Cours Structures de Données Avancées

Option G.L.
Semestre 3

Chapitre 3 : Structures de données arborescentes

1. Le type abstrait Arbre

Les structures d’arbres possèdent un intérêt particulier dans la structuration des données qui interviennent dans de nombreux problèmes (hiérarchies d’objets, choix et décisions, arbres syntaxiques, etc.). L’usage des arbres est très répandu pour la structuration des informations L’intérêt de la formalisation informatique des arbres réside dans l’ensemble d’algorithmes que l’on peut définir sur eux de manière générale, et qui auront autant de traitements disponibles pour chaque arbre en particulier. On peut par exemple rechercher l’ensemble des descendants, compter les noeuds, savoir si un noeud est ancêtre d’un autre, etc.

En plus, une propriété intrinsèque des arbres est la récursivité. Les définitions des arbres et les algorithmes qui manipulent cette structure s’écrivent très naturellement avec la récursivité.

Définition: Un arbre se compose de deux ensembles N et A appelés respectivement l’ensemble des noeuds et l’ensemble des arcs et d’un noeud particulier r appelé racine de l’arbre. Les éléments de A sont des paires (n1, n2) d’éléments de N. Un arc (n1,n2) établit une relation entre n1, appelé noeud parent, et n2, appelé noeud enfant de n1. A doit être tel que chaque noeud, sauf la racine, a exactement un parent. La racine n’a pas de parent.

La structure d’arbre binaire est utilisée dans de nombreuses applications informatiques; de plus elle permet de représenter les arbre généraux. Dans les paragraphes suivants, nous nous intéressons à l’étude des arbres binaires.

Définition Un Arbre binaire est soit vide (noté ∅), soit de la forme B = <o, B1, B2>, où B1 et B2 sont des arbres binaires disjoints et o est un noeud appelé racine de B.
Il est important de noter la non-symétrie gauche/droite des arbres binaires : l’arbre <o, A,B> et l’arbre <o,B ,A> sont des arbres binaires différents.

TAD Arbre Binaire
Sorte Arbre_binaire
Utilise Noeud, Elément
Opérations:
arbre_vide: −→ Arbre_binaire
<_,_,_>: Noeud, Arbre_binaire, Arbre_binaire −→ Arbre_binaire
gauche: Arbre_binaire −→ Arbre_binaire
droit: Arbre_binaire −→ Arbre_binaire
racine: Arbre_binaire −→ Noeud
contenu: Noeud −→ Elément
Préconditions:
racine(B) est définie ssi B 6= arbre_vide
gauche(B) est définie ssi B 6= arbre_vide
droit(B) est définie ssi B 6= arbre_vide
Axiomes:
racine(<o,B1,B2>) ≡ o
gauche(<o,B1,B2>) ≡ B1

droit(<o,B1,B2>) ≡ B2
Variables:
o: Noeud, B1, B2, B: Arbre_binaire

2. Les parcours d'arbres

Une des opérations les plus fréquentes mise en oeuvre par les algorithmes qui manipulent les arbres consiste à examiner systématiquement, dans un certain ordre, chacun des noeuds de l'arbre pour y effectuer un même traitement. Cette opération est appelée parcours ou traversée de l'arbre. Contrairement à la structure de liste ou de file, il n'y a pas d'ordre canonique pour parcourir un arbre, même ordonné. On distingue deux grandes catégories d'ordre de parcours:

1. Les parcours en profondeur
2. Les parcours en largeur

2.1 Les parcours en profondeur

On étudie dans cette section un algorithme de parcours d'un arbre binaire qu'on appelle parcours en profondeur à gauche sous sa forme itérative et récursive avec les trois ordres induits sur les noeuds. En général, dans le parcours en profondeur, on part de la racine et on descend dans le sous-arbre gauche qu'on explore complètement, puis on passe au sous-arbre droit. Lors du PPG (parcours en profondeur gauche) de l'arbre A, chaque noeud est rencontré trois fois. On peut alors faire correspondre à chacune des trois rencontres un traitement particulier. Appelons traitement1, traitement2, traitement 3 respectivement la suite d'actions exécutées lorsqu'un noeud de A est rencontré pour la première, deuxième et troisième fois respectivement. De plus envisageons pour les arbres vides un traitement spécial terminaison. Le parcours d'arbre est alors décrit par la procédure récursive Parcours suivante:

```
Procédure PPG (donnée/résultat A: arbre)
Début
Si A= arbre_vide alors
  terminaison
sinon
        traitement1(racine(A))
        PPG (gauche(A))
        traitement2(racine(A))
        PPG(droit(A))
        traitement3(racine(A))
fsi
Fin
```

En particulier, il existe trois ordres classiques d'exploration d'un arbre dans le parcours en profondeur:

1- Préfixé (préordre): correspond au cas ou le traitement2 et traitement3 sont des opérations vides.

• traiter la racine,
• parcourir le sous-arbre gauche,
• parcourir le sous-arbre droit.

2- Postfixé (postordre): correspond au cas ou le traitement1 et traitement2 sont des opérations vides.

• parcourir le sous-arbre gauche,
• parcourir le sous-arbre droit,
• traiter la racine.

3- Infixé (en ordre): correspond au cas ou le traitement1 et traitement3 sont des opérations vides.
• parcourir le sous-arbre gauche,
• traiter la racine,
• parcourir le sous-arbre droit.

2.2 Extension du TAD arbre avec les trois types de parcours

Nous enrichissons le TAD Arbre_binaire avec les opérations suivantes et les axiomes correspondants.

Profil:
préfixé: arbre_binaire −→ liste
infixé: arbre_binaire −→ liste
postfixé: arbre_binaire −→ liste
Axiomes:
préfixé(A)≡ si A =arbre_vide alors liste_vide sinon
concatèner(cons(racine(A),préfixé(gauche(A))),préfixé(droit(A)))
infixe(A)≡ si A =arbre_vide alors liste_vide sinon

2.3 Les parcours en largeur

Dans le parcours en largeur, on parcours l'arbre dans l'ordre suivant :

1. La racine,
2. Les noeuds du niveau 1 (de gauche vers la droite),
3. etc.
4. Les noeuds du dernier niveau (de gauche vers la droite).

Nous présentons l'algorithme sous forme itérative de parcours en largeur:

```
Procédure PL(donnée/résultat A: arbre)
Déclarations
F: file /* Une file de noeuds */
T: noeud
Début
        F← file_vide
        F← ajouter(F, racine(A))
        Tantque non(est_vide(F)) faire
        T←premier(F)
        traiter(T)
        F←retirer(F)
        Si non(est_vide(gauche(T))) alors
        F←ajouter(F,racine(gauche(T)))
        fsi
        Si non(est_vide(droit(T))) alors
        F←ajouter(F,racine(droit(T)))
        fsi
        fintq
Fin
```

3. Implémentation des arbres

Nous définissons à ce niveau quelques représentations possibles des arbres binaires.
Il existe tant d'autres. Ce sont les opérations que l'on désire effectuer sur les arbres qui déterminent le type de représentation à choisir.

3.1 Représentation chaînée

Chaque noeud d'un arbre binaire peut être représenté par un enregistrement à trois champs : gauche, droit et racine qui contiendront respectivement un pointeur vers le sous-arbre gauche, un pointeur vers le sous-arbre droit et la valeur du noeud. Evidemment l'arbre binaire dans ce cas est un pointeur vers l'enregistrement contenant la racine.

Donc une déclaration de la structure arbre pourrait être:

- Un champ de type élément pour représenter la racine;
- Deux champs pour les deux pointeurs g, et d pointant vers respectivement le sous arbre gauche et droit;

Programmation des opérations de base

```
Fonction gauche (arbre B): arbre
Début
  retourner(B→g);
Fin
Fonction droit (arbre B): arbre
Début
  retourner(B→d);
Fin

Fonction cons_arbre (élément racine; arbre B1, B2): arbre
/* correspond à l'opération interne constructeur <_,_,_> */
Déclaration
   B: arbre;
 Début
   B→racine=racine;
   B→g=B1;
   B→d=B2;
  retourner(B);
Fin
```

4. Le type abstrait Graphe

Le graphe est une structure relationnelle qui nous permet de représenter de nombreuses situations rencontrées dans le monde réel telles que : Flux de contrôle d'un programme, circuits éléctriques, réseaux de transport (ferriés, routiers, aériens), réseaux d'ordinateurs, ordonnancement d'un ensemble de tâches, etc.

Définition Un graphe G = (X,A) est donné par un ensemble X de sommets et par un sous-ensemble A du produit cartésien X ×X appelé ensemble des arcs de G.

Les éléments de X sont appelés sommets du graphe G.

Dans un premier temps, on définit le TAD Ensemble [Sommet] à partir du TAD ensemble du Chapitre 7, en remplaçant la sorte Elément par la sorte Sommet. Ensuite, on spécifie un TAD Graphe (ensemble[sommet]) de type graphe paramétré par la sorte Ensemble dont les éléments sont de type sommet. Ce TAD est défini de la manière suivante:

```
TAD Graphe
Sorte Graphe
Utilise Booléen, Ensemble, Sommet
Opérations:
Graphe_vide: −→ graphe
ajouter_sommet: graphe, sommet −→ graphe
ajouter_arc: graphe, sommet, sommet −→ graphe
supprimer_sommet: graphe, sommet −→ graphe
supprimer_arc: graphe, sommet, sommet −→ graphe
appartient_sommet: graphe, sommet −→ booléen
```

appartient_arc: graphe, sommet, sommet −→ booléen
ensemble_sommet: graphe −→ ensemble
ensemble_arc: graphe −→ ensemble
ensemble_successeurs: graphe, sommet −→ ensemble
ensemble_prédécesseurs: graphe, sommet −→ ensemble
Préconditions:
Pré(ajouter_sommet(g,x)) définie ssi non (appartient_sommet(g,x))
Pré(ajouter_arc(g,x,y)) définie ssi appartient_sommet(g,x) et
appartient_sommet(g,y) et
non appartient_arc(g,x,y)
Pré(supprimer_sommet(g,x)) définie ssi appartient_sommet(g,x) et
ensemble_successeurs(g,x)=ensemble_vide et
ensemble_prédécesseurs(g,x)=ensemble_vide
134 Les graphes
Pré(supprimer_arc(g,x,y)) définie ssi appartient_sommet(g,x) et
appartient_sommet(g,y) et
appartient_arc(g,x,y)
Pré(appartient_arc(g,x,y)) définie ssi appartient_sommet(g,x) et
appartient_sommet(g,y)
Axiomes:
ensemble_sommet(graphe_vide) ≡ ensemble_vide
ensemble_sommet(ajouter_sommet(g,x)) ≡ ajouter(x,ensemble_sommet(g))
ensemble_sommet(ajouter_arc(g,x,y)) ≡ ensemble_sommet(g)
ensemble_arc(graphe_vide) ≡ ensemble_vide
ensemble_arc(ajouter_sommet(g,x)) ≡ ensemble_arc(g)
ensemble_arc(ajouter_arc(g,x,y)) ≡ ajouter((x,y),ensemble_arc(g))
ensemble_successeurs(graphe_vide,x) ≡ ensemble_vide
ensemble_successeurs(ajouter_sommet(g,x),x’) ≡ si x=x’ alors
ensemble_vide sinon ensemble_successeurs(g,x’)
ensemble_successeurs(ajouter_arc(g,x,y),x’) ≡ si x=x’ alors
ajouter(y, ensemble_successeurs(g,x)) sinon
ensemble_successeurs(g,x’)
ensemble_prédécesseurs(graphe_vide,x) ≡ ensemble_vide
ensemble_prédécesseurs(ajouter_sommet(g,x),x’) ≡ si x=x’ alors
ensemble_vide sinon ensemble_prédécesseurs(g,x’)
ensemble_prédécesseurs(ajouter_arc(g,x,y),x’) ≡ si x=x’ alors
ajouter(x, ensemble_prédécesseurs(g,y)) sinon
ensemble_prédécesseurs(g,x’)
supprimer_sommet(ajouter_sommet(g,x),x’) ≡ si x=x’ alors g sinon
ajouter_sommet(supprimer_sommet(g,x’),x)
supprimer_sommet(ajouter_arc(g,x,y),x’) ≡ ajouter_arc(supprimer_sommet(g,x’),x,y)
supprimer_arc(ajouter_sommet(g,x),x’,y’) ≡ ajouter_sommet(supprimer_arc(g,x’,y’),x)
supprimer_arc(ajouter_arc(g,x,y),x’,y’) ≡ si (x,y)=(x’,y’) alors g sinon
ajouter_arc(supprimer_arc(g,x’,y’),x,y)
appartient_sommet(g,x) ≡ appartient(x,ensemble_sommet(g))
appartient_arc(g,x,y) ≡ appartient((x,y),ensemble_arc(g))
Variables:
x,x’,y,y’: Sommet; g: graphe.

Remarques:

- On peut utiliser la sorte arc pour éviter le produit cartésien (sommet× sommet).
- Le type abstrait graphe peut être spécifié en utilisant le TAD liste de la même façon que précédemment en remplaçant la sorte et les opérations relatives au TAD Ensemble par celles du TAD liste.

En fonction du type d’algorithme que l’on veut utiliser, on considère différentes implémentations des graphes. Les techniques les plus classiques pour représenter un graphe sont classées en deux catégories:

1. Représentations en utilisant les listes (liste de sommets, liste d’arcs, liste d’adjacence,etc.)
2. Représentations matricielles.