Introduction à la Programmation Fonctionnelle

String Builder

Auteure : Zhao JIN

Table des matières

I. Introduction

La majorité des langages de programmation fournissent une notion primitive de chaîne de caractères. Si ces chaînes s'avèrent adaptées à la manipulation de mots ou de textes relativement courts, elles deviennent généralement inutilisables sur de très grands textes. L'une des raisons de cette inefficacité est la duplication d'un trop grand nombre de caractères lors des opérations de concaténation ou l'extraction d'une sous-chaîne.

Or il existe des domaines où la manipulation efficace de grandes chaînes de caractères est essentielle (représentation du génome en bio-informatique, éditeurs de texte, ...). Ce projet propose une alternative à la notion usuelle de chaîne de caractères que nous appelons **string_builder**. Un **string_builder** est un arbre binaire, dont les feuilles sont des chaînes de caractères usuelles et dont les nœuds internes représentent des concaténations.

II. Réalisation

II.1 La structure de données

La structure de données est définie par la consigne, le type **string_builder** est soit une feuille contenant une chaîne de caractères (je l'appelle **mot** dans ce rapport) en conservant sa longueur, soit un nœud contenant deux **string_builder** (respecter la figure donnée).

II.2 Les fonctions usuelles pour le projet

J'ai d'abord défini les fonctions usuelles pour bien définir les autres fonctions.

- **length** (string_builder -> int = <fun>)

  Cette fonction sert à calculer la somme des longueurs des mots (c'est donc la longueur totale pour un string_builder).

- **nb_noeud** (string_builder -> int = <fun>)

  Cette fonction permet de calculer le nombre total de feuilles.

- **max_depth** (string_builder -> int = <fun>)

  Cette fonction renvoie la profondeur maximum d'un string_builder

II.3 Algorithme de char_at

Pour la réalisation de **chat_at**, j'ai utilisé la fonction **length** qui est définie précédemment. L'exception de "Out of bound" sera déclenchée par la méthode **String.get**.

Algorithme **char_at** :

Entrée : un string_builder **sb**, un indice **i**

Sortie : un caractère

Si **sb(s, l)** est un mot *Alors*

    Renvoie **String.get s i**

Sinon // est un nœud

    l <- la longueur de **string_builder** à gauche

    Si **i** < **l** *Alors*

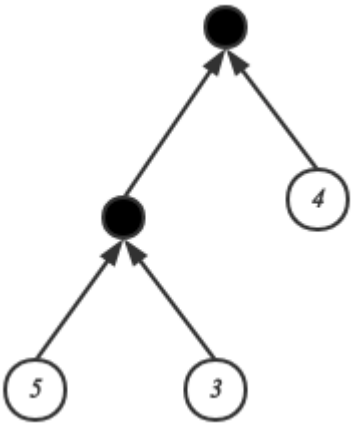
        Renvoie **char_at (string_builder** à gauche**) i**

    Sinon

        Renvoie **char_at (string_builder** à droite**) (i - l)**

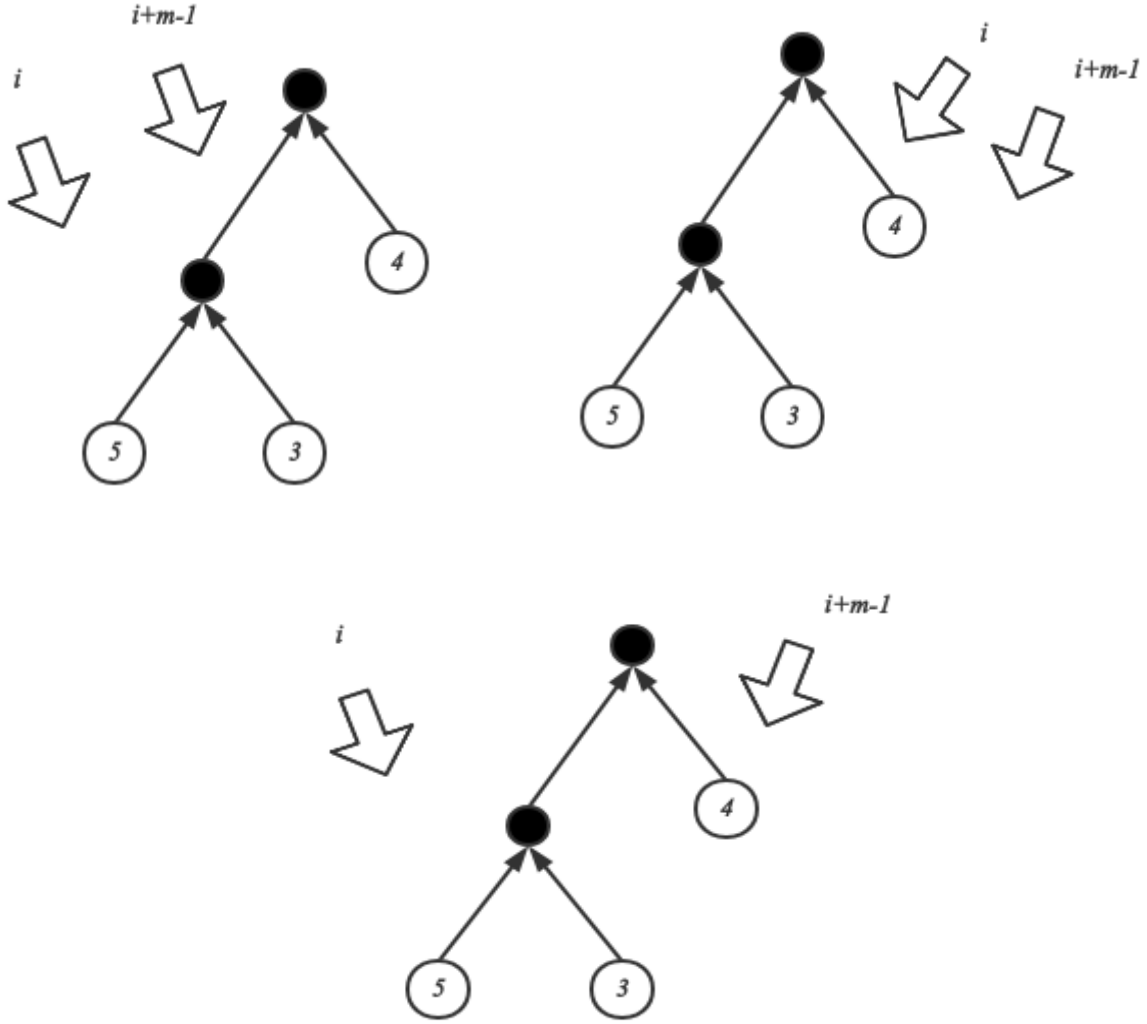
    Fin Si

Fin Si



Voici un exemple de string_builder, avec une longueur de mot donnée. Si nous prenons **i** égal à 5 (l'indice d'OCaml est de 0). Celui plus en haut est un nœud, donc nous calculons d'abord la longueur de string_builder à gauche. Cela fait 8, qui est inférieure à **i**. Ensuite, nous appelons récursivement l'algorithme **char_at** avec les paramètres string_builder à gauche et indice **i**. Maintenant, nous avons un nœud avec deux feuilles. Leur longueur est 5 et 3. Comme c'est un nœud, nous prenons la longueur à gauche, c'est 5, qui n'est pas inférieure strictement à l'indice **i**. Donc, nous parcourons le string_builder à droite avec les paramètres string_builder à droite et indice - longueur(5-5) de string_builder à gauche. Cela veut dire que l'indice de premier caractère de string_builder à gauche est équivalent à la longueur de string_builder à droite. Et puis, nous arrivons dans un mot, donc l'algorithme renvoie String.get de string dans le mot avec l'indice 0 (que nous avons calculé toute à l'heure). Pour les mots à gauche et à droite dans un string_builder. une exception de "index out of bound" sera déclenchée par la méthode **String.get**.

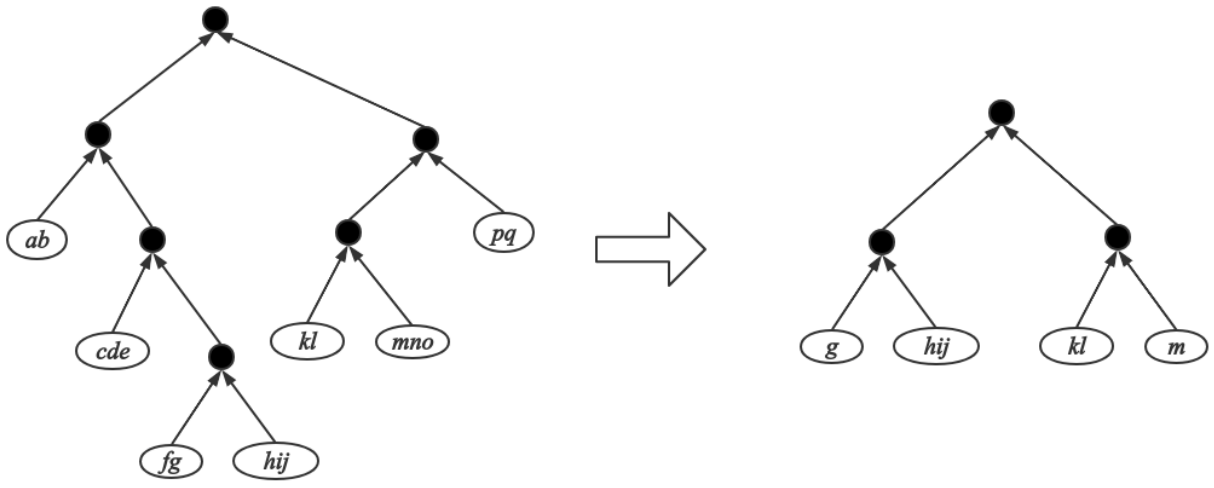
II.4 Algorithme de sub_string



En analysant la question, nous avons trois paramètres, un string_builder sb, un indice **i** et **m**. Le sujet nous demande de renvoyer le sub_string de c_i à c_{i+m-1}, où **m** est la longueur de sub_string. Il y a trois conditions (au-dessus), la troisième condition (au-dessus en bas) peut être décomposée en la condition un (en haut à gauche) et la condition deux (en haut à droite).

Algorithme **sub_string** :
Entrée **:** un string_builder **sb**, un indice **i**, une longueur **m**
Sortie : une concaténation de sub_string en string_builder de string_builder origine
Si i est inférieur à 0 ou (i + m) est supérieur à la longueur totale ou m < 1 *Alors*
    Déclenche une exception
Fin Si
Si **sb(s, l)** est un mot *Alors*
    *// Dans la fonction **String.sub s i m**, m est aussi une longueurs de sub_string*
    Renvoie **String.sub s i m**
Sinon // est un nœud
    l <- la longueur de string_builder à gauche
    Si **i+m <= l** *Alors* *// i+m-1 <= l-1, condition 1*
        Renvoie **sub_string** string_builder à gauche **i m**
    Sinon Si **i>=l** *Alors* *// condition 2*
        Renvoie **sub_string** string_builder à droite **(i - l) m**
    Sinon *// condition 3*
        Renvoie **concat (sub_string** string_builder à gauche **i (l-i))**
                (sub_string string_builder à droite **0 (m-l+i))**
    Fin Si
Fin Si

Après avoir vérifié les paramètres **i** et **m** au début, donc, ce programme ne déclenche pas d'exception au milieu. Cet algorithme permet d'avoir le même arbre que celui en entrée.



II.5 Algorithme random_string

J'ai divisé cet algorithme en trois parties : un algorithme pour générer un string en random, un algorithme pour ajouter un mot dans un string_builder et un dernier algorithme pour générer un random_string (un string_builder).

Algorithme **get_random_str** :
Entrée : la longueur de string **i**
Sortie : un string généré en random
Constante : le maximum et le minimum de code ascii d'un caractère
Si **i** n'est pas positive *Alors*
    Déclenche une exception
Fin Si
Si i est égal à 1 *Alors*
    Renvoie un string d'un seul caractère généré en random
    // le code de ce caractère est entre le code minimum et le code maximum
Sinon
    Renvoie la concaténation (^)
        d'un string d'un seul caractère généré en random
        et **get_random_str (i-1)**
Fin Si

Algorithme **add_node** :
Entrée : un mot à rajouter **m**, un string_builder **sb** qui prend ce mot
Sortie : un string_builder ayant pris le mot
Si **sb** est un mot *Alors*
    b <- un booléen généré par **Random.bool()**
    Si **b** est true *Alors*
        Renvoie la concaténation de **sb** et **m**
    Sinon
        Renvoie la concaténation de **m** et **sb**
    Fin Si
Sinon // un nœud
    b <- un booléen généré par **Random.bool()**
    Si **b** est true *Alors*
        Renvoie **sb** en rajoutant **m** pour le string_builder à gauche de **sb**
    Sinon
        Renvoie **sb** en rajoutant **m** pour le string_builder à droite de **sb**
    Fin Si
Fin Si

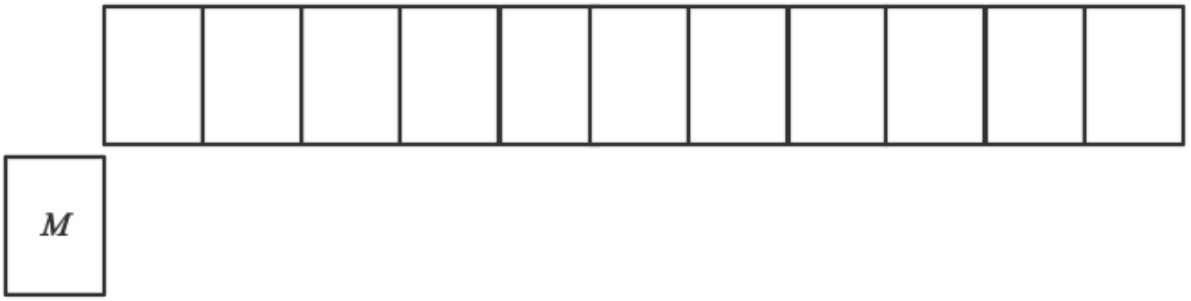
Algorithme **random_string** :
Entrée : la profondeur **i**
Sortie : un string_builder qui a la profondeur maximum de **i**
Constante : la longueur maximum d'un string généré aléatoirement **l**
Si **i** est négatif *Alors*
    Déclenche une exception
Fin Si
sb <- **word** (**get_random_str** ((**Random.int l**)+1))

Tant que la profondeur maximum de **sb** est inférieur strictement à **i** *Faire*
    m <- **word** (**get_random_str** ((**Random.int l**)+1))
    sb <- **add_node m sb**
Fin Tant que
// donc si i=0 alors cette boucle n'exécute pas
// la réalisation est en récursive
Renvoie **sb**

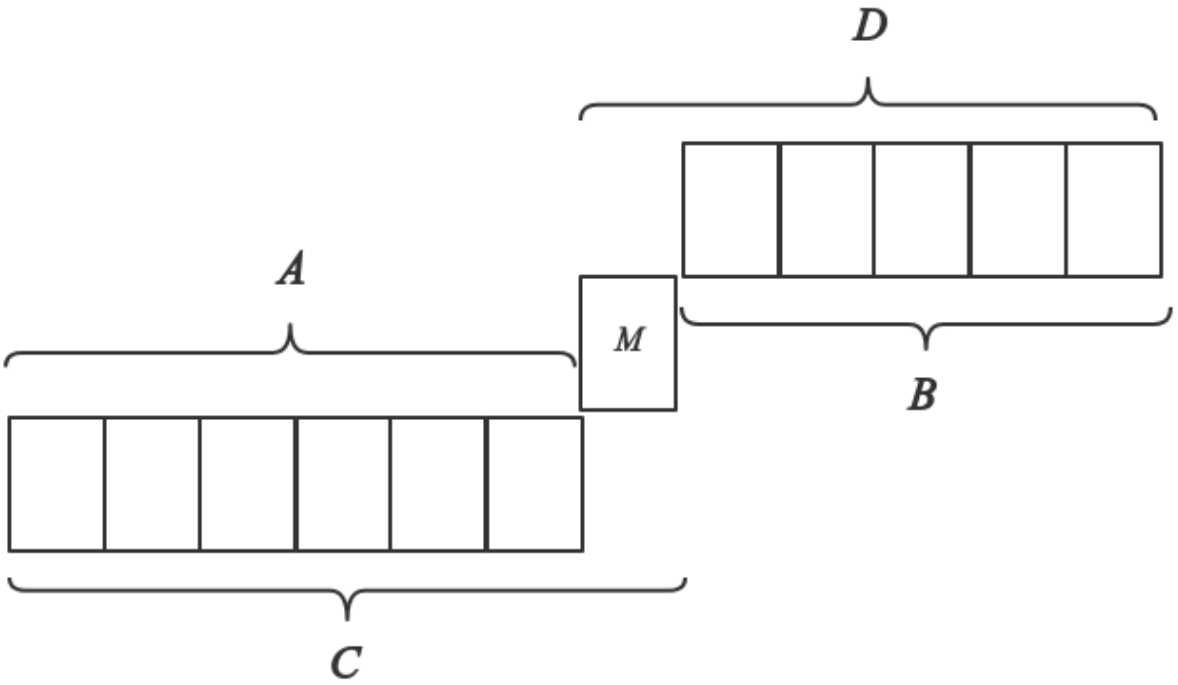
Dans cet algorithme, les choses aléatoires sont : les caractères dans le string généré, la longueur du string généré, et le choix de sous arbre pour mettre le string généré (mot). Donc, un string_builder généré par cet algorithme a un nombre de mots (feuilles) de i+1 à 2^i.

II.6 Algorithme balance

Je considère qu'un string_builder est un "Weighted Binary Tree". Donc pour équilibrer un string_builder, nous devons considérer le poids des feuilles, c'est-à-dire la longueur d'un mot.



Voici un schéma pour expliquer comment découper une liste de string (ou mot). D'abord, nous avons une liste vide (liste à gauche) est une liste de string entière (liste à droite). Nous prenons la première case, et calculons la longueur de string de cette case plus la longueur de string pour la liste à gauche, ainsi que la longueur de la liste à droite en retirant cet élément.



Dans le schéma au-dessus, A est la longueur du string de la liste à gauche, B est celle à droite en retirant l'élément M, C est A plus la longueur de M, D est B plus la longueur de M.

Quand B - C est négatif, cela veut dire qu'on est au milieu, donc il y a deux choix : soit mettre M dans la liste gauche, soit mettre M dans la liste droite. Cette étape assure que D - A et C - B sont positifs ou nuls.

Nous comparons la valeur entre C- B et D - A. Si C - B est inférieur à D - A, nous mettons M à gauche, sinon nous mettons M à droite.

Pour construire un arbre, nous calculons le nombre d'éléments de la liste. S'il n'y a qu'un seul élément, nous renvoyons un mot, s'il y a deux éléments, nous renvoyons une concaténation de deux mots, sinon nous découpons la liste en utilisant l'algorithme précédent (split). Ensuite nous renvoyons la concaténation des deux sous arbres générés par ces deux listes. Ceci est une méthode récursive.

Algorithme **split** :
Entrée : une liste **lst**
Sortie : une paire de liste équilibrés sur la longueur du string
(La longueur d'une liste n'est pas le nombre d'élément d'une liste, c'est la somme de longueur de string)
Tant que la longueur de la liste gauche est inférieur à celle de la liste droite *Faire*
    Déplacer un élément de la liste droite dans la liste droite
Fin Tant que
Retirer le dernier élément de la liste gauche et le marquer comme M
Si (C-B) < (D-A) *Alors*
    Renvoie la liste gauche avec M et la liste droite
Sinon
    Renvoie la liste gauche et la liste droite avec M
Fin Si

Algorithme **treefy** :
Entrée : une liste de mot **lst**
Sortie : un string_builder équilibré
len <- le nombre d'élément de liste **lst**
Si **len** < 1 *Alors*
    Déclenche une exception "list vide"
Sinon Si **len** = 1 *Alors*
    Renvoie le premier élément de **lst** (un mot)
Sinon Si **len** = 2 *Alors*
    Renvoie la concaténation du premier élément et le deuxième élément
Sinon
    Renvoie la concaténation de deux arbres générés par le découpage de **lst**
Fin Si

Algorithme **balance** :
Entrée : un string_builder **sb**
Sortie : un string_builder équilibré
lst <- **list_of_string sb**
Renvoie **treefy (List.map word lst)** *// faut transférer la liste de string en la liste de mot*

II.7 Les gains de coût de la fonction balance

Le coût principal de la fonction balance est la méthode **treefy** et **split**, la méthode **split** parcours la liste avec le coût de **O(N)**, la méthode **treefy** est une méthode récursive. L'espérance du coût est **$O(log_2N)$**, mais dans le pire cas, le coût devient **O(N)**. Donc la méthode **balance** a un coût entre **$O(Nlog_2N)$** et **$O(N^2)$**.

Mais l'équilibrage ne s'effectue qu'une fois pour un string_builder généré. Nous avons **le coût total = le coût de balance + n * le coût d'accès**. **n** est le nombre d'accès à un string_builder. En considérant que le nombre **n** est assez grand, le coût de la fonction balance est négligeable.

Dans la fonction **gains**, j'ai proposé de calculer la différence entre le coût d'accès avant et après l'équilibrage. Normalement, si la fonction **balance** permet de bien équilibrer un string_builder, la valeur **min** ne doit pas être négative.

III. Conclusion

En conclusion, dans ce projet, j'ai réalisé un string_builder en utilisant la structure de donnée "Weight Binary Tree" qui permet de générer et manipuler de grands string. La réalisation de la méthode **balance** peut bien marcher. Mais je pense que cette méthode peut être encore améliorée, le coût peut être plus proche de **O(N)**. D'après les réalisations proposées dans le sujet, il y a encore des fonctions intéressantes à réaliser, par exemple, **contain**, **insert**, **reverse**, etc…

IV. Remerciement

Je remercie d'abord Mme. Stefania DUMBRAVA qui nous a enseigné la programmation fonctionnelle en OCaml. Je remercie aussi Dr. Michael Ryan Clarkson qui publie les cours sur OCaml. Je remercie Lucas DARFEUILLE qui m’aide à améliorer l’écriture de ce rapport au niveau de langue française.

Références

1. Javadoc StringBuilder
   https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/StringBuilder.html
2. Programmation en OCaml & Utilisation d'OUnit2
   https://cs3110.github.io/textbook/cover.html