Projet Arbre 2-3-4

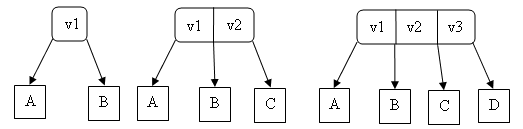
* 1. Objectif

Ce projet a pour but d’implémenter en Coq une structure informatique : les arbres 2-3-4. L’implémentation inclura les spécifications de la structure en elle-même ainsi que les opérations de base de la structure (l’ajout / retrait d’un élément, tests d’appartenance…). Dans un second temps, l’objectif sera de formaliser et prouver des prédicats correspondant notamment aux propriétés de cette structure.

* 1. Definition

Un arbre 2-3-4 est un arbre dont les feuilles sont vide, et dont les nœuds contiennent 1 élément et deux fils(binode) ou bien 2 éléments et 3 fils (trinode) ou bien 3 éléments et 4 fils (quadnode). Les fils étant des arbres 2-3-4.

En ce qui concerne l’ordonnancement des éléments :



Les sous-arbres *A*, *B*, *C* et *D* ont les propriétés suivantes :

* tous les nœuds du sous arbre *A* ont une valeur inférieure à *v1*.
* tous les nœuds du sous arbre *B* ont une valeur supérieure ou égale à *v1* et inférieure à *v2*, pour les nœuds 3 ou 4.
* tous les nœuds du sous arbre *C* ont une valeur supérieure ou égale à *v2* et inférieure à *v3*, pour les nœuds 4.
* tous les nœuds du sous arbre *D* ont une valeur supérieure ou égale à *v3*.

* 1. Specification
     1. Structure

La structure des arbres 2-3-4 est une structure construite par induction avec 4 types d’élément :

Les feuilles, vides, les binodes contenant un élément et deux sous arbres. Les trinodes avec 2 éléments et 3 sous arbres et les quadnodes avec 3 éléments et 4 sous-arbres

Inductive tree (A: Type ) : Type :=

leaf : tree A

|binode: A -> tree A -> tree A -> tree A

|trinode : A -> A -> tree A -> tree A -> tree A -> tree A

|quadnode : A -> A -> A -> tree A -> tree A -> tree A -> tree A -> tree A.

* + 1. Opérations
* L’existence : exist (a:nat)(T:tree nat): bool retourne vrai si l’élément a est dans l’arbre T, faux sinon. Parcours de l’arbre de manière descendante jusqu’à trouver l’élément a en faisant des tests d’ordre sur les différents élémetn des nœuds.
* L’ajout add (a:nat) (T : tree nat): tree nat retourne un arbre qui contient tous les éléments de T et l’élément a. Parcours de l’arbre de manière descendante. Si on rencontre un quadnode on effectue un éclatement.
* Le retrait delete (a:nat)(T: tree nat): tree nat retourne un arbre qui contient tous les éléments de T sauf a. On utilise une fonction intermédiaire qui transforme l’arbre en liste puis on supprime l’élément de cette liste qu’on transforme ensuite en arbre.
* Test de l’équilibre
  + 1. Opérations secondaires
* to\_list (a:nat)(T: tree nat): list nat retourne une liste qui contient tous les éléments d’un arbre sauf a (utilisé pour la fonction delete)
* from\_list (l : list nat): tree nat retourne un arbre contenant tous les éléments de la liste l
* count (T: tree nat): nat retourne le nombre d’éléments contenu dans un arbre
* equals\_value (t1 : tree nat)(t2 : tree nat): bool retourne vrai si les deux arbres contiennent les mêmes éléments
* hauteurMax(t:tree nat ) : nat retourne la hauteur max d’un arbre
* hauteurMin(t:tree nat ) : nat retourne la hauteur min d’un arbre
* is\_balanced\_height(t:tree nat): bool retourne vrai si un arbre est équilibré (si la hauteur max et la hauteur min sont égales) faux sinon.
* ordered(t: tree nat): bool retourne vrai si les éléments de l’arbre sont bien ordonnés
  1. Preuves

Les lemmes que nous pouvons démontrer à partir de cette spécification vont correspondre aux différentes propriétés des arbres 2-3-4, comme la hauteur équilibrée et l’ordonnancement des éléments, et vont aussi correspondre aux « post  condition » des fonctions. Par exemple que si on ajoute un élément e dans un arbre T, e existe dans T après l’opération.

* + 1. Propriétés

La principale propriété des arbres 2-3-4 réside dans l’équilibre et l’ordonnancement des éléments.

* Un arbre reste équilibré par sa construction, lorsqu’on ajoute un élément :

forall T: tree nat ,forall X: nat, (is\_balanced\_height(T)=true) -> (is\_balanced\_height(add X T)=true).

* Un arbre reste bien ordonné par sa construction forall T : tree nat , forall X : nat, (ordered (T) = true => (ordered(add X T) = true.

Pour démontrer ces propriétés difficiles il faut démontrer de plus petits lemmes :

* Propriété arithmétique des hauteurs, le père d’un nœud dont la hauteur vaut h a une hauteur qui vaut h+1. Nous avons décomposé les preuves en fonctions du type de nœuds

forall T2 T1 T3:tree nat,forall n:nat,forall x: nat,(T1 = binode x T2 T3) /\ (n = (max (hauteurMax T2) (hauteurMax T3)) -> (hauteurMax T1) = (S n).

* Propriété de la structure, si un Arbre est équilibré alors ses sous arbres sont équilibrés

forall T2 T1 T3:tree nat,forall x: nat,(T1 = binode x T2 T3) /\ is\_balanced\_height(T1)=true -> is\_balanced\_height(T2)=true.

* (OTHERS)
  + 1. Post Conditions des opérations structurelles
* Opération add : forall T: tree nat ,forall X: nat, exist X (add X T) = true.
* Opération delete : forall T : tree nat, forall X : nat, exist X (delete X T) = false
  + 1. Difficultés rencontrées

Concernant la partie preuve du projet, nous n'avons pas réussi a prouver les lemmes des propriétés. Nous nous sommes rendu comptes de la difficulté de la preuve et nous avons donc pensé prouvé des lemmes plus faciles qui nous serviraient d'intermédiaire pour les plus gros lemmes. Cependant, par manque de temps et à cause de la difficulté nous n'avons pas réussi a atteindre ce but. La difficulté résidait toujours au même phase de la preuve, la structure étant complexe, les opérations nécessitent de nombreux « match » pour gérer tous les cas de nœud, d'ordre etc...

* + 1. Conclusion

En conclusio, ce projet nous aura permi d'explorer les possibilités du langage Coq dans le domaine de l'algorithmique sur les structures complexes tels que les arbres 2-3-4 ainsi que les preuves liées à leur propriétés dont la difficulté implique de passer par de lemmes intermédiaires.