compte rendu du TPL 1 d'Algo 2

Joffrey Bion, Polisano Kévin

25 mars 2011

1 Implémentation

Nous serons relativement bref sur l'explication du code au sein du compte-rendu, car en plus des commentaires ajoutés dans les fichiers `*.adb`, nous avons pris soin de créer un code qui soit le plus clair et concis possible, en privilégiant l'approche récursive. De ce fait nous nous attarderons plus particulièrement sur l'analyse des coûts, que nous validerons ensuite par des tests.

1.1 Paquetage Dictionnaire

Les procédures `Hauteur`, `AjusteHauteur` et `Parcourir` étaient données dans le sujet, les procédures `Vider`, `Cherche` et `EstPresente` ne présentaient aucune difficulté majeure, quant aux procédures `Equilibrer` et `Rotation` le sens gauche était donné dans le sujet ainsi nous avons symétriquement pris en compte l'autre cas suivant le paramètre `Sens`. Nous avons légèrement adapté les procédures `Verif` et `Inserer` de façon a prendre en compte l'équilibrage.

La procédure `ParcourirInterv` a un comportement similaire à `Parcourir` sauf que les appels récursifs et l'appel à `Traiter` dépendent de comment est située la clef par rapport à l'intervalle $[CI, CS]$. En effet tant que la clef est strictement supérieure à CI on peut « descendre » dans le fils gauche, puis on traite la clef (à condition qu'elle soit dans l'intervalle $[CI, CS]$) et on descend de même dans le fils droit tant que la clef est strictement inférieure à CS. Ainsi de suite en sortant des appels récursifs on se déplace dans l'arbre en infixé en traitant les clefs étant dans $[CI, CS]$.

Les procédures délicates à implémenter furent celles de suppression d'un élément dans le dictionnaire. Là encore nous avons opté pour la récursivité qui s'avère être plus concise et optimale que la méthode itérative que nous avions employé au prime abord. La procédure `SupprimeMin` est naturelle, on se déplace dans les fils gauche successifs tant que ceux-ci ne sont pas vides, sinon (c'est-à-dire une fois arrivé sur le noeud le plus à gauche dans l'arbre, soit le minimum) on supprime cette clef minimum en pointant sur le fils droit et en libérant le pointeur sur la clef minimum.

La procédure `Supprime` a globalement la même structure récursive que la procédure `Cherche`, puisque pour supprimer une clef il faut bien sûr la localiser dans l'arbre. La difficulté est de savoir : que faire de ses sous-arbres ? Si le fils droit est vide c'est simple on remplace l'arbre par son fils gauche, ce qui conserve l'ordre sur les clefs et donc la structure d'ABR, sinon on va remplacer l'élément supprimé par le minimum du fils droit (en faisant appel à la procédure `SupprimeMin`), de cette façon toutes les clefs du nouveau fils droit sont supérieures à ce noeud, et toutes celles du fils gauche y sont inférieures (puisque ce noeud était précédemment dans le fils droit), donc la structure d'ABR est de nouveau conservée.

1.2 Paquetage Mémoïze

On instancie ici le paquetage Dictionnaire avec pour unique clef et donnée des entiers naturels qui correspondent au couple $(N, F(N))$. On appelle Graphe cette nouvelle instance.

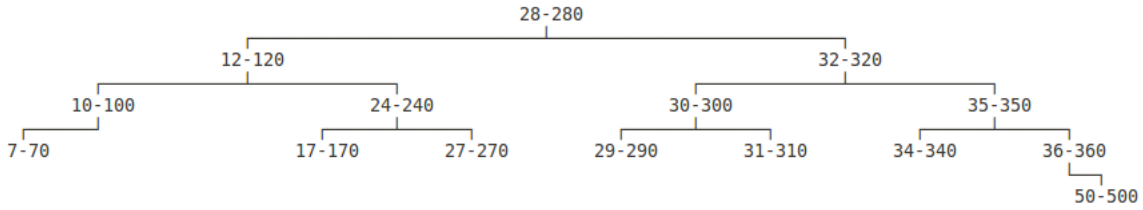
La fonction `Eval` commence par chercher dans le graphe si $F(N)$ a précédemment été calculé (c'est-à-dire si la clef N et la donnée $F(N)$ associée y sont présents), si c'est le cas elle retourne la donnée $F(N)$, sinon elle calcule $F(N)$ et l'insère dans le graphe.

Les procédures de parcours sont instanciées par la procédure auxiliaire `AfficherCouple`.
Ainsi `GrapheAppels` et `GrapheAppels(A,B:Natural)` font simplement appel à `ParcourirGraphe` qui par surcharge désigne soit l'une soit l'autre procédure de parcours.

2 Tests

2.1 Tests du dictionnaire

Afin de pouvoir tester facilement les différentes procédures du paquetage Dictionnaire nous avons réfléchi à une manière de représenter visuellement l'arbre dans la console, et avons ainsi implémenté dans le fichier `Dictionnaire-affichierarbre.adb` plusieurs procédures permettant l'affichage de l'arbre proche de la façon dont on dessine les arbres sur papier. Voici le rendu de notre programme :



Nous avons ensuite effectué les tests dans le programme `testarbre.adb`, en particulier nous avons vérifié le bon fonctionnement des procédures d'insertion (avec et sans équilibrage) et surtout de suppression d'éléments dans l'arbre, qui n'est pas utilisée dans les programmes demandés. Ces derniers, `codespostaux.adb`, `fibonacci.adb`, `fybracuse.adb` et `madfunc.adb` ont le comportement attendu, à savoir :

⋆ Le programme `codespostaux.adb` construit simultanément deux arbres `VC` et `CV` qui ont respectivement pour clef/données une ville et ses code postaux, et un code postal et des villes ayant ce code postal. Lors de l'exécution suivant que l'on tape 1 et un préfixe de ville ou 2 et un préfixe de code postal, le programme affiche toutes les clefs commençant par ce préfixe lors du parcours de l'arbre correspondant.

⋆ Pour chacune de ces fonctions récursives on affiche les 10 001 premières valeurs en consultant le graphe des appels comme expliqué dans la partie Mémoïzation.

2.2 Analyse théorique des coûts

Dans cette sous-partie nous étudions la complexité en moyenne et dans le pire cas des procédures du Dictionnaire suivant la valeur du boolean `EstAVL`.

★ Sans équilibrage (ABR)

D'une manière générale la recherche et l'insertion dans un ABR se font en $O(log_2 n)$ en moyenne, car c'est équivalent à une recherche dichotomique dans un tableau de taille n. De même pour la suppression. En revanche dans le pire des cas - c'est-à-dire quand l'arbre est peigne - ces procédures nécessitent de parcourir la liste donc on a une complexité en $O(n)$ cette fois. C'est le cas pour le graphe des appels des 3 fonctions récursives puisque les clefs sont 1, 2, ..., n ; et c'est pratiquement le cas aussi pour l'arbre `CV` puisque on le construit à partir d'un fichier texte où les clefs (codes postaux) sont presque triées.

Supposons pour fixer les idées que les clefs soient complètement triées (par ordre croissant ici), ainsi l'arbre est totalement déséquilibré à droite. Pour insérer le code postal figurant à la ligne i du fichier texte il faut donc au préalable parcourir le peigne de hauteur $i-1$ donc effectuer $i-1$ comparaisons, puis l'insertion $+1$, donc le coût total de construction de l'arbre est (n étant le nombre de lignes du fichier texte) :

$$C(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \sim \frac{n^2}{2}$$

★ Avec équilibrage (AVL)

L'avantage des AVL est que même dans le pire des cas la complexité des procédures sus-citées est un $O(log_2 n)$. En effet le fait que ce soit un ABR dont la hauteur de chaque fils droit/gauche diffère d'au plus un, garanti que la hauteur de l'arbre est un $O(log_2 n)$. Démontrons ce résultat :

$$log_2(1+n) \overset{1}{\leqslant} 1 + h \overset{2}{\leqslant} 2\ 1.44 log_2(2+n)$$

$\boxed{\overset{1}{\leqslant}}$ Cette inégalité est vraie pour tout arbre binaire de hauteur h, qui a au plus $2^{h+1}-1$ sommets.

$\boxed{\overset{2}{\leqslant}}$ On considère $N(h)$ le nombre minimal de noeuds d'un AVL de hauteur h. $N(0) = 1$, $N(1) = 2$

$$\forall h \geqslant 2, \quad N(h) = 1 + N(h-1) + N(h-2)$$

En posant $F(h) = 1 + N(h)$ on se ramène à une suite de Fibonacci d'où $F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi^{-(h+3)})$ et par suite :

$$1 + n \geqslant F(h) > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1) \Rightarrow h + 3 < log_\phi(\sqrt{5}(2+n)) < \frac{log_2(2+n)}{log_2 \phi} + 2$$

Et comme le nombre de comparaisons effectuées correspond au nombre de noeuds explorés suivant un chemin, il est borné par la hauteur de l'arbre, la complexité des procédures est bien en $O(log_2 n)$ dans tous les cas (ceci aussi parce que l'équilibrage s'effectue à coût constant).

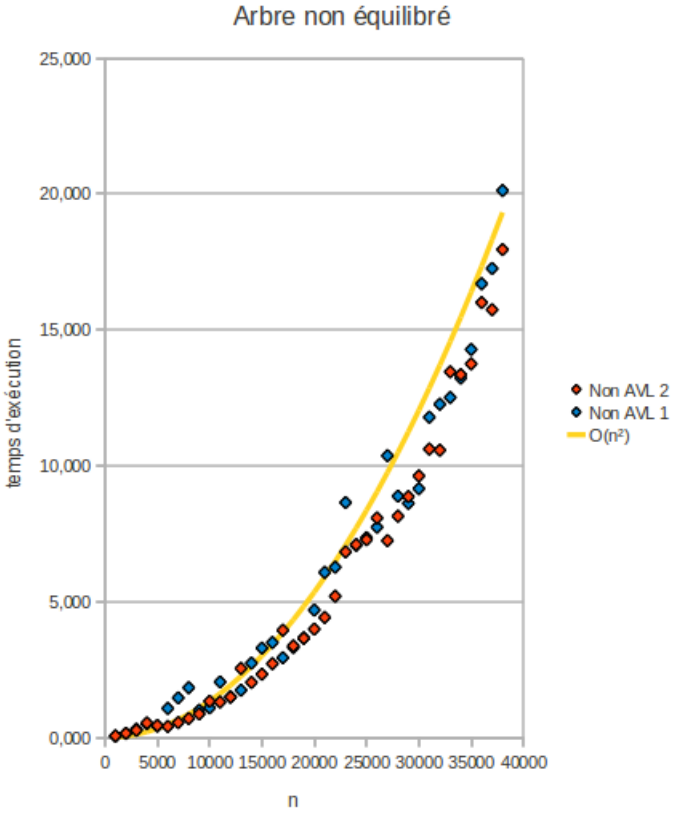
On s'attend alors à ce que l'initialisation du dictionnaire en version équilibrée s'effectue en :

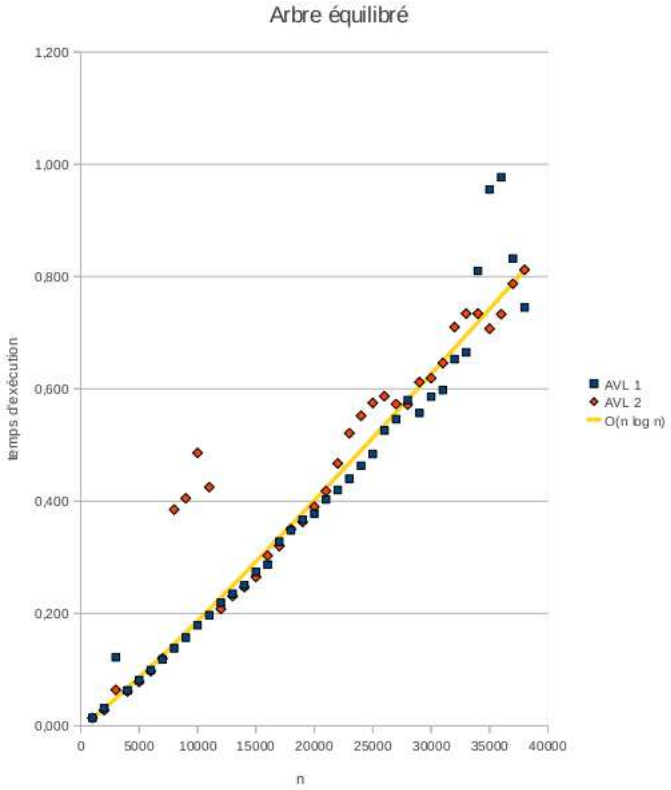
$$\sum_{i=1}^{n} log_2(i) = log_2(i!) \sim log_2\left[\left(\frac{n}{e}\right)^n \sqrt{2\pi n}\right] \sim n log_2(n)$$

2.3 Tests de performance

★ Initialisation du dictionnaire

Nous avons échantilloné le fichier `codes_postaux.txt` en plusieurs fichiers `codes-N.txt` où N varie de 1000 à 38000 (nombre de codes dans le fichier), puis nous avons créé un script shell `temps.sh` qui calcule les temps d'initialisation des arbres (dictionnaires). Nous avons enfin traité les données dans un tableur en réalisant deux fois l'expérience pour chaque cas (d'où les courbes 1 et 2 de chaque graphe) pour vérifier que la tendance est toujours la même, et voici les deux graphiques correspondants, obtenus après homogénéisation du coefficient directeur de la courbe théorique :





Les irrégularités proviennent du fait que nous avons effectué ces tests sur telesun, par conséquent le trafic a perturbé les résultats, mais globalement on observe les croissances annoncées dans la partie théorique.

★ Mémoïzation

Premier constat : en utilisant la forme récursive brute des fonctions `fybracuse.adb` et `madfunc.adb` le temps de calcul des 10 001 premières valeurs est démesuré.

Le principe de Mémoïzation prend alors tout son intérêt. Nous récapitulons dans le tableau suivant les temps d'exécution obtenus pour le calcul des 10 001 valeurs de ces fonctions :

	ARBRE NON ÉQUILIBRÉ	ARBRE ÉQUILIBRÉ
`fybracuse.adb`	0 s 890 ms	0 s 570 ms
`madfunc.adb`	1 m 13 s 335 ms	0 s 719 ms

Les avantages de l'équilibrage apparait alors évident sur l'exemple de la fonction `madfunc.adb` qui possède un comportement chaotique et ainsi un graphe d'appels de très grande taille.