Algorithmique des structures de données arborescentes

Feuille 8 - Retour sur les parcours d'arbres.

Rappel : il existe deux types de parcours d'arbres :

- En profondeur :
  - Préfixe : on traite la Racine avant de parcourir les sous-arbres.
  - Suffixe ou postfixe : on traite la Racine après avoir parcouru les sous-arbres.
  - Infixe : on traite la Racine après le sous-arbre gauche et avant le sous-arbre droit.
- En largeur : niveau par niveau.

Exercice 1 : Exemples de parcours

Donner les listes de obtenues en parcourant l'arbre de la figure 8.1 selon les différents parcours d'arbres.

1 Arbres binaires : Parcours en profondeur

Nous avons déjà vu le parcours en profondeur infixe. Le parcours en ordre préfixe est détaillé dans le cours. Nous allons adapter l'algorithme pour le parcours en ordre ***suffixe***.
le parcours en ordre ***suffixe*** d'un arbre binaire produit une liste des nœuds de l'arbre ordonnés de la façon suivante :
Si l'arbre est vide, le parcours en ordre suffixe de l'arbre produit la liste [].
Sinon, le parcours en ordre suffixe de l'arbre est la concaténation, dans cet ordre, des 3 listes suivantes :

- la liste en ordre suffixe des nœuds du sous-arbre gauche,
- la liste en ordre suffixe des nœuds du sous-arbre droit,
- la liste d'un seul élément contenant l'étiquette de la racine.

Exercice 2 : Arbres binaires vers liste suffixe : version "naïve"

Écrire une version "naïve" (mais inefficace) de la fonction `suffixe_list_of_btree` de type `'a btree -> 'a list` qui produit la liste obtenue en parcourant l'arbre en ordre ***suffixe***.
Quelle est la complexité de cette fonction ?

1.1 Utilisation d'une pile

Pour obtenir un parcours en profondeur suffixe efficace, nous devrons maintenir les sous-arbres non traités dans une pile. Rappel sur les piles : on utilise 2 opérations de pile, Push(x) et Pop().

Le parcours en ordre suffixe de l'arbre s'effectuera donc de la façon suivante :

- t = Pop() ;
- si t est une feuille, alors traiter(root(t)), sinon :
  - push(leaf(root(t))) ; (* arbre avec un seul noeud, d'étiquette $root(t)$. *)
  - Push(right(t)) ;
  - Push(left(t)) ;

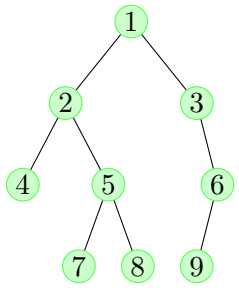

FIG. 8.1 : Arbre binaire t_1

Exercice 3 : Arbres binaires vers liste suffixe : version efficace

Écrire une version efficace de la fonction `suffixe_list_of_btree` qui produit la liste obtenue en parcourant l'arbre en ordre ***suffixe***, en maintenant une pile des sous-arbres non traités.

2 Arbres binaires : Parcours en largeur

le parcours en ***largeur*** d'un arbre binaire produit une liste des nœuds de l'arbre ordonnés de la façon suivante :
Si l'arbre est vide, le parcours en largeur de l'arbre produit la liste [].
Sinon, le parcours en largeur de l'arbre est la concaténation des listes :

- liste des nœuds à niveau 0 (= la racine),
- liste des nœuds à niveau 1 (= les fils de la racine),
- liste des nœuds à niveau 2,
- etc...

Pour obtenir un parcours en largeur efficace, nous devrons maintenir les sous-arbres non traités dans une file. Nous simulerons la file à l'aide de deux piles. Au cours de l'algorithme, les sous-arbres à traiter sont dans la pile 2 (dépilée avec Pop2). On empile dans la pile 1 les sous-arbres en attente de traitement (empilés avec Push1). Quand la pile 2 est vide, on la remplit avec les arbres en attente dans la pile 1.

Le parcours en largeur de l'arbre s'effectuera donc de la façon suivante :

- si pile2 non vide, t = Pop2() ;
- traiter(root(t)) ;
- Push1(left(t)) ;
- Push1(right(t)) ;
- si pile2 vide (ie : fin traitement d'un niveau) :
  pop toute la pile1 et push dans pile2

Exercice 4 : Arbres binaires parcours en largeur : version efficace

Écrire une version efficace de la fonction `btree_bfs` qui produit la liste obtenue en parcourant l'arbre en ***largeur***, en simulant une file des sous-arbres non traités à l'aide de deux piles.