Université de Lille 1 SCIENCES ET TECHNOLOGIES

ASD

15 mai 2015
Licence SESI - Semestre 4

DS2 - documents de cours, TD, TP autorisés

Note : lorsque, dans une question, est demandée la complexité, c'est une fonction de la taille de la donnée qui est attendue. Lorsque c'est le comportement asymptotique, la réponse attendue est soit une notation $\mathcal{O}$, soit une notation Θ soit une notation Ω.

Vous pourrez utiliser toute fonction vue en cours, TD, TP à la condition expresse de spécifier correctement les entrées et les sorties.

Exercice 1 : Recherche de mots

Compétences évaluées : calculer des complexités, réinvestir des algorithmes connus.

On s'intéresse dans cet exercice à la recherche d'un mot u dans un texte v (on suppose que la longueur de u est plus petite que la longueur de v).

Travail préliminaire. On rappelle qu'un suffixe d'un mot v de longueur n est un sous-mot débutant à une position i, $0 \leq i \leq n$ et contenant toutes les lettres de v de i (compris) jusqu'à la fin du mot. La chaîne vide est un suffixe également.

Par exemple, les suffixes de `"abracadabra"` sont : `"abracadabra"` , `"bracadabra"` , `"racadabra"` , `"acadabra"` , `"cadabra"` , `"adabra"` , `"dabra"` , `"abra"` , `"bra"` , `"ra"` , `"a"` , `""`

Q 1.1 Combien de suffixes contient une chaîne de caractères de longueur n ?

Q 1.2 Ecrire une fonction `tableau_des_suffixes` qui étant donnée une chaine de caractère v de longueur n retourne un tableau trié dans l'ordre lexicographique contenant tous les suffixes de v (à écrire en pseudo code, C ou Caml). Par exemple, pour `"abracadabra"` on obtient le tableau : [`""` , `"a"` , `"abra"` , `"abracadabra"` , `"acadabra"` , `"adabra"` , `"bra"` , `"bracadabra"` , `"cadabra"` , `"dabra"` , `"ra"` , `"racadabra"`]

On s'intéresse à la complexité en temps où l'opération qu'on compte est la comparaison de caractères.

Q 1.3 Décrire un pire des cas. Quel est le comportement asymptotique en fonction de n ?

Q 1.4 Décrire un meilleur des cas. Quel est le comportement asymptotique en fonction de n ?

Algorithme de recherche

Q 1.5 Proposer un algorithme de recherche d'un mot u de taille m dans un texte v de taille n qui exploite le tableau des suffixes.

Q 1.6 Indiquer si il existe ou non un pire des cas, un meilleur des cas. Justifier. Donner la complexité de cet algorithme en fonction de n et m.

Exercice 2 : Arbre équivalents

Compétences évaluées : manipuler des arbres binaires, utiliser des structures de données adéquates pour résoudre un problème.

Dans cet exercice on manipule des arbres binaires dont les étiquettes sont des entiers. Vous pourrez utiliser toutes les fonctions vues au cours du TP numéro 5 sans les redéfinir (voir annexes du sujet).

Arbres à couches équivalentes On dit que deux arbres A et B sont à couches équivalentes si, pour une profondeur donnée p, la somme des étiquettes des noeuds à profondeur p est la même dans A et B.

Par exemple les arbres A et B ci-dessous sont à couches équivalentes

puisque dans les deux arbres, la somme des valeurs à profondeur 0 est 4, à profondeur 1 est 9, à profondeur 2 est 10.

Q 2.1 Quelle(s) structure(s) de données complémentaire utiliser pour construire un algorithme qui détermine si deux arbres binaires sont à couches équivalentes ? Justifier.

Q 2.2 Proposer l'écriture d'un prédicat `a_couches_equivalentes` qui prend en paramètres deux arbres A et B (à écrire en pseudo code, C ou Caml).

Q 2.3 Y a-t-il un pire des cas ? Si oui le donner, sinon justifier.

Q 2.4 Donner le comportement asymptotique dans le pire des cas ou dans le cas général suivant la réponse à la question précédente en fonction du nombre n des noeuds de A et du nombre m des noeuds de B.

Arbres à feuilles équivalentes Etant donné un arbre binaire A, on associe à chaque feuille x de A la fonction $f(x)$ définie comme la somme des valeurs de noeuds sur la branche menant de la racine A à x (racine et feuille incluses).

Etant donné un arbre binaire A, on définit l'ensemble $\mathcal{S}(A) = \{f(x) \text{pour toute feuille } x \text{ de } A\}$.

On dit que deux arbres A et B sont à feuilles équivalentes si A et B ont même nombre de feuilles et que $\mathcal{S}(A) = \mathcal{S}(B)$.

Par exemple, pour les deux arbres A et B ci-dessous

$\mathcal{S}(A) = \{7, 9, 17\}$ et $\mathcal{S}(B) = \{9, 17, 7\}$. Ils sont donc à feuilles équivalentes.

Q 2.5 Quelle(s) structures(s) de données utiliser pour construire un algorithme qui détermine si deux arbres sont à feuilles équivalentes ? Justifier.

Q 2.6 Proposer l'écriture d'un prédicat `a_feuilles_equivalentes` qui prend en paramètres deux arbres A et B (à écrire en pseudo code, C ou Caml).

Q 2.7 Y a-t-il un pire des cas ? Si oui le donner, sinon justifier.

Q 2.8 Donner le comportement asymptotique dans le pire des cas ou dans le cas général suivant la réponse à la question précédente en fonction du nombre n des noeuds de A et du nombre m des noeuds de B.

Exercice 3 : Arbre à insertion par la racine

Compétences évaluées : manipuler des structures de données connues, réinvestir des raisonnements vus en cours.

On considère dans cet exercice des arbres binaires de recherche. L'application qui utilisera ces arbres effectue des opérations de recherche d'étiquettes dans l'arbre. Elle présente la particularité de réaliser plus de recherches sur les derniers éléments insérés que sur les premiers. On aura ainsi tout intérêt à ajouter une nouvelle étiquette à la racine.

Ci-dessous l'exemple d'un arbre avant et après insertion de la valeur 5.

L'arbre résultant est toujours un ABR, mail il a été réorganisé.

La procédure d'ajout peut-être décrite ainsi :

```
ajouter (e,a):
  ag, ad = partitionner_en_arbres (a, e)
  retourner nouveau_noeud (e,ag,ad)
```

où `partitionner_en_arbres` crée deux arbres binaires à partir de l'arbre a en fonction de l'étiquette e. Pour partionner en arbes, nous commencerons par partitionner en deux listes d'étiquettes.

Vous disposez dans cet exercice de toutes les fonctions de manipulations des arbres binaires vues dans le TP numéro 5 (voir annexes du sujet). N'oubliez pas que pour faciliter le travail, il peut être utile, voire nécessaire d'écrire des sous-fonctions.

Q 3.1 Ecrire une fonction `partitionner_en_listes` qui prend en entrée un arbre binaire de recherche a et une étiquette e et retourne un couple de listes, la première contenant toutes les étiquettes de a inférieures ou égales à e, la seconde les autres.

Q 3.2 Ecrire une fonction `partitionner_en_arbres` qui prend en entrée un arbre binaire de recherche a et une étiquette e et retourne un couple d'arbres binaires, le premier contenant toutes les étiquettes inférieures ou égales e de a, le second les autres.

Q 3.3 Donner un encadrement de la hauteur de l'arbre obtenu après un appel à ajouter sur un arbre a contenant n noeuds et de hauteur h. Justifiez très clairement.

Q 3.4 On suppose qu'après l'ajout de $n = 2^p$ étiquettes avec la procédure ajouter, on aboutit à un arbre équilibré. On suppose que la probabilité de recherche de l'étiquette ajoutée à l'étape i est $p(i) = 1 - \frac{1}{2^i}$. Déterminer le coût moyen d'une recherche d'une des étiquettes n présente dans l'arbre.

ANNEXES (`abres.mli` et `ArbreBinaire.h`)

```
(**

    Module de manipulation d'arbres binaires dont les valeurs
    associees aux noeuds sont des entiers.

*)

type arbre

exception ArbreVide
```

```
(*
  [vide] : retourne un arbre vide
*)
val vide : unit -> arbre

(*
  [est_vide] : prédicat de test qu'un arbre est vide
*)
val est_vide : arbre -> bool

(*
  [creer v g d] : retourne un nouvel arbre dont la valeur a la racine est [v] et les fils gauche et droit sont les arb
*)
val creer : int -> arbre -> arbre -> arbre

(*
  [remplacer_gauche a g] : remplace le fils gauche de [a] par le sous-arbre [g]

  Leve l'exception ArbreVide si [a] est Vide
*)
val remplacer_gauche : arbre -> arbre -> unit

(*
  [remplacer_droite a d] : remplace le fils gauche de [a] par le sous-arbre [d]

  Leve l'exception ArbreVide si [a] est Vide
*)
val remplacer_droite : arbre -> arbre -> unit

(*
  [valeur a] : retourne la valeur associé a la racine de [a].

  Leve l'exception ArbreVide si [a] est Vide
*)
val valeur : arbre -> int

(*
  [fils_gauche a] : retourne le sous-arbre gauche de [a]

  Leve l'exception ArbreVide si [a] est Vide
*)
val fils_gauche : arbre -> arbre

(*
  [fils_droit a] : retourne le sous-arbre droit de [a]

  Leve l'exception ArbreVide si [a] est Vide
*)
val fils_droit : arbre -> arbre

(*
  [est_feuille a] : vrai si [a] est une feuille

  Leve l'exception ArbreVide si [a] est Vide
*)
val est_feuille : arbre -> bool
```

```
#define ARBREBINAIRE

#define FILSEXISTANT 1
#define BADARGUMENT 2
#define ARBREVIDE 3

typedef signed long int value_t ;
typedef struct NoeudBinaire_m * Noeud_t ;
```

```
struct NoeudBinaire_m
{
value_t  val ;
Noeud_t  filsgauche ;
Noeud_t  filsdroit ;
} ;


/* cr\'ee un arbre vide */
Noeud_t CreerArbreVide(void) ;

/* cr\'ee une feuille avec une valeur associ\'ee */
Noeud_t CreerNoeud(value_t) ;

/* pr\'edicat de test qu'un noeud est vide */
int EstVide (Noeud_t);

/* pr\'edicat de test qu'un noeud est une feuille.

   CU: le noeud ne doit pas être vide, sinon produit une erreur. */
int EstFeuille (Noeud_t);

/* retourne la valeur associ\'e au noeud pass\'e en param\`etre.

   CU: le noeud ne doit pas \^etre vide */
value_t ValeurDuNoeud(Noeud_t) ;

/* ajoute en fils gauche au noeud premier argument le noeud second
   argument.

   CU: le fils gauche doit \^etre vide, sinon produit une erreur
*/
void AjouterFilsGauche(Noeud_t,Noeud_t) ;

/* ajoute en fils droit au noeud premier argumentle noeud second
   argument.

   CU: le fils droit doit \^etre vide sinon produit une erreur
*/
void AjouterFilsDroit(Noeud_t,Noeud_t) ;

/* remplace le fils gauche du noeud premier argument par le noeud
   second argument.
*/
void RemplacerFilsGauche(Noeud_t,Noeud_t) ;

/* remplace le fils droit du noeud premier argument par le noeud
   second argument.
*/
void RemplacerFilsDroit(Noeud_t,Noeud_t) ;

/* retourne le fils gauche du noeud premier argument */
Noeud_t FilsGauche(Noeud_t);

/* retourne le fils droit du noeud premier argument */
Noeud_t FilsDroit(Noeud_t);
```