Rapport Projet
Programmation Fonctionnelle

Adib Habbou

13 Mai 2022

Table des matières

1. Introduction

L'objectif de ce projet est d'implémenter les `string_builder` une alternative à la notion usuelle de chaîne de caractères.

Un `string_builder` est un arbre binaire composé soit de feuilles et de noeuds. Les feuilles sont des chaînes de caractères et leur longueur, tandis que les noeuds sont des concaténations de feuilles avec la longueur correspondante.

Le but de cette implémentation est de pouvoir réaliser des concaténations et un partage de caractères entre différentes chaînes de manière plus efficace en terme de temps et de mémoire.

2. Types et fonctions

2.1 Question 1

2.1.1 Type string_builder

On définit `string_builder` comme étant soit une feuille `Leaf` contenant une chaîne de caractères et sa longueur, soit un noeud `Node` contenant deux `string_builder` ainsi que la longueur de la chaîne de caractères correspondantes.

2.1.2 Fonction word

La fonction `word` qui prend en paramètre une chaîne de caractères retourne la feuille correspondante grâce à `Leaf`. On récupère la longueur de la chaîne de caractères avec `String.length`.

2.1.3 Fonction length

La fonction `length` qui renvoie la longueur de la chaîne de caractères représenter par un `string_builder`. On utilise un pattern matching qui renvoie à chaque fois la longueur correspondant que notre `string_builder` soit une feuille `Leaf` ou un noeud `Node`.

2.1.4 Fonction concat

La fonction `concat` concatène deux `string_builder` en utilisant `Node` avec comme longueur la somme des longueurs des deux `string_builder` en paramètre obtenue grâce à `length` et en mettant nos deux `string_builder` un à gauche et l'autre à droite.

2.2 Question 2

2.2.1 Fonction char_at

La fonction `char_at` renvoie le caractère à la position i, pour ce faire on effectue un pattern matching avec trois cas :

— si le `string_builder` est une feuille `Leaf` on retourne le résultat de la fonction `String.get` sur la chaîne de caractère ;

— si le `string_builder` est un noeud `Node` et que l'indice i est inférieur à la longueur du `string_builder` gauche (c'est-à-dire que le caractère se trouve dans `sb_left`) on retourne `char_at` de `sb_left` pour le même indice i ;

— si le `string_builder` est un noeud `Node` et que l'indice i est supérieur ou égale à la longueur du `string_builder` gauche (c'est-à-dire que le caractère ne se trouve pas dans `sb_left`) on retourne `char_at` de `sb_right` pour l'indice i moins la longueur du `string_builder` gauche (car il faut compter les caractères présents dans `sb_left`).

2.3 Question 3

2.3.1 Fonction sub_string

La fonction `sub_string` qui renvoie une sous-chaîne de caractères qui commence au i-ème caractère et qui a une longueur m. On effectue un pattern matching et on distingue quatres cas :

— si le `string_builder` est une feuille `Leaf` on retourne le résultat de la fonction `String.sub` sur la chaîne de caractères ;

— si le `string_builder` est un noeud `Node`, que l'indice i est inférieur à la longueur du `string_builder` gauche et que la somme de i avec la longueur m est elle aussi inférieur à longueur de `sb_left` (c'est-à-dire que la sous-chaîne de caractères se trouve dans `sb_left`) on retourne `sub_string` de `sb_left` avec comme indice i et comme longueur m ;

— si le `string_builder` est un noeud `Node`, que l'indice i est inférieur à la longueur du `string_builder` gauche et que la somme de i avec la longueur m est supérieure ou égale à longueur de `sb_left` (c'est-à-dire que la sous-chaîne de caractères se trouve à moitié dans `sb_left` et à moitié dans `sb_right`) on retourne la concaténation de `sub_string` de `sb_left` avec comme indice i et comme longueur la longueur de `sb_left` moins i et `sub_string` de `sb_right` avec comme indice 0 et longueur m - i plus la longueur de `sb_left` (pour prendre compléter la sous-chaîne avec la longueur nécessaire en partant du début de la chaîne de `sb_left`) ;

— si le `string_builder` est un noeud `Node` et que l'indice i est supérieur à la longueur du `string_builder` gauche (c'est-à-dire que la sous-chaîne de caractères se trouve dans `sb_right`) on retourne `sub_string` de `sb_right` avec comme indice i moins la longueur de `sb_left` et comme longueur m.

2.4 Question 4

2.4.1 Fonction cost_aux

La fonction `cost_aux` prend comme paramètre un `string_builder` et une profondeur `depth` qui fait un patter matching sur le `string_builder` en distinguant deux cas :

— si le `string_builder` est une feuille `Leaf` on renvoie le produit le la longueur de la chaîne de caractères avec la profondeur `depth` ;

— si le `string_builder` est un noeud `Node` on renvoie la somme de `cost_aux` appliquée à `sb_left` et `sb_right` en incrémentant la profondeur de 1 (de manière à effectuer récursivement des appels à `cost_aux` jusqu'à arrivé à une feuille en ayant bien pris soin de calculer la profondeur au fur et à mesure).

2.4.2 Fonction cost

La fonction `cost` appelle la fonction `cost_aux` avec comme profondeur 0 puisqu'on part de la racine du `string_builder`.

2.5 Question 5

2.5.1 Fonction random_word_aux

La fonction `random_word_aux` prend un entier n en paramètre et renvoie une chaîne de caractères aléatoires de taille n composé uniquement de "A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z".

2.5.2 Fonction random_word

La fonction `random_word` qui appelle la fonction `random_word_aux` avec une taille aléatoire comprise entre 0 et 10.

2.5.3 Fonction random_leaf

La fonction `random_leaf` renvoie une feuille `Leaf` représentant une chaîne de caractères générée aléatoirement par la fonction `random_word`.

2.5.4 Fonction random_string_aux

La fonction `random_string_aux` prend en paramètre un entier i qui représente la profondeur voulu et un entier `depth` qui représente la profondeur actuelle. Une fois que la profondeur actuelle a atteint la profondeur voulu on retourne juste une feuille aléatoire grâce à `random_leaf`. Tant que la profondeur actuelle n'a pas atteint la profondeur voulu, on choisi aléatoirement un nombre entre 0, 1 et 2 qui va donc nous indiquer si l'on insère une feuille aléatoire à cette profondeur ou pas et si l'on insère est-ce qu'on l'insère à gauche ou à droite. On distingue donc trois cas :

- si `choice` est égale à 0 on ne rajoute pas de feuille aléatoire on ne fait que concaténer deux appels récursifs de `random_string_aux` en incrémentant `depth` de 1 ;
- si `choice` est égale à 1 on concatène l'appel récursif de `random_string_aux` en incrémentant `depth` de 1 avec une feuille aléatoire à droite ;
- si `choice` est égale à 2 on concatène l'appel récursif de `random_string_aux` en incrémentant `depth` de 1 avec une feuille aléatoire à gauche ;

2.5.5 Fonction random_string

La fonction `random_string` ne fait qu'appeler la fonction `random_string_aux` avec comme profondeur initiale 0 puisqu'on commence à construire notre arbre aléatoire à partir de rien.

2.6 Question 6

2.6.1 Fonction list_of_string

La fonction `list_of_string` renvoie la liste des chaînes de caractères dans le même ordre que dans le `string_builder` en effectuant un patter matching :

— si le `string_builder` est une feuille `Leaf` on renvoie la liste contenant la chaîne de caractères présentes dans la feuille ;

— si le `string_builder` est un noeud `Node` on appelle récursivement la fonction `list_of_string` sur la suite du `string_builder` c'est-à-dire sur `sb_left` et `sb_right` (afin d'appliquer `list_of_string` jusqu'à arrivé à une feuille).

2.7 Question 7

2.7.1 Fonction init_min

La fonction `init_min` sert à initialiser le min pour la fonction `least_cost` en calculant le coût de la concaténation des deux premiers éléments de la liste `sb_leaf`.

2.7.2 Fonction least_index_cost

La fonction `least_index_cost` renvoie l'indice du premier des deux éléments successifs dont la concaténation a le coût le plus faible. Pour ce faire on utilise un fonction `least_index_cost_aux` qui prend en paramètre une liste un minimum et deux entiers i et j. Elle va ensuite parcourir la liste en vérifiant si le coût de concaténation de deux éléments est plus faible que le `min` : si oui elle met à jour le `min` et l'entier `j` qui stocke la valeur que l'on souhaite retourner.

2.7.3 Fonction concat_least_cost

La fonction `concat_least_cost` concatène les deux éléments successifs à partir de la position indiqué en argument. Elle retourne donc une liste de `string_builder` contenant la concaténation.

2.7.4 Fonction balance

La fonction `balance` applique l'algorithme donnée dans le sujet. On applique d'abord la fonction `list_of_string` puis la fonction `word` pour obtenir une liste des feuilles de notre `string_builder` que l'on appellera `sb_leaf`. On utilise ensuite une fonction auxiliaire `balance_aux` qui utilise la fonction `least_index_cost` pour trouver l'indice de où est ce que doit être effectuer la concaténation, puis elle applique cette concaténation grâce à la fonction `concat_least_cost`. Enfin on applique `balance_aux` à `sb_leaf` puis on retourne le premier élément de la liste obtenu à l'aide de `List.hd`. Cet élément est par conséquent le `string_builder` balancé puisque l'on applique `balance_aux` jusqu'à ce qu'il n'y ait plus qu'un seul élément dans notre liste.

2.8 Question 8

2.9 Fonction random_generator

La fonction `random_generator` nous sert à générer de manière aléatoire une liste de taille entre 1 et 100 composé de `string_builder` de profondeur entre 0 et 5. Pour ce faire, on utilise une fonction auxiliaire `random_generator_aux` qui prend un entier `n` en argument. Elle génère ensuite une liste de taille `n` en utilisant la fonction `random_string` pour générer un `string_builder`.

2.9.1 Fonction list_cost

La fonction `list_cost` renvoie une liste des coûts de chaque `string_builder` dans la liste en appliquant `cost`.

2.9.2 Fonction list_cost_balanced

La fonction `list_cost_balanced` renvoie une liste des coûts de chaque `string_builder` balancé dans la liste en appliquant `cost` après `balance`.

2.9.3 Fonction sum_list

La fonction `sum_list` renvoie la somme des éléments de la liste l'aide de `List.fold_left` en appliquant une fonction qui renvoie la somme de deux entiers.

2.9.4 Fonction min_list

La fonction `min_list` renvoie le minimum d'une liste à l'aide de `List.fold_left` en appliquant une fonction qui renvoie le minimum de deux entiers.

2.9.5 Fonction avg_list

La fonction `avg_list` renvoie la moyenne d'une liste à l'aide de `List.fold_left` en appliquant la fonction `sum_list` puis en divisant par la taille de la liste.

2.9.6 Fonction med_list

La fonction `med_list` renvoie la médiane d'une liste. On vérifie d'abord si la taille `n` de la liste est impaire :

— si oui on renvoie le float correspondant à l'élément d'indice `(n-1)/2` grâce à la fonction `List.nth` ;

— si non on renvoie le float correspondant à la moyenne des éléments d'indice `(n/2)-1` et `n/2`.

2.9.7 Fonction pnl

La fonction `pnl` fonctionne de la manière suivante :

— on génère une liste aléatoire de `string_builder` qu'on appelle `random_sb_list` ;

— on calcule la liste des coûts des `string_builder` avant et après les avoir balancés à l'aide de `list_cost` et `list_cost_balanced` ;

— on applique ensuite les fonctions `max_list`, `min_list`, `avg_list` et `med_list` sur les deux listes précédentes ;

— on applique la fonction `sum_list` sur les deux listes précédentes afin de calculer la différence et pouvoir établir le gain de coût lorsqu'on applique notre algorithme d'équilibrage.

3. Tests

Les résultats retournés sont identiques aux résultats attendus pour l'ensemble des tests. Tous les tests sont présent dans le fichier `string_builder.ml` avec en commentaire le résultat attendu.

Les résultats retournées par la fonction `random_string` et `pnl` sont là à titre d'exemple, leur caractère aléatoire fait que les résultats sont différents à chaque fois qu'on relance le programme.

Tests string_builder

```
let sb_1 = Leaf (10, "OCaml test") ;;
val sb_1 : string_builder = Leaf (10, "OCaml test")
let sb_2 = Node (9, Leaf (5, "OCaml"), Leaf (4, "test")) ;;
val sb_2 : string_builder = Node (9, Leaf (5, "OCaml"), Leaf (4, "test"))
let sb_3 = Node(32, Node (9, Leaf(5, "OCaml"), Leaf (4, "is a")), Node (23, Leaf (16, "n
val sb_3 : string_builder =
  Node (32, Node (9, Leaf (5, "OCaml"), Leaf (4, "is a")),
   Node (23, Leaf (16, "nice programming"), Leaf (7, "langage")))
```

Tests word

```
let word_1 = word "OCaml" ;;
val word_1 : string_builder = Leaf (5, "OCaml")
let word_2 = word "is" ;;
val word_2 : string_builder = Leaf (2, "is")
let word_3 = word "a" ;;
val word_3 : string_builder = Leaf (1, "a")
let word_4 = word "nice" ;;
val word_4 : string_builder = Leaf (4, "nice")
let word_5 = word "programming" ;;
val word_5 : string_builder = Leaf (11, "programming")
let word_6 = word "langage" ;;
val word_6 : string_builder = Leaf (7, "langage")
```

Tests length

```
let len_1 = length sb_1 ;;
val len_1 : int = 10
let len_2 = length sb_2 ;;
val len_2 : int = 9
let len_3 = length sb_3 ;;
val len_3 : int = 32
```

Test concat

```
let sentence = concat (concat word_1 word_2)
                 (concat (concat word_3 word_4) (concat word_5 word_6)) ;;
val sentence : string_builder =
  Node (30, Node (7, Leaf (5, "OCaml"), Leaf (2, "is")),
   Node (23, Node (5, Leaf (1, "a"), Leaf (4, "nice")),
    Node (18, Leaf (11, "programming"), Leaf (7, "langage"))))
```

Tests char_at

```
let char_at_word_0 = char_at word_1 0 ;;
val char_at_word_0 : char = 'O'
```

```
let char_at_word_4 = char_at word_1 4 ;;
val char_at_word_4 : char = 'l'
let char_at_sentence_0 = char_at sentence 0 ;;
val char_at_sentence_0 : char = 'O'
let char_at_sentence_1 = char_at sentence 1 ;;
val char_at_sentence_1 : char = 'C'
let char_at_sentence_10 = char_at sentence 10 ;;
val char_at_sentence_10 : char = 'c'
let char_at_sentence_20 = char_at sentence 20 ;;
val char_at_sentence_20 : char = 'i'
```

Tests sub_string

```
let sub_string_word = sub_string word_5 4 3 ;;
val sub_string_word : string_builder = Leaf (3, "ram")
let sub_string_sentence = sub_string sentence 10 7 ;;
val sub_string_sentence : string_builder =
  Node (7, Leaf (2, "ce"), Leaf (5, "progr"))
```

Tests cost

```
let cost_word = cost word_5 ;;
val cost_word : int = 0
let cost_sentence = cost sentence ;;
val cost_sentence : int = 83
```

Tests random_string

```
let random_string_1 = random_string 1 ;;
val random_string_1 : string_builder =
  Node (6, Leaf (6, "JAGATH"), Leaf (0, ""))
let random_string_5 = random_string 3 ;;
val random_string_5 : string_builder =
  Node (28, Leaf (9, "TYIXLYHLE"),
   Node (19, Node (14, Leaf (5, "BRJNO"), Leaf (9, "GGHGYRJYV")),
    Node (5, Leaf (0, ""), Leaf (5, "GZWFM"))))
let random_string_10 = random_string 5 ;;
val random_string_10 : string_builder =
  Node (31,
   Node (30,
    Node (15,
     Node (8, Node (3, Leaf (3, "VUX"), Leaf (0, "")), Leaf (5, "FGZSO")),
     Leaf (7, "PXRFFBO")),
    Node (15, Leaf (4, "WMW"),
     Node (11, Node (6, Leaf (5, "OZAYM"), Leaf (1, "N")),
      Node (5, Leaf (0, ""), Leaf (5, "IUKHR"))))),
   Leaf (1, "U"))
```

Tests list_of_string

```
let list_word = list_of_string word_5 ;;
val list_word : string list = ["programming"]
let list_sentence = list_of_string sentence ;;
val list_sentence : string list =
  ["OCaml"; "is"; "a"; "nice"; "programming"; "langage"]
```

Tests least_index_cost

```
let l1 = "AAA"::"BB"::"C"::"D"::"EEEEE"::[] ;;
val l1 : string list = ["AAA"; "BB"; "C"; "D"; "EEEEE"]
```

```
let l1' = List.map word l1 ;;
val l1' : string_builder list =
  [Leaf (3, "AAA"); Leaf (2, "BB"); Leaf (1, "C"); Leaf (1, "D");
   Leaf (5, "EEEEE")]
let index_l1 = least_index_cost l1' ;;
val index_l1 : int = 2
let l2 = "AA"::"BBB"::"CCCC"::"DDDD"::"E"::"F"::"GG"::"HHH"::[] ;;
val l2 : string list = ["AA"; "BBB"; "CCCC"; "DDDD"; "E"; "F"; "GG"; "HHH"]
let l2' = List.map word l2 ;;
val l2' : string_builder list =
  [Leaf (2, "AA"); Leaf (3, "BBB"); Leaf (4, "CCCC"); Leaf (4, "DDDD");
   Leaf (1, "E"); Leaf (1, "F"); Leaf (2, "GG"); Leaf (3, "HHH")]
let index_l2 = least_index_cost l2' ;;
val index_l2 : int = 4
```

Tests balance

```
let balanced_word = balance word_5 ;;
val balanced_word : string_builder = Leaf (11, "programming")
let balanced_sentence_1 = balance sentence ;;
val balanced_sentence_1 : string_builder =
  Node (30,
   Node (12, Leaf (5, "OCaml"),
    Node (7, Node (3, Leaf (2, "is"), Leaf (1, "a")), Leaf (4, "nice"))),
   Node (18, Leaf (11, "programming"), Leaf (7, "langage")))
```

Tests pnl

```
let test_1 = pnl () ;;
val test_1 : int * int * float * float * int * int * float * float * int =
  (8619, 6, 827.666666666666629, 380., 6589, 6, 606.372549019607845, 233.,
   11286)
let test_2 = pnl () ;;
val test_2 : int * int * float * float * int * int * float * float * int =
  (7879, 4, 819., 1938.5, 5923, 4, 572.159574468085111, 1205.5, 23203)
let test_3 = pnl () ;;
val test_3 : int * int * float * float * int * int * float * float * int =
  (6007, 6, 755.535211267605632, 4261., 4503, 6, 528.647887323943678, 2881.,
   16109)
let test_4 = pnl () ;;
val test_4 : int * int * float * float * int * int * float * float * int =
  (7184, 12, 1070.94871794871801, 126., 5322, 12, 725.28205128205127, 104.,
   13481)
let test_5 = pnl () ;;
val test_5 : int * int * float * float * int * int * float * float * int =
  (5447, 3, 548.340425531914889, 22., 3757, 3, 398.297872340425556, 12.,
   7052)
```

4. Améliorations possibles

Pour améliorer le code, on peut essayer d'utiliser moins de fonctions auxiliaires en codant certaines fonctionnalités directement dans les fonctions principales. Néanmoins cela aboutirait à un code beaucoup moins lisible et moins structuré même si ce serait sûrement un gain d'efficacité en terme de temps d'exécution.

On pourrait également augmenter la taille des chaînes de caractères générées aléatoirement par la fonction `random_word` et augmenter également la taille des listes qu'on génère avec la fonction `random_generator`, même si l'on obtient des résultat probant lorsqu'on applique la fonction `pnl` avec des chaînes de caractères de taille maximale 10 et des listes de taille maximale 100.

On pourrait aussi améliorer la manière dont on calcule le maximum, le minimum, la moyenne et la médiane. En codant tous dans la même fonction faisant ainsi une seule fois le parcours de la liste lorsqu'on souhaite obtenir les quatres indicateurs.

5. Conclusion

L'implémentation du type `string_builder` ainsi que des différentes fonctions permet de se faire une idée de l'efficacité des arbres équilibrés.

En regardant les valeurs obtenues par la fonction `pnl` sur un large échantillon de d'arbres générés aléatoirement, on se rend compte que la fonction `balance` offre un réel gain en terme de coût pour les `string_builder`.

On remarque que le gain varie entre 25 000 et 1000, ces valeurs dépendent de ce qu'on a pris comme taille arbitraire pour `random_word` et `random_generator`. Dans ce cas précis on génère des listes de taille entre 0 et 100 avec des `string_builder` de profondeur comprise entre 0 et 5.

On peut donc dire que l'algorithme d'équilibrage présenté dans le sujet a un intérêt réel pour réduire de manière considérable le coût d'accès à tous les caractères d'un `string_builder`.

6. Annexe

Profil des fonctions

```
val word : string -> string_builder = <fun>

val length : string_builder -> int = <fun>

val concat : string_builder -> string_builder -> string_builder = <fun>

val char_at : string_builder -> int -> char = <fun>

val sub_string : string_builder -> int -> int -> string_builder = <fun>

val cost : string_builder -> int = <fun>

val random_word : unit -> string = <fun>

val random_leaf : unit -> string_builder = <fun>

val random_string : int -> string_builder = <fun>

val list_of_string : string_builder -> string list = <fun>

val init_min : string_builder list -> int = <fun>

val least_index_cost : string_builder list -> int = <fun>

val concat_least_cost : string_builder list -> int -> string_builder list = <fun>

val balance : string_builder -> string_builder = <fun>

val random_generator : unit -> string_builder list = <fun>

val list_cost : string_builder list -> int list = <fun>

val list_cost_balanced : string_builder list -> int list = <fun>

val sum_list : int list -> int = <fun>

val max_list : 'a list -> 'a = <fun>

val min_list : 'a list -> 'a = <fun>

val avg_list : int list -> float = <fun>

val med_list : int list -> float = <fun>

val pnl : unit -> int * int * float * float * int * int * float * float * int = <fun>
```