Cours Programmation avancée

Modalités

13 séances : 5 cours, 6 TD/TPs, 1 TP noté, 1 examen

Objectifs

Manipulation d'arbres

Complexité

Structures de données avancées (tables de hash, ABR, ...)

Modularité

Structures d'arbres

Données très utilisées en Informatique

HTML, XML : les *feuilles* sont du texte, les *noeuds* sont les balises. La *racine* est la balise `HTML`.

La syntaxe abstraite d'un langage de programmation. Le texte source d'un programme est transformé en un arbre de syntaxe abstraite (AST).

Permettent de stocker les données de manière compacte. La *hauteur* d'un arbre est la plus grande longueur d'un chemin de la racine à une feuille.

Dans un système de fichiers, les points sont des dossiers ou des fichiers. Les points d'un arbre sont des *noeuds* qui ont des *fils* ou des feuilles. Tout point d'un arbre qui n'est pas la racine possède un *père*.

un *arbre binaire* de hauteur h permet de stocker dans ses *etiquettes* et ses feuilles $2^h - 1$ valeurs.

Codage, représentation

Types de données algébriques

Voir la fin du cours de programmation impérative.

Codage en tableau

Les arbres binaires peuvent se coder en un tableau T : à la hauteur i on place les valeurs $T[2^{i-1} - 1]$ à $T[2^i - 1]$.
Accès en temps constant au fils gauche $n \to 2n$, droit $n \to 2n + 1$, au père $n \to n/2$ et d'accèder aux étiquettes $n \to T[n]$ (modèle de tableaux mémoire).
Toutefois si l'arbre est de hauteur 512 il contient $2^{512} - 1$ points qui de l'ordre de 10^{150} et on peut stocker toutes les particules de l'Univers dans un arbre de hauteur raisonable.

Énumerations

Une énumération des valeurs d'un type τ est une bijection entre les valeurs de τ et $\mathbb{N}$. Tous les types algébriques admettent une énumération. Ils ont une quantité dénombrable de valeurs.

Complexité

Estimation du temps de calcul d'un programme

Classes de complexité, algorithme linéaire, quadratique, exponentiel... Certains algorithmes n'ont pas de complexité : fonction d'Ackerman.
Complexité en temps, en mémoire, dans le pire des cas, en moyenne, borne inférieure de complexité.

Les notation $\mathcal{O}$ et o

f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}+$.
On dit que $f = \mathcal{O}(g)$ si la limte de f/g est une constante non nulle. Relation réflexive, symétrique et transitive.
On dit que $f = o(g)$ si la limite de f/g est nulle. Relation irréflexive et transitive (un préordre)

Classes de problèmes formalisables

La taille N d'un problème $\mathcal{P}$ est la quantité d'information nécessaire à l'énoncer dans un langage formel.

Décidable, non décidable : on connait ou on sait qu'il n'existe pas d'algorithme pour résoudre $\mathcal{P}$.

Polynomial : On connait un algorithme en $\mathcal{O}(N^k)$ pour résoudre $\mathcal{P}$

NP : On connait un algorithme polynomial pour vérifier la solution de $\mathcal{P}$

NP Complet : On connait un algorithme exponentiel (énumération des 2^N possibilités) pour résoudre $\mathcal{P}$ mais pas d'algorithme polynomial.

Le quick sort

Le tri rapide d'un tableau T de taille N consiste à choisir un pivot $p = T[i]$ et à placer p à sa place j dans le tableau de façon à ce que tous les éléments d'indice inférieur à j soient plus petits que p et ceux d'indice plus grand que j soient plus grands que p. le coût pour placer p à sa place est $\mathcal{O}(N)$ soit N et on doit ensuite récursivement trier les deux sous tableaux à gauche et à droite. Soit S_N le temps de calcul de la méthode. On a

$$S_N = N + S_{N/2} + S_{N/2}$$
$$T_k = S(2^N)$$
$$T_{k+1} = 2^k + 2T_k$$

Suite récurrente linéaire non homogène. La partie homogène vérifie

$$U_{k+1} = 2U_k$$
$$U_k = 2^k$$

et $V_k = \frac{U_k}{2^k}$ vérifie

$$V_{k+1} = 1/2 + V_k V_k = N/2$$

et la solution est $T_k = k2^{k-1}$, on revient à $N = 2^k$ et $S_N = \mathcal{O}N\log_2(N)$.
La méthode est optimale car il y a $N!$ ordres possibles.

Des expressions arithmétiques

```
type operateur = Plus | Moins | Mult | Div | Rem;;
type arith =
  | Feuille of int
  | Noeud of operateur * arith * arith
;; (* le sens des operateurs *)
let get_op op = function (n, m) -> match op with
  | Plus -> n + m
  | Moins -> n - m
  | Mult -> n * m
  | Div -> n / m
  | Rem -> n mod m
;; (* le sens d'une expression *)
let eval( exp) = match exp with
  | Feuille f -> f
  | Noeud(op, og, od) -> get_op(o)(eval(og, eval(od))
;;
```