Rapport Projet IPF : String Builders

Teddy Alexandre

Mai 2022

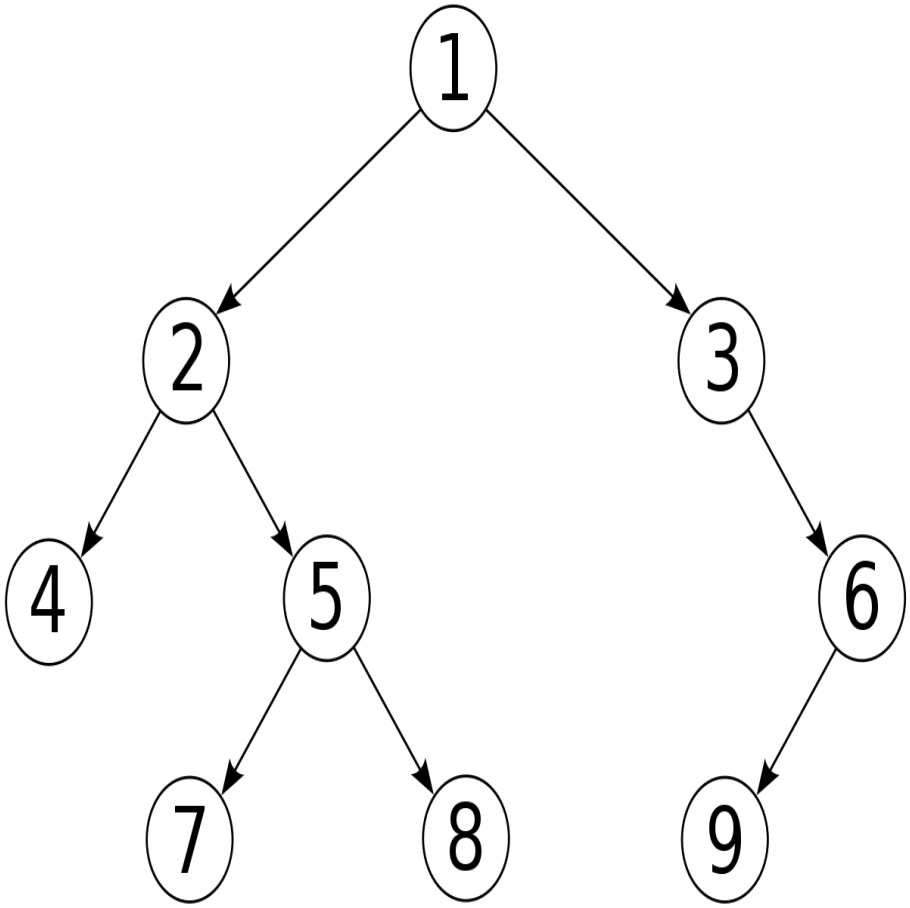

Table des matières

1 Introduction

Le projet consiste en l'optimisation des opérations classiques effectuées sur les chaînes de caractères, notamment la concaténation et l'extraction d'une sous-chaîne. En effet, celles-ci tendent à devenir inefficaces lorsque leur taille grandit. Ainsi, le projet présente une autre approche, fondée sur l'utilisation notamment des arbres binaires. Ceci donne ce que l'on appellera plus tard un ***string_builder***.

2 Description des types utilisés

Le principal type utilisé lors du projet est celui de ***string_builder*** : c'est un **arbre binaire** (donc défini récursivement), avec soit des feuilles (munies d'une chaîne de caractères (*string* en OCaml) et de la longueur associée (type *int*)), soit des noeuds internes ayant pour fils gauche et droit des ***string_builder***. La longueur d'un nœud est donc définie comme la somme des deux longueurs des fils de ce nœud.

3 Description des choix effectués dans le projet

Les premières questions servent à définir et manipuler les ***string_builder*** avec des opérations définies dessus (construction d'un ***string_builder*** à partir d'une chaîne de caractères, concaténation de deux ***string_builder***, extraction du i^e caractère et extraction d'un ***string_builder*** à partir d'un autre). La partie principale du projet concerne le reste des questions, où l'on souhaite évaluer le gain réalisé en termes de coût (défini dans l'énoncé) après avoir appliqué un algorithme d'équilibrage sur un ***string_builder*** généré aléatoirement.

Tout d'abord, je vais détailler la façon dont j'ai procédé pour générer un arbre de profondeur i aléatoirement (question 5). Le principe de l'algorithme est le suivant :

- A l'aide de deux booléens prenant aléatoirement les valeurs true et false (avec *Random.bool ()*), construire la structure de l'arbre aléatoire :
  - si les deux valent faux, recommencer ;
  - si l'un est vrai et l'autre faux, descendre (ie faire $i \leftarrow i - 1$) dans l'arbre du côté où true a été renvoyé, et mettre i à 0 de l'autre côté ;
  - si les deux sont vrais, descendre dans les deux côtés ;
- Lorsque i vaut 0, créer une chaîne de caractères choisis aléatoirement (*Char.chr i* renvoie le caractère dont le code ASCII est i, *Char.escaped c* transforme le *Char c* en *String*, et est l'opérateur de concaténation, de longueur comprise entre 1 et 8 (inclus). Ce choix de longueurs a été fait afin de ne pas avoir d'arbre trop "coûteux".

- Une fois ceci fait, recalculer les longueurs des noeuds internes avec un parcours de l'arbre.

Concernant l'algorithme d'équilibrage (question 7), je me suis servi de pas mal de fonctions auxiliaires.

La première était de transformer une liste de string en une liste de feuilles de ces string afin d'y appliquer l'algorithme présenté. Cela s'effectue avec la fonction *map*.

J'ai ensuite défini 3 fonctions à appliquer sur cette liste, de suppression (de deux éléments consécutifs), insertion (d'un seul élément) et de recherche d'indice (par rapport à deux éléments consécutifs) utilisées pour implémenter l'algorithme.

J'ai enfin défini une nouvelle fonction qui parcourt la liste de feuilles afin de déterminer les deux éléments consécutifs qui minimisent le coût défini dans l'énoncé.

Toutes ces fonctions sont utilisées dans *balance*, qui se contente d'appliquer l'algorithme et de renvoyer un arbre équilibré par rapport à celui de départ.

Pour ensuite effectuer la comparaison demandée (question 8), j'ai choisi de renvoyer un 4-uplet, composé du gain min, du gain max, du gain moyen (type *float*) et du gain médian (*float* également). On génère donc un grand nombre d'arbres aléatoires, on les "équilibre" le plus possible, et on obtient une liste d'entiers, généralement positifs, représentant le gain en coût par rapport à l'arbre de départ. On travaille donc sur cette liste en définissant des fonctions qui calculent le minimum/maximum d'une liste d'entiers. La moyenne, le minimum et le maximum d'une liste d'entiers se calculent via d'autres fonctions utilisant *fold_left*.

Le calcul de la médiane requiert que la liste d'entiers soit triée, ce qui n'est pas le cas. On utilise un algorithme de tri classique et relativement efficace, le **tri fusion**, en $O(n\,log(n))$. Une dernière fonction se charge alors que trier cette liste avant d'effectivement calculer la médiane.

4 Conclusion

On observe qu'après un grand nombre de répétitions en console (entre $N = 1000$ et $N = 10000$ arbres générés raisonnablement), **le gain moyen est significatif** (de l'ordre de plusieurs centaines à quelques milliers), et qu'il est parfois remarquable (des gains maximum peuvent approcher ou atteindre les 15000) ce qui justifie l'efficacité de l'algorithme d'équilibrage. Il existe des cas de perte (gains minimum négatifs) ce qui reste rare et avec des valeurs très faibles en valeur absolue (quelques unités), et peut s'expliquer par des arbres (quasiment) déjà équilibrés. Mais de manière générale, le gain en coût est grandement appréciable.