ENSIIE

Promotion 2024

Rapport Projet IPF Semestre 2

Valentin Buchon

Année 2021-2022

Table des matières

I Introduction **3**
1 Présentation du sujet 3

II Conceptions du projet **4**
1 Échauffement 4
2 Équilibrage 4

Voici le rapport du projet de semestre 2 dans l'UE IPF par Valentin Buchon

Partie I

Introduction

Sommaire

1 Présentation du sujet 3

a) Objectif 3

b) Définition 3

c) Exemple 3

1 Présentation du sujet

a) Objectif

Le **sujet** du projet a pour objectif d'implémenter une structure de `string_builder` permettant une optimisation de l'opération de concaténation et de l'extraction de sous chaîne.

b) Définition

Un `string_builder` est un arbre binaire, dont les feuilles sont des chaînes de caractères usuelles et dont les noeuds internes représentent des concaténations.

c) Exemple

Ainsi le `string_builder` suivant représente le mot `GATTACA`, obtenu par concaténation de quatre mots `G`, `ATT`, `A` et `CA`. L'intérêt des `string_builder` est d'offrir une concaténation immédiate et un partage possible de caractères entre plusieurs chaînes, au prix d'un accès aux caractères un peu plus coûteux.

Partie II

Conceptions du projet

Sommaire

1 Échauffement **4**
2 Équilibrage **4**

1 Échauffement

Question 1

La structure à implémenter est simplement celle d'un arbre binaire avec un `int` stocker sur chaque noeud et un tuple `string*int` sur chaque feuille. Ainsi les implémentations des fonctions `word` et `concat` sont naturelles.

Question 2

Concernant la fonction `chat_at` nous allons parcourir l'arbre de la racine jusqu'à la bonne feuille en utilisant les `int` dans les nœuds. Par exemple si nous sommes sur un noeud et que nous cherchons le 4ème caractères, posons $i = 4$, et que le `int` sur le fils de gauche est $g = 5$ et celui sur le fils de droite est $d = 6$. Sachant que $i < g$ nous descendons sur l'élément de gauche. Mais si $i > 5$ le caractère recherché est sur la branche de droite donc nous descendons sur l'élément de droite et $i = i - g$ pour adapter la valeur de i à la branche.

Question 3

Pour cette fonction `sub_string`, je l'ai décomposé en 2 fonctions auxiliaires. Une `erase_start` et l'autre `erase_end` qui supprime le début et la fin du mot dans le `string_builder`. Ceci me permet de garder une structure très proche de l'originale. L'idée est de parcourir l'arbre en supprimant tous les branches qui contiennent des caractères à supprimer avec un parcours similaire à `char_at` et modifiant également les `int` dans les nœuds.

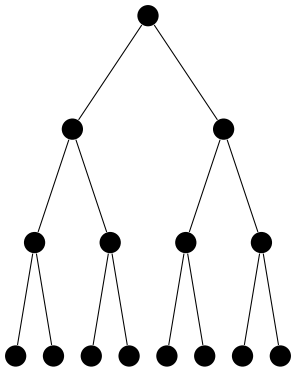
2 Équilibrage

Question 4

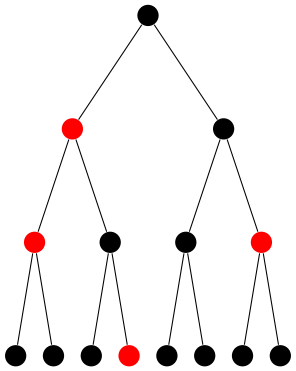
La fonction `cost` parcours simplement l'arbre en faisant les opérations voulues. Par convention nous poserons la hauteur de la racine à 0.

Question 5

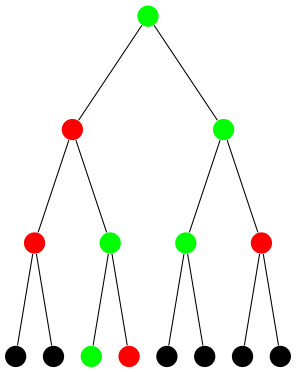
Afin de créer `random_string` j'ai utilisé une méthode de colorisation où la couleur d'un nœud est stocker dans le `int` . Au départ partons d'un arbre complet avec `create_complet` de la hauteur voulu. Ici dans l'exemple il sera de hauteur 3.



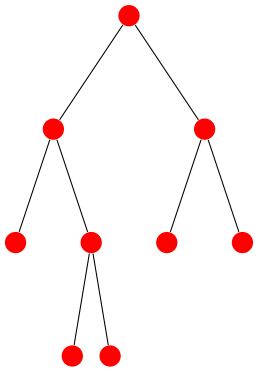
Ensuite nous allons colorer un nombre aléatoire de nœud en des positions aléatoires. Pour gagner un peu en temps `create_complet` colore également les noeuds. Le paramètre `proba` dans la fonction `aux` permet de donner la probabilité à chaque nœud d'être coloré. Celle-ci est arbitraire et peut être modifié afin d'affecter la densité de l'arbre crée. Nous appliquons également `color_base` pour de colorier une feuille aléatoire afin d'être sûr d'avoir la hauteur voulue. Voici ce que nous pouvons obtenir :



A partir de maintenant à l'aide d'une DFS nous allons colorier des nœuds selon une règle particulière. Quand nous sommes sur un noeud, si l'un des deux fils est rouge alors les deux fils et le noeud où je suis deviennent rouges, c'est la fonction `colorisation`. Je mets en vert les nouveaux noeuds pour plus de lisibilité. Nous obtenons donc :



Il nous suffit donc maintenant de supprimer tous les noeuds non colorier avec la fonction `erase_uncolor`.



Nous avons donc maintenant un arbre aléatoire de hauteur voulu. Il n'y a plus qu'à utiliser `gen_random_str` pour remplir les feuilles de `string` aléatoire et utiliser `set_length` afin de donner les bonnes valeurs de concaténations dans les nœuds.

Question 6

Pour cette fonction `list_of_string` nous avons simplement à parcourir les feuilles de l'arbre dans l'ordre de lecteur et d'ajout le mot de chacune dans une liste.

Question 7

`balance` est construit selon l'algorithme donné dans le sujet. La fonction `create_cost_list` renvoie la liste des coûts de la concaténation de chaque élément consécutif de la liste. Ainsi on parcourt cette liste avec `find_min` pour trouver le premier indice de la paire à fusionner avec `fusion`. En répétant cela, nous obtenons le résultat attendu.

Question 8

`gain_balance` est une fonction qui nous permet de faire des statistiques et de comparer les coûts entre une liste de `string_builder` équilibré ou non. En particulier elle renvoie un tableau contenant la différence des moyennes, des maximums, des minimums et les médianes des deux différentes listes. Différentes fonctions ont été implémenter afin de faire ces calculs. Voici des résultats obtenu pour différentes valeurs :
Avec Taille 100 Hauteur 10

- Différence de moyenne sans balance - avec balance : 189
- Différence de maximum : 337
- Différence de minimum : -6
- Différence des médianes : 195

Avec Taille 1000 Hauteur 10

- Différence de moyenne sans balance - avec balance : 178
- Différence de maximum : 367
- Différence de minimum : -80
- Différence des médianes : 193

Avec Taille 100 Hauteur 13

- Différence de moyenne sans balance - avec balance : 1219
- Différence de maximum : 2180
- Différence de minimum : -530
- Différence des médianes : 1332

Nous observons des différences assez significatives. La fonction balance est donc nécessaire afin d'optimiser l'utilisation des `string_builder`. Cependant les résultats négatifs pour le minimum est sont très étonnants et peuvent révéler une erreur dans l'implémentation de certaines fonctions.

Typage

- val length_string_builder : 'a string_builder -> int = <fun>
- val height_stb : 'a string_builder -> int = <fun>
- val word : string -> string string_builder = <fun>
- val concat : 'a string_builder -> 'a string_builder -> 'a string_builder = <fun>
- val char_at : string string_builder -> int -> char = <fun>
- val erase_end : string string_builder -> int -> string string_builder = <fun>
- val erase_start : string string_builder -> int -> string string_builder = <fun>
- val sub_string : string string_builder -> int -> int -> string string_builder = <fun>
- val cost : 'a string_builder -> int = <fun>
- val gen_random_str : int -> string = <fun>
- val erase_uncolor : 'a string_builder -> string string_builder = <fun>
- val create_complet : int -> string string_builder = <fun>
- val color_base : 'a string_builder -> 'a string_builder = <fun>
- val change_c : 'a string_builder -> 'a string_builder = <fun>
- val get_color : 'a string_builder -> int = <fun>
- val colorisation : 'a string_builder -> 'a string_builder = <fun>
- val set_length : 'a string_builder -> 'a string_builder = <fun>
- val random_string : int -> string string_builder = <fun>
- val list_of_string : 'a string_builder -> 'a list = <fun>
- val create_cost_list : 'a string_builder list -> int list = <fun>
- val find_min : int list -> int = <fun>
- val fusion : 'a string_builder list -> int -> 'a string_builder list = <fun>
- val list_of_Feuille : 'a string_builder -> 'a string_builder list = <fun>
- val balance : 'a string_builder -> 'a string_builder = <fun>
- val somme : int list -> int = <fun>
- val maximum : int list -> int = <fun>
- val minimum : int list -> int = <fun>
- val creer_liste_random_stb : int -> int -> string string_builder list = <fun>
- val division : 'a list -> 'a list * 'a list = <fun>
- val fusion : 'a list -> 'a list -> 'a list = <fun>
- val tri_fusion : 'a list -> 'a list = <fun>
- val mediane : int list -> int = <fun>
- val substraction_list : int list -> int list -> int list = <fun>
- val gain_balance : int -> int -> int list = <fun>

Améliorations

Certaines améliorations peuvent être implémentées. En particulier sur la fonction `sub_string` qui peut être factorisée et donc réduire son nombre de lignes. `list_of_feuilles` contient des opérations de concaténations, cela aurait pu être fait sans et donc gagner un peu plus de temps. Dans la fonction `balance`, on peut réduire le nombre d'opérations en évitant de recalculer plusieurs fois le coup de la concaténation de deux éléments. En effet entre 2 itérations, beaucoup d'éléments n'ont pas changé et nous recalculons quand même le coup d'une fusion.