

Arbres non binaires et Tries

March 2, 2022

Rappel Le contenu du cours auquel se TP fait référence se trouve aussi à l'adresse https://marcdefalco.github.io/chap_structuresdonnees_arbres.html

1 Arbres non binaires

1.1 Premières fonctions

On va considérer ici les arbres d'arité quelconque avec des listes d'enfants pour les nœuds.

```
[1]: type 'a arbre = { etiquette : 'a; enfants : 'a arbre list }
```

```
[1]: type 'a arbre = { etiquette : 'a; enfants : 'a arbre list; }
```

Afin d'effectuer des tests, on pourra considérer l'arbre suivant :

```
[2]: let t = { etiquette = 1;
  enfants = [
    { etiquette = 2; enfants = [
      { etiquette = 5; enfants = [] }
    ] };
    { etiquette = 3; enfants = [
      { etiquette = 6; enfants = [
        { etiquette = 10; enfants = [] }
      ] };
      { etiquette = 7; enfants = [] }
    ] };
    { etiquette = 4; enfants = [
      { etiquette = 8; enfants = [] };
      { etiquette = 9; enfants = [] }
    ] }
  ] }
```

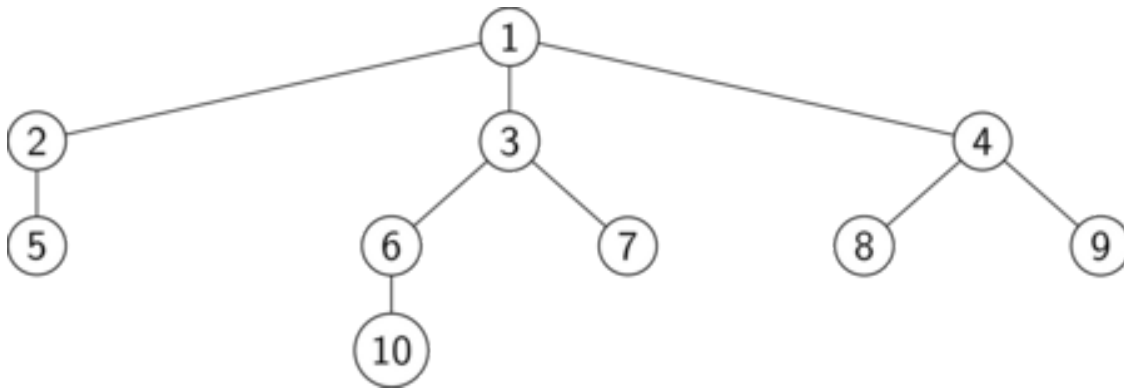
```
[2]: val t : int arbre =
  {etiquette = 1;
  enfants =
    [{etiquette = 2; enfants = [{etiquette = 5; enfants = []}]};
    {etiquette = 3;
    enfants =
```

```

    [{etiquette = 6; enfants = [{etiquette = 10; enfants = []}]}];
    {etiquette = 7; enfants = []}];
{etiquette = 4;
 enfants =
  [{etiquette = 8; enfants = []}; {etiquette = 9; enfants = []}]]}]

```

Cet arbre est représenté ici :



Comme présenté dans le cours, on peut réaliser des fonctions mutuellement récursives pour travailler avec ces arbres et sur la forêt induite par les enfants d'un nœud :

```

[3]: let rec taille_arbre a =
      1 + taille_foret a.enfants
      and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q

```

```

[3]: val taille_arbre : 'a arbre -> int = <fun>
      val taille_foret : 'a arbre list -> int = <fun>

```

Question Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre sur le modèle de la fonction précédente. On fera attention au fait qu'il n'est pas possible de représenter l'arbre vide mais la forêt vide jouera le même rôle vis-à-vis de `-1`.

```

[4]: (* BEGIN SOLUTION *)
      let rec hauteur_arbre a =
        1 + hauteur_foret a.enfants
        and hauteur_foret l = match l with
          | [] -> -1
          | t::q -> max (hauteur_arbre t) (hauteur_foret q)
      (* END SOLUTION *)

```

```

[4]: val hauteur_arbre : 'a arbre -> int = <fun>
      val hauteur_foret : 'a arbre list -> int = <fun>

```

```
[5]: hauteur_arbre { etiquette=(); enfants = [] }
```

```
[5]: - : int = 0
```

```
[6]: assert (hauteur_arbre { etiquette=(); enfants = [] } = 0);  
      assert (hauteur_arbre t = 3)
```

```
[6]: - : unit = ()
```

Question A l'aide d'un parcours récursif, écrire une fonction `affiche_etiquettes : int tree -> unit` qui affiche toutes les étiquettes d'un `int tree`.

Note : on pourra écrire `Printf.printf "%d\n%!" n` pour afficher `n` sur une ligne et forcer l'affichage du buffer même avec Jupyter.

```
[7]: (* BEGIN SOLUTION *)  
      let rec affiche_etiquettes a =  
          Printf.printf "%d\n%!" a.etiquette;  
          affiche_foret a.enfants  
          and affiche_foret l = match l with  
              | [] -> ()  
              | t::q -> affiche_etiquettes t; affiche_foret q  
      (* END SOLUTION *)
```

```
[7]: val affiche_etiquettes : int arbre -> unit = <fun>  
      val affiche_foret : int arbre list -> unit = <fun>
```

```
[8]: (* Doit afficher sur plusieurs lignes 1 2 5 3 6 10 7 4 8 9 avec un traitement_  
      ↪prefixe *)  
      affiche_etiquettes t
```

```
1  
2  
5  
3  
6  
10  
7  
4  
8  
9
```

```
[8]: - : unit = ()
```

1.2 Représentation par des arbres binaires

On va maintenant revenir ici sur la représentation d'un 'a arbre en tant qu'arbre binaire. Pour cela, on va avoir besoin d'étiquettes factices pour les nouveaux nœuds qui ne servent qu'à supporter la structure des nœuds d'arité > 2. On va ainsi les représenter par des a option arbre_bin où l'étiquette sera celle des nœuds factices.

```
[9]: type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

```
[9]: type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

Question On va commencer par écrire une fonction `peigne : 'a option arbre_bin list -> 'a option arbre_bin` qui va transformer une forêt d'arbres binaires en sa représentation en peigne à l'aide de nœuds d'étiquettes `None`. **Note** Dans le cas où la liste est vide, on renverra l'arbre vide et si c'est un singleton, on renverra directement l'arbre qu'il contient.

```
[10]: let rec peigne l =  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> Nil  
      | [x] -> x  
      | a::q -> Noeud(a, None, peigne q)  
      (* END SOLUTION *)
```

```
[10]: val peigne : 'a option arbre_bin list -> 'a option arbre_bin = <fun>
```

```
[11]: let a i = Noeud(Nil, Some i, Nil) in  
      assert( peigne [] = Nil);  
      assert( peigne (List.init 1 a) = Noeud(Nil, Some 0, Nil));  
      assert( peigne (List.init 2 a) = Noeud (Noeud (Nil, Some 0, Nil), None, Noeud_  
        ↪(Nil, Some 1, Nil)) );  
      assert( peigne (List.init 3 a) = Noeud (Noeud (Nil, Some 0, Nil), None,  
        Noeud (Noeud (Nil, Some 1, Nil), None, Noeud (Nil, Some 2, Nil))) );  
      "OK"
```

```
[11]: - : string = "OK"
```

Question Écrire une fonction `convert : 'a arbre -> 'a option arbre_bin` qui va réaliser la conversion vue en classe.

```
[12]: let rec convert a =  
      (* BEGIN SOLUTION *)  
      match a.enfants with  
      | [] -> Noeud (Nil, Some a.etiquette, Nil)  
      | a1::q -> Noeud(convert a1, Some a.etiquette,  
        peigne (List.map convert q))
```

```
(* END SOLUTION *)
```

```
[12]: val convert : 'a arbre -> 'a option arbre_bin = <fun>
```

Afin de tester votre fonction, on vous donne une fonction d’affichage textuel d’un tel arbre :

```
[13]: let rec print_arbre_bin pref t =  
  match t with  
  | Nil -> Printf.printf "%snil\n%!" pref  
  | Noeud(g,x,d) -> Printf.printf "%s%s\n%!" pref (match x with None -> "*" |  
↳Some n -> string_of_int n);  
    print_arbre_bin (String.make (String.length pref) ' ' ^ "|" ^ String.  
↳make 5 '-') g;  
    print_arbre_bin (String.make (String.length pref) ' ' ^ "|" ^ String.  
↳make 5 '-') d
```

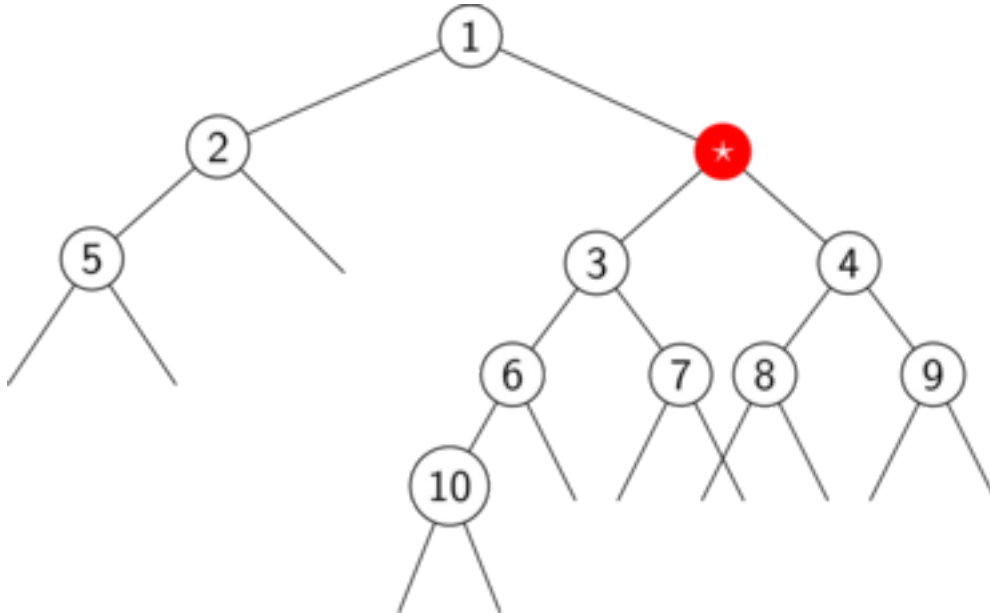
```
[13]: val print_arbre_bin : string -> int option arbre_bin -> unit = <fun>
```

```
[14]: print_arbre_bin "" (convert t)
```

```
1  
|-----2  
    |-----5  
        |-----nil  
        |-----nil  
    |-----nil  
|-----*  
    |-----3  
        |-----6  
            |-----10  
                |-----nil  
                |-----nil  
            |-----nil  
        |-----7  
            |-----nil  
            |-----nil  
    |-----4  
        |-----8  
            |-----nil  
            |-----nil  
        |-----9  
            |-----nil  
            |-----nil
```

```
[14]: - : unit = ()
```

L'affichage doit correspondre à la représentation suivante :



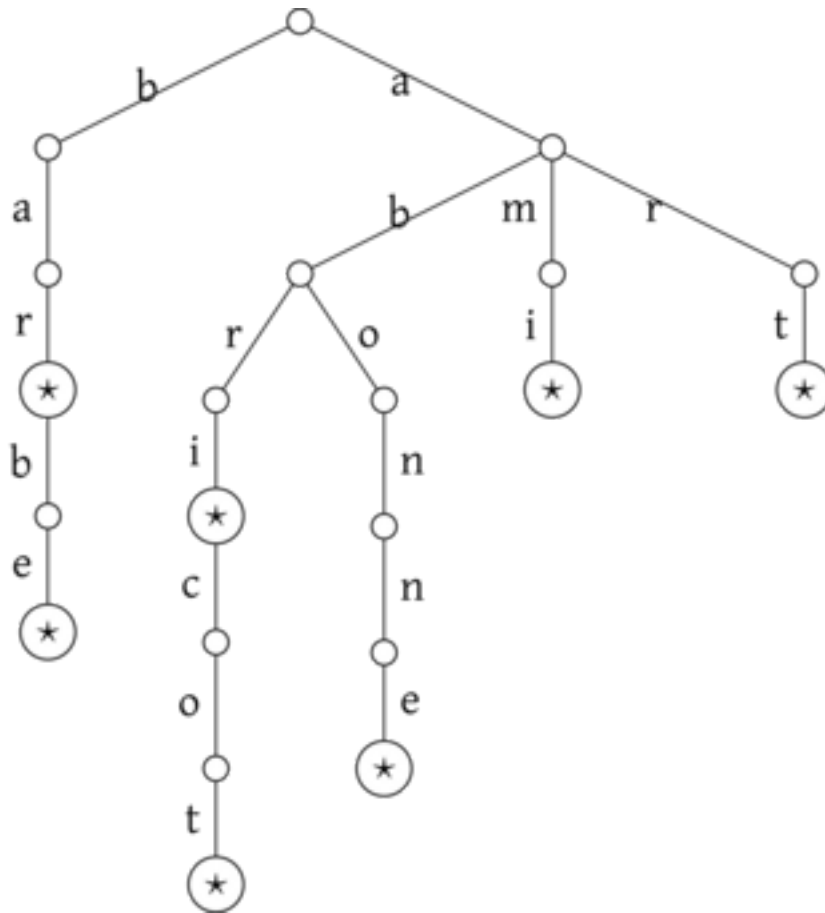
2 Tries

On va reprendre les tries vus en cours pour représenter un ensemble de mots. Pour cela, on réutilise le type :

```
[15]: type trie = {
      mot : bool;
      enfants : (char * trie) list
    }
```

```
[15]: type trie = { mot : bool; enfants : (char * trie) list; }
```

On donne ici un exemple de trie pour pouvoir effectuer des tests. Il correspond à l'arbre vu en



classe :

```

[16]: let t = {mot = false;
  enfants =
    [('b',
      {mot = false;
        enfants =
          [('a',
            {mot = false;
              enfants =
                [('r',
                  {mot = true;
                    enfants =
                      [('b',
                        {mot = false; enfants = [('e', {mot = true; enfants =
↳ [[]]})])])])])])])])])])]);
    ('a',
      {mot = false;
        enfants =
          [('b',
            {mot = false;
              enfants =
                [('r',

```



```

    enfants =
      [('i',
        {mot = true;
         enfants =
           [('c',
            {mot = false;
             enfants =
               [('o',
                {mot = false;
                 enfants = [('t', {mot = true; enfants =
[]}})]})]);
              ('o',
                {mot = false;
                 enfants =
                   [('n',
                    {mot = false;
                     enfants =
                       [('n',
                        {mot = false;
                         enfants = [('e', {mot = true; enfants = []})])])]);
                     ('m', {mot = false; enfants = [('i', {mot = true; enfants = []})])});
                     ('r', {mot = false; enfants = [('t', {mot = true; enfants =
[]})])})])])])

```

2.1 Premières manipulations

Question Écrire une fonction `taille : trie -> int` qui renvoie le nombre de nœuds d'un trie et une fonction `nb_mots : trie -> int` qui renvoie le nombre de mots d'un trie. On s'inspirera directement des fonctions de la première partie du TP.

```

[17]: let rec taille a =
      1 + taille_foret a.enfants
    and taille_foret l = match l with
      | [] -> 0
      | (_, t) :: q -> taille t + taille_foret q

    let rec nb_mots a =
      (if a.mot then 1 else 0) + nb_mots_foret a.enfants
    and nb_mots_foret l = match l with
      | [] -> 0
      | (_, t) :: q -> nb_mots t + nb_mots_foret q

```

```

[17]: val taille : trie -> int = <fun>
      val taille_foret : (char * trie) list -> int = <fun>

```

```
[17]: val nb_mots : trie -> int = <fun>
      val nb_mots_foret : (char * trie) list -> int = <fun>
```

```
[18]: assert (taille t = 21);
      assert(nb_mots t = 7);
      "OK"
```

```
[18]: - : string = "OK"
```

Question Écrire une fonction `trie_apres_char` tel que l'appel à `trie_apres_char l c` va renvoyer `Some t'` si il y a un couple (c, t') dans la liste `l` et `None` sinon. On pourra s'en servir avec `trie_apres_char t.enfants c` pour trouver le trie potentiel dont l'arête est étiquetée par `c`.

```
[19]: let rec trie_apres_char (l:(char * trie) list) (c:char) =
      (* BEGIN SOLUTION *)
      match l with
      | [] -> None
      | (c',t)::q -> if c = c' then Some t else trie_apres_char q c
      (* END SOLUTION *)
```

```
[19]: val trie_apres_char : (char * trie) list -> char -> trie option = <fun>
```

```
[20]: assert( trie_apres_char t.enfants 'b' = Some
      {mot = false;
      enfants =
        [('a',
          {mot = false;
          enfants =
            [('r',
              {mot = true;
              enfants =
                [('b',
                  {mot = false; enfants = [('e', {mot = true; enfants =
↳ []})])])])])])])});
      assert(trie_apres_char t.enfants 'c' = None);
      "OK"
```

```
[20]: - : string = "OK"
```

Question En déduire une fonction `contient : trie -> string -> bool` qui teste si un mot est présent dans le trie.

```
[21]: let contient (t:trie) (s:string) : bool =
  (* BEGIN SOLUTION *)
  let rec aux t i =
    if i = String.length s
    then t.mot
    else match trie_apres_char t.enfants s.[i] with
      | None -> false
      | Some t' -> aux t' (i+1)
  in aux t 0
  (* END SOLUTION *)
```

```
[21]: val contient : trie -> string -> bool = <fun>
```

```
[22]: assert(contient t "bar");
assert(not (contient t "barb"));
assert(not (contient t ""));
"OK"
```

```
[22]: - : string = "OK"
```

Question Écrire une fonction `enumere : trie -> string list` qui renvoie tous les mots contenus dans un trie par un parcours récursif.

Indication : on pourra utiliser :

- `List.concat` pour passer d'une liste de listes à la concaténée de ses éléments. Ainsi `List.concat [ [1;2]; [3]; [4;5] ]` va renvoyer `[1;2;3;4;5]`.
- `String.make 1 c` transforme le caractère `c` en chaîne de caractère
- On rappelle que `^` permet de concaténer deux chaînes.

```
[23]: let enumere (t:trie) =
  (* BEGIN SOLUTION *)
  let rec aux t acc =
    let l = List.concat (List.map (fun (c,t') -> aux t' (acc ^ String.make 1 c)) t.enfants) in
    if t.mot then acc :: l else l
  in aux t ""
  (* END SOLUTION *)
```

```
[23]: val enumere : trie -> string list = <fun>
```

```
[24]: (* Doit afficher à permutation près ["bar"; "barbe"; "abri"; "abricot";
  ↪ "abonne"; "ami"; "art"] *)
enumere t
```

```
[24]: - : string list = ["bar"; "barbe"; "abri"; "abricot"; "abonne"; "ami"; "art"]
```

2.2 Ajout d'un mot à un trie

Pour effectuer l'ajout d'un mot à un trie, on va commencer par écrire une fonction qui étant donné une chaîne de caractère $s = "c_0 \dots c_{n-1}"$ et un entier $i \leq n$ va renvoyer le trie t correspondant à la



branche

Question écrire la fonction `suffixe : int -> string -> trie`

```
[25]: let rec suffixe (i:int) (s:string) : trie =
  (* BEGIN SOLUTION *)
  let n = String.length s in
  if i = n
  then { mot = true; enfants = [] }
  else { mot = false; enfants = [ (s.[i], suffixe (i+1) s) ] }
  (* END SOLUTION *)
```

```
[25]: val suffixe : int -> string -> trie = <fun>
```

```
[26]: assert (suffixe 0 "bar" = {mot = false;
  enfants =
    [ ('b',
      {mot = false;
        enfants =
          [ ('a', {mot = false; enfants = [ ('r', {mot = true; enfants = [] }) ] }) ] }) ]
    )
  =>);
assert( suffixe 1 "bar" = {mot = false;
  enfants =
    [ ('a', {mot = false; enfants = [ ('r', {mot = true; enfants = [] }) ] }) ] ) ;
assert( suffixe 2 "bar" = {mot = false; enfants = [ ('r', {mot = true; enfants = [] }) ] } );
assert( suffixe 3 "bar" = {mot = true; enfants = [] } );
"OK"
```

```
[26]: - : string = "OK"
```

Question Écrire une fonction `avec_trie_pour_char : (char * trie) list -> char -> trie -> (char * trie) list` telle que `avec_trie_pour_char l c t` va renvoyer la liste obtenue en remplaçant `(c,t')` par `(c,t)` dans `l` si un tel couple est présent, ou en ajoutant le couple `(c,t)` sinon à la fin.

```
[27]: let rec avec_trie_pour_char l c t =  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> [ (c, t) ]  
      | (c', t') :: q -> if c = c' then (c, t) :: q else (c', t') ::  
      ↪ avec_trie_pour_char q c t  
      (* END SOLUTION *)
```

```
[27]: val avec_trie_pour_char : ('a * 'b) list -> 'a -> 'b -> ('a * 'b) list =  
      <fun>
```

```
[28]: assert( avec_trie_pour_char [ ('a', { mot=false; enfants=[] }) ; ('b', {  
      ↪ mot=false; enfants=[] }) ] 'b' { mot=true; enfants=[] } = [ ('a', {mot =  
      ↪ false; enfants = []}); ('b', {mot = true; enfants = []}) ] );  
assert( avec_trie_pour_char [ ('a', { mot=false; enfants=[] }) ; ('b', {  
      ↪ mot=false; enfants=[] }) ] 'c' { mot=true; enfants=[] } = [ ('a', {mot =  
      ↪ false; enfants = []}); ('b', {mot = false; enfants = []}); ('c', {mot = true;  
      ↪ enfants = []}) ] );  
"OK"
```

```
[28]: - : string = "OK"
```

Question Écrire une fonction `ajoute : trie -> string -> trie` qui renvoie le trie obtenu en ajoutant un nouveau mot.

```
[29]: let ajoute t s =  
      (* BEGIN SOLUTION *)  
      let n = String.length s in  
      let rec aux t i =  
          if i = n  
          then { t with mot = true }  
          else let t' = match trie_apres_char t.enfants s.[i] with  
              | None -> suffixe (i+1) s  
              | Some t' -> aux t' (i+1) in  
              { t with enfants = avec_trie_pour_char t.enfants s.[i] t' }  
      in aux t 0  
      (* END SOLUTION *)
```

```
[29]: val ajoute : trie -> string -> trie = <fun>
```

On va utiliser la fonction précédente pour retrouver le trie donné en début de partie.

```
[30]: let rec trie_of_list l = match l with
      | [] -> { mot = false; enfants = [] }
      | s::q -> ajoute (trie_of_list q) s
```

```
[30]: val trie_of_list : string list -> trie = <fun>
```

```
[31]: let t' = trie_of_list [ "bar"; "barbe"; "abri"; "abricot"; "abonne"; "ami";
      ↪ "art" ]
```

```
[31]: val t' : trie =
      {mot = false;
      enfants =
        [('a',
          {mot = false;
          enfants =
            [('r', {mot = false; enfants = [('t', {mot = true; enfants = []})})]);
            ('m', {mot = false; enfants = [('i', {mot = true; enfants = []})})]);
            ('b',
              {mot = false;
              enfants =
                [('o',
                  {mot = false;
                  enfants =
                    [('n',
                      {mot = false;
                      enfants =
                        [('n',
                          {mot = false;
                          enfants =
                            [('e', {mot = true; enfants = []})})])])])]);
                    ('r',
                      {mot = false;
                      enfants =
                        [('i',
                          {mot = true;
                          enfants =
                            [('c',
                              {mot = false;
                              enfants =
                                [('o',
                                  {mot = false;
                                  enfants =
                                    [('t', {mot = true; enfants =
```

On va maintenant récupérer une grande liste de mots dans le fichier `wordlist` à raison d'un mot par ligne :

```
[32]: val wordlist: string list =
  ["a"; "a-horizon"; "a-ok"; "aardvark"; "aardwolf"; "ab"; "aba"; "abaca";
   "abacist"; "aback"; "abactinal"; "abacus"; "abaddon"; "abaft";
   "abalienate"; "abalienation"; "abalone"; "abampere"; "abandon";
   "abandoned"; "abandonment"; "abarticulation"; "abase"; "abased";
   "abasement"; "abash"; "abashed"; "abashment"; "abasia"; "abasic"; "abate";
   "abatement"; "abating"; "abatis"; "abatjour"; "abattis"; "abattoir";
   "abaxial"; "abba"; "abbacy"; "abbatial"; "abbatical"; "abbatis"; "abbe";
   "abbess"; "abbey"; "abbot"; "abbreviate"; "abbreviated"; "abbreviation";
   "abbreviature"; "abc"; "abcoulomb"; "abdal"; "abderite"; "abdicable";
   "abdicant"; "abdicate"; "abdication"; "abdicator"; "abditory"; "abditos";
   "abdomen"; "abdominal"; "abdominocentesis"; "abdominoscope";
   "abdominoscopy"; "abdominous"; "abdominousness"; "abdominovesical";
   "abduce"; "abducent"; "abduct"; "abduction"; "abductive"; "abductor";
   "abeam"; "abecedarian"; "abecedarius"; "abecedary"; "abed"; "abel";
   "abelia"; "abelmoschus"; "abelmosk"; "abends"; "aber"; "aberdeen";
   "aberdevine"; "aberrance"; "aberrant"; "aberration"; "abest"; "abet";
   "abetalipoproteinemia"; "abetment"; "abettor"; "abeunt"; "abeyance";
   "abeyant"; "abfarad"; "abhenry"; "abhor"; "abhorrence"; "abhorrent";
   "abhorrer"; "abibis"; "abidance"; "abide"; "abiding"; "abidjan";
   "abience"; "abient"; "abies"; "abigail"; "abiit"; "abilities"; "ability";
```

"abiogenesis"; "abiogenetic"; "abiogenist"; "abiotrophy"; "abito";
 "abject"; "abjection"; "abjectly"; "abjectness"; "abjunction";
 "abjuration"; "abjurationabjurement"; "abjure"; "abkari"; "ablactation";
 "ablated"; "ablation"; "ablative"; "ablaut"; "ablaze"; "ablaze(p)";
 "able"; "ablebodied"; "ablegate"; "ableism"; "ableness"; "ablepharia";
 "ablepsia"; "ablepsy"; "abloom"; "ablude"; "ablution"; "ablutionary";
 "abnaki"; "abnegation"; "abnegator"; "abnormal"; "abnormality";
 "abnormalize"; "abnormally"; "abnormis"; "abnormity"; "abnormous";
 "aboard"; "abocclusion"; "abode"; "abodement"; "aboding"; "abohm";
 "aboideau"; "abois"; "aboiteau"; "abolengo"; "abolish"; "abolishable";
 "abolishment"; "abolition"; "abolitionary"; "abolitionism";
 "abolitionist"; "abolitionize"; "abomasal"; "abomasum"; "abominable";
 "abominate"; "abomination"; "abominator"; "aborad"; "aboral"; "abord";
 "aboriginal"; "aborigine"; "aborigines"; "aborning"; "abort";
 "aborticide"; "abortifacient"; "abortion"; "abortionist"; "abortive";
 "abortively"; "abortus"; "abound"; "abounding"; "about"; "about(p)";
 "about-face"; "abouts"; "above"; "above-mentioned"; "aboveboard";
 "aboveground"; "abovementioned"; "abovesaid"; "abovestairs"; "abra";
 "abracadabra"; "abrachia"; "abrade"; "abraded"; "abrader"; "abraham";
 "abramis"; "abbranchiate"; "abrasion"; "abrasive"; "abreast"; "abrege";
 "abreption"; "abridge"; "abridged"; "abridger"; "abridgment"; "abroach";
 "abroad"; "abrocoma"; "abrocome"; "abrogate"; "abrogated"; "abrogation";
 "abronia"; "abrupt"; "abruption"; "abruptly"; "abruptness"; "abruzzi";
 "abscess"; "abscessed"; "abscond"; "abscision"; "abscissa"; "abscission";
 "abscond"; "absconder"; "abscondment"; "absence"; "absens"; "absent";
 "absentee"; "absenteeism"; "absently"; "absentminded"; "absentmindedness";
 "absento"; "absents"; "absinth"; "absinthe"; "absolute"; "absolutely";
 "absoluteness"; "absolution"; "absolutism"; "absolutist"; "absolve";
 "absolved"; "absolver"; "absolvitory"; "absolvitur"; "absonant";
 "absonous"; "absorb"; "absorbable"; "absorbate"; "absorbed";
 "absorbefacient"; "absorbency"; "absorbent"; "absorber"; "absorbing";
 "absorption"; "absorptivi"... (* string length 12; truncated *);
 "absquatul"... (* string length 12; truncated *); "abstain";
 "abstain"... (* string length 9; truncated *);
 "abstem"... (* string length 10; truncated *);
 "abste"... (* string length 12; truncated *);
 "abst"... (* string length 14; truncated *);
 "abs"... (* string length 10; truncated *);
 "ab"... (* string length 8; truncated *);
 "a"... (* string length 10; truncated *);
 ""... (* string length 10; truncated *); ...]

Question En déduire un trie permettant de représenter cette wordlist et comparez :

- Le nombre de caractère total
- Le nombre de nœuds du trie

Que peut-on en déduire ?

```
[33]: (* BEGIN SOLUTION *)  
      let taille_totale = List.fold_left (+) 0 (List.map String.length wordlist) in  
      taille_totale, taille (trie_of_list wordlist)  
      (* END SOLUTION *)
```

```
[33]: - : int * int = (595775, 229239)
```