dire celui des indices le plus grands. L'opération `liat n t` supprime le dernier élément d'un (arbre représentant le) tableau t de taille n. L'opération `snoc n t x` ajoute un élément x à la fin d'un (arbre représentant le) tableau t de taille n.

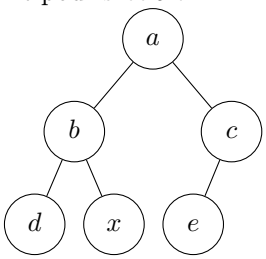
En reprenant le tableau donné en introduction, on obtient pour t :



Pour l'arbre `liat 5 t` :



Et pour `snoc 5 t x` :



Voici une implémentation de la fonction `liat` :

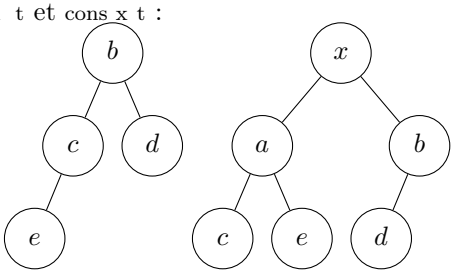
```
let rec liat n t = match t with
| N(E, _, E) -> E
| N(l, x, r) -> if n mod 2 = 0
    then N(liat (n/2) l, x , r)
    else N(l,  x, liat (n/2) r)
| E -> assert false
```

Question 9. Pour un tableau flexible t de taille n, donner la complexité de `liat n t` en fonction de n.

Question 10. Montrer que sous l'hypothèse que t est un tableau flexible de taille $n \geqslant 1$, alors `liat n t` termine et renvoie bien un tableau de taille $n-1$ avec les éléments attendus.

Question 11. Donner un code pour la fonction `snoc: int -> tree -> elt -> tree`. Pour un tableau flexible t de taille n, et un élément x, l'appel `snoc n t x` renvoie un tableau flexible de taille $n+1$ avec les mêmes éléments que t aux indices $0 \leqslant i < n$ et l'élément x à l'indice n. La complexité doit être $O(n)$. On ne demande pas de la justifier.

Agrandir ou rétrécir le tableau du côté gauche, c'est-à-dire du côté des indices les plus petits, est plus délicat à réaliser, car on décale alors les indices des éléments. L'opérations `tail t` supprime le premier élément du tableau t. L'opération `cons x t` ajoute un élément x au début du tableau t. Si on reprend toujours le même exemples, alors ces deux opérations donnent respectivement les arbres suivant pour `tail t` et `cons x t` :



On prendra le temps de bien comprendre ce qui se passe ici. En particulier, on observera comment les éléments passent d'un côté à l'autre de l'arbre. En effet, les éléments situés à des indices pairs (resp. impairs) avant se retrouvent à des indices impairs (resp. pairs) après.

Question 12. Donner un code pour la fonction `tail : tree -> tree`. Pour un tableau flexible t de taille $n \geqslant 1$, l'appel `tail t` renvoie un tableau flexible de taille $n-1$ où le premier élément de t a été supprimé. La complexité doit être en $O(\log n)$ et on demande de la justifier.

Question 13. Donner un code pour la fonction `cons: elt -> tree -> tree`. Pour un tableau flexible t de taille n, l'appel `cons x t` renvoie un tableau flexible de taille $n+1$ où un premier élément x a été ajouté à gauche des éléments de t. La complexité doit être en $O(\log n)$. On ne demande pas de la justifier.

Si on se donne un (vrai) tableau a de type `elt array` et de taille n, on peut chercher à construire un tableau flexible contenant les mêmes éléments que a. Si on le fait naïvement en appelant n fois la fonction `snoc` (ou n fois la fonction `cons`) alors la complexité sera $O(n \log n)$ au total. Mais on peut faire mieux.

Question 14. Écrire une fonction `of_array: elt array -> tree` qui reçoit en argument un tableau a de taille n et construit un tableau flexible ayant les mêmes éléments que a, en temps $O(n)$.

On pourra introduire une fonction auxiliaire qui, pour deux entiers $m \geqslant 1$ et $k \geqslant 0$ donnés, construit le tableau flexible des éléments de a d'indices $mi + k$.

Correction : On utilise une fonction auxiliaire comme proposé dans le sujet. La condition d'arrêt est le fait que le tableau que l'on cherche à construire est vide, c'est-à-dire quand il n'y a pas d'éléments qui correspond au plus petit indice de la forme $mi + k$, c'est-à-dire quand $k \geqslant n$.

```
let construire a =
    let n = Array.length a in
    let rec aux m k =
        if k>= n then E
        else N(aux (2 * m) (k + m), a.(k), aux (2*m) (k + 2 * m))
    in aux 1 0
```

Question 15. Pour cette dernière question, on cherche à construire un tableau flexible d'une certaine taille n dont tous les éléments sont identiques à un élément donné. Comme pour la question précédente, on pourrait le construire à partir des fonctions cons ou snoc, avec une complexité totale $O(n \log n)$. Mais là encore, on peut faire mieux, et même beaucoup mieux !

Écrire une fonction make: int ->elt -> tree qui reçoit en arguments un entier $n \geqslant 0$ et un élément x, et renvoie un tableau flexible de taille n dont tous les éléments sont égaux à x. La complexité doit être $O(\log n)$. On demande de la justifier. Donner par ailleurs la complexité en espace.

Correction : Ici, une mauvaise idée est d'appliquer récursivement la fonction sur $\left\lceil \frac{n}{2} \right\rceil$ et $\left\lfloor \frac{n}{2} \right\rfloor$ car on obtient une mauvaise complexité (un peu à la manière de l'exponentiation rapide). Il faut donc construire dans le cas pair le deuxième arbre à partir du premier arbre.

```
let rec make n elt = match n with
| 0 -> E
| _ when n mod 2 = 1 ->
    let sous_arbre = make (n/2) elt in
    N(sous_arbre, elt, sous_arbre)
| _ ->
    let sous_arbre = make (n/2) elt in
    N(sous_arbre, elt, liat (n/2) sous_arbre)
```

La complexité spatiale est aussi en $O(\log n)$, en effet, à un moment donné, il n'y a au plus que $O(\log n)$ appels sur la pile, que ce soit pour la fonction make ou la fonction liat, et par ailleurs, la structure résultante prend une place en $O(\log n)$.

Ce problème est la partie 2 du sujet de composition de l'agrégation d'informatique 2024.