1- (a)

```
for (i=1 ; i ≤ longueur(l) ; i++ )
    afficher (Element(l,i)) ;
```

(b)

```
boolean appartient (liste l, E x)
{
    for (i=1 ; i ≤ longueur(l) ; i++ )
        if (Element(l,i) == x) return True;
    return false ;
}
```

Affichage des "singletons" :

```
for (i=1 ; i ≤ longueur(l) ; i++)
    if (nb_occurrences (l, Element(l,i)) == 1 )
        afficher (Element(l,i))
```

```
int nb_occurrences (liste l, E x)
{ c=0 ;
  for (i=1 ; i ≤ longueur(l) ; i++)
      if (Element(l,i) == x)  c++
  return c ;
}
```

Complexité : nb_occurrences : $O(n)$ n = longueur

affichage : $n \times$ nb_occ

$\rightarrow O(n^2)$

en supposant que Element() a un coût constant.

09²³

1- suite

(c) Idée : on parcourt la liste depuis le début, et pour chaque élément, on supprime toutes les occurrences de cet élément qui arrivent après la position courante.

```
p = 1 ;
do {
    e = Element (l, p) ;
    for (i = p ; i <= Longueur (l) ; i++)
    i = p+1 ;
    do {
        if (e == Element (l, i))
            Supprimer (l, i) ;
        else
            i++ ;
    } while (i <= Longueur (l)) ;
    p++ ;
} while (p <= Longueur (l)) ;
```

e est le p^e el^t de la liste

suppression de toutes les (autres) occurrences de e après la position p.

pas d'incrémental de i (car le i^e el^t a changé).

position suivante

Rq : Algo ~~général~~ de complexité $O(n^2)$

2 -

(a) Recherche d'un mot dans l'arbre

(i) Version itérative :

```
boolean appartient (char[] mot)
{
    X = racine();
    for (i = 1 ; i ≤ length(mot) ; i++)
        if (! existe_fils_lettre(X, mot[i]))
            return false ;
        else
            X = fils_lettre(X, mot[i]) ;
    return existe_fils_lettre(X, '$') ;
}
```

(ii) Version récursive :

Principe : le mot $u = \boxed{x}\boxed{u'}$ appartient à l'arbre si & seulement si le mot u' appartient au sous-arbre ~~d'origine~~ de racine x -

```
boolean app (char[] mot, int debut, noeud r)
{
    if ((debut == length(mot)) &&
        existe_fils_lettre(r, '$'))  return True ;
    else
        if (existe_fils_lettre(r, mot[debut]))
            return app(mot, debut+1, fils_lettre(r, mot[debut]));
        else
            return false ;
}
```

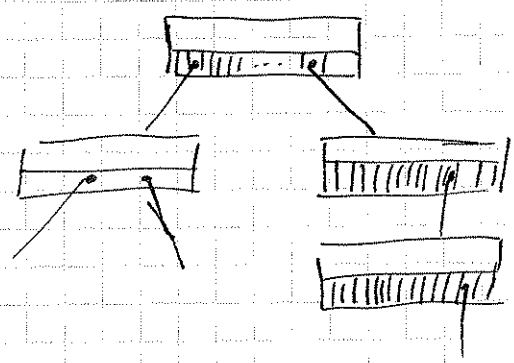
appel initial : app(mot, 1, racine())

2. Suite

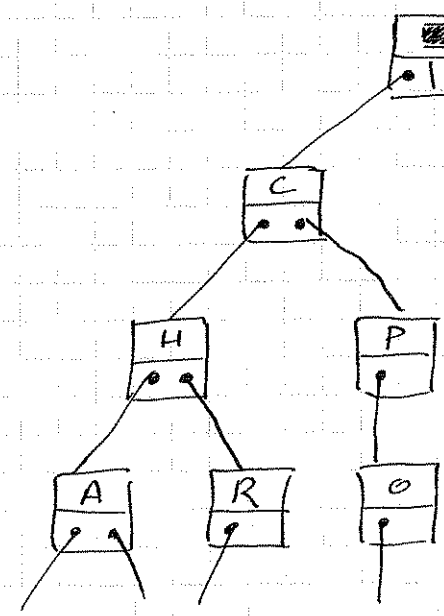
(b) Structure de données.

Le nombre de fils est potentiellement de 27. Une structure chaînée père-fils (α) est donc peu envisageable.
On peut s'engager vers une structure chaînée "binarisée" (β), ce qui complique légèrement l'implémentation de la primitive est_fils_lettre().

(α)



(β)



Complexité de la primitive :
pour un nœud donné, il faut
(1) passer à son (premier) fils
(2) suivre la liste des frères jusqu'à trouver la valeur recherchée.
la complexité est donc bornée par le nombre de fils de chaque fratrie.

N.B. On peut aussi proposer une sol. contiguë

2- suite

(c) Pour la version itérative il serait plus commode d'avoir les primitives frère & fils; pour la version récursive la liste des fils serait plus pertinente.

Version itérative :

```
X = racine (A)
do { if { existe_fils (X)
       { X = premier_fils (X) ;
         empiler (X)
   else { afficher la pile
          while (! existe_frere (X))
              X = dépiler ()
          X = frere (X)
          dépiler ()
          empiler (X)
} while (X ≠ racine())
```

Version récursive

```
void aff_mot (mot, x)
{
   if (existe_fils_lettre (x, '$'))
       afficher (mot) ;
   for (x='a' ; x ≤ 'z' ; x++)
       if (existe_fils_lettre (x, x))
           aff_mot (mot+x, x) ;
}
```

concaténation

2 - suite

(d) Ajout d'un mot :

C'est comme le parcours (a), avec un changement quand on constate l'absence.

```
x = racine()
for (i = 1 ; i ≤ length(mot) ; i++)
    if (existe_fils_lettre (x, mot[i]))
        x = fils_lettre (x, mot[i])
    else
        x = ajoute_noeud_lettre (x, mot[i])
~~ajoute~~ if (! existe_fils_lettre (x, '$'))
    ajoute_noeud_lettre (x, '$')
```

ajoute_noeud_lettre() renvoie le noeud nouvellement créé.

NB : le code pourrait être simplifié : une fois qu'on ajoute une lettre, toute la suite du mot est ajoutée, sauf qu'il soit nécessaire de tester à chaque fois que le fils existe.

2- Suite

(c) Pour supprimer un mot, il faut partir de la fin, & supprimer les nœuds qui ne sont pas utilisés par d'autres mots.

Donc :

(1) On fait une "descente" jusqu'à la feuille '$'.

(2) On supprime '$'

(3) On passe au père,
et si ce nœud n'a aucun frère,
on le supprime
On recommence,
et l'algorithme se termine dès
que le nœud courant a un frère.