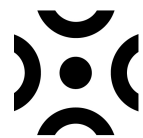
Algorithmique des arbres

Arbres préfixes, Arbres lexicographiques
Représentation d'un lexique

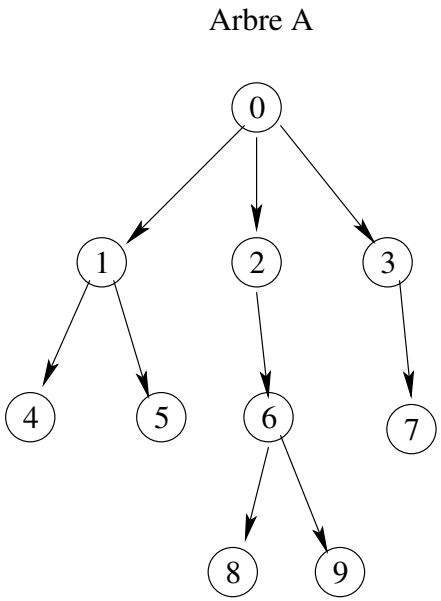
L2 Mathématique et Informatique



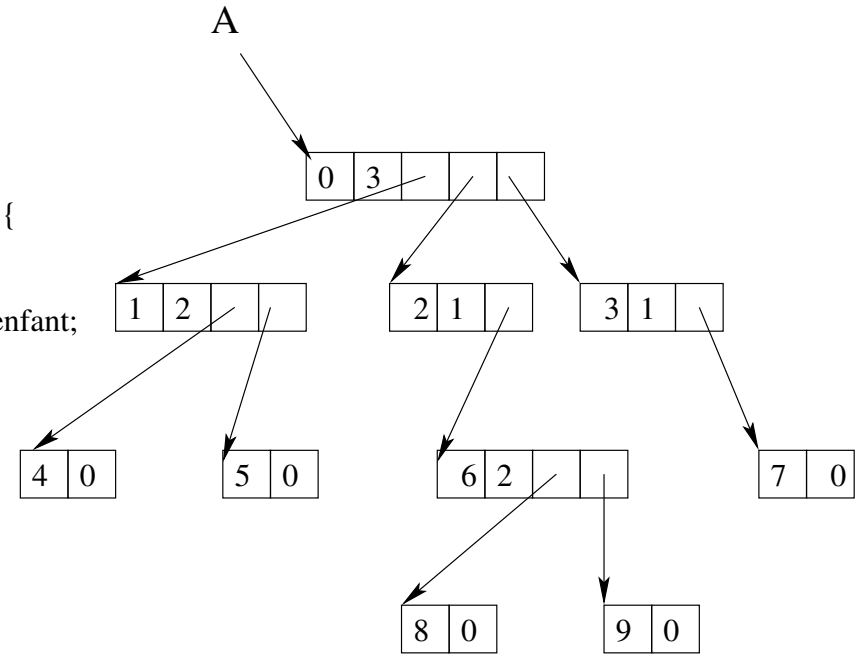
Outline

Implantation par pointeurs 1 / 2

- liens du nœud parent vers chaque enfant



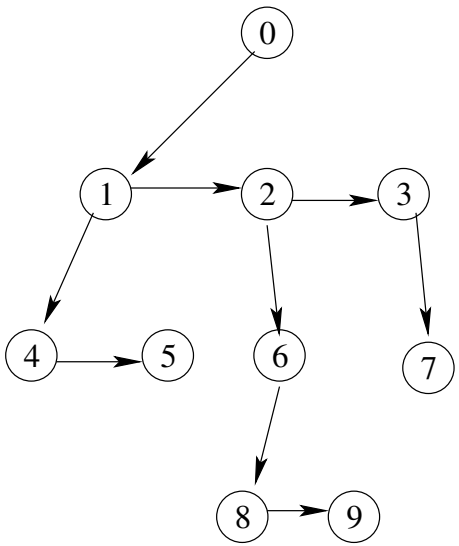

```
typedef struct noeud {
        Element valeur;
        int nbenfant
        struct noeud **enfant;
} Noeud,*Arbre
```



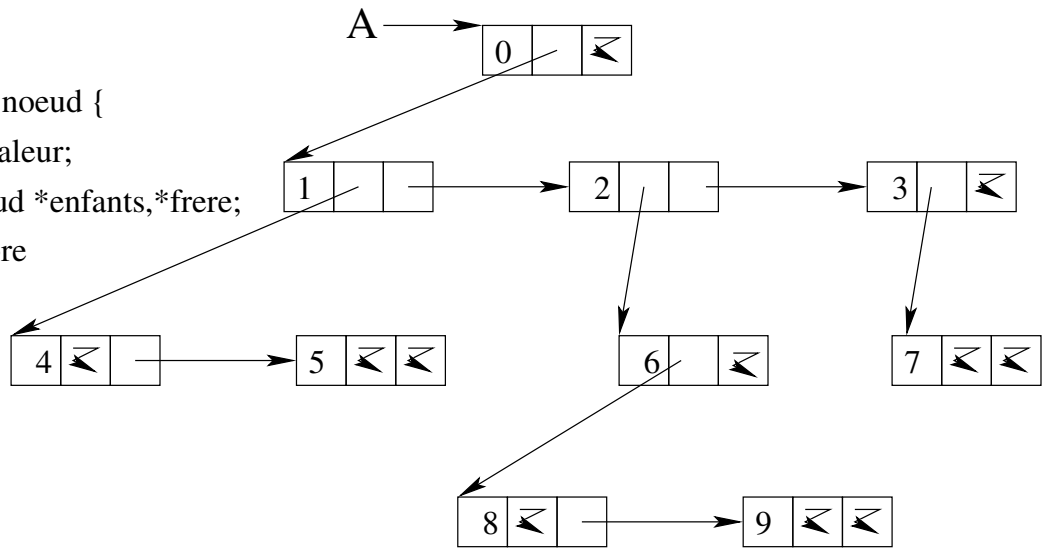
Remarque : On a identifié un arbre à l'adresse de sa racine.

Implantation par pointeurs 2 / 2

- lien du nœud parent vers liste des enfants




```
typedef struct noeud {
    Element valeur;
    struct noeud *enfants,*frere;
} Noeud,*Arbre
```



Remarque : On a identifié un arbre à l'adresse de sa racine.

Représentation d'un arbre n-aire par un arbre binaire

L'ensemble des fils possibles d'un nœud est représenté par une liste chainée.

Un nœud possède donc deux liens:

- l'un vers le suivant dans la liste de ses frères,
- l'autre vers le premier nœud de la liste de ses fils.

Cette représentation est appelée **fils gauche-frère droit**.

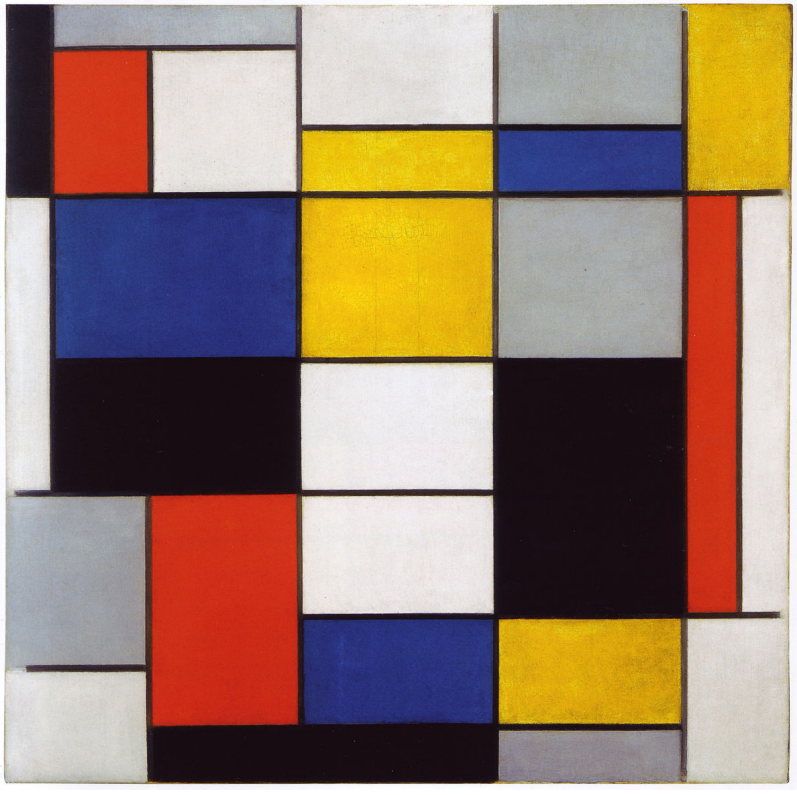
Objectif de la séance

Apprendre à représenter un ensemble de mots par :

- un arbre 27-aire ;
- un arbre fils gauche-frère droit ;
- un arbre lexical ternaire.

Objectif de la séance

Arbre argenté, 1911
Piet Mondrian



Composition A, 1920
Piet Mondrian

Outline

Arbre préfixe

Définition

Soit E un ensemble ordonné.
Un **arbre préfixe** étiqueté par E est un arbre n-aire permettant de stocker un élément de $E^\star$ (*i.e.* une suite d'éléments de E) :

- la racine n'est pas étiqueté ;
- les étiquettes des nœuds sont des éléments de E ;
- l'information codée par un nœud est la suite des étiquettes rencontrées sur le chemin de la racine à ce nœud.

Convention :

Les enfants d'un nœud sont ordonnés selon la valeur de leur étiquette.

Définition

Un arbre préfixe sur $E = \{\mathtt{a}, \mathtt{b}, \cdots, \mathtt{z}\}$ est appelé un **arbre lexicographique**.

Comment identifier les mots préfixes d'autres mots ?

On introduit un caractère spécial "fin de mot" :

- l'ensemble E contiendra toujours ce caractère ;
- c'est le seul caractère dont on est sûr qu'il n'y en a qu'un par mot ;
- ce caractère doit être inférieur à tous les autres éléments de E.

Remarques :

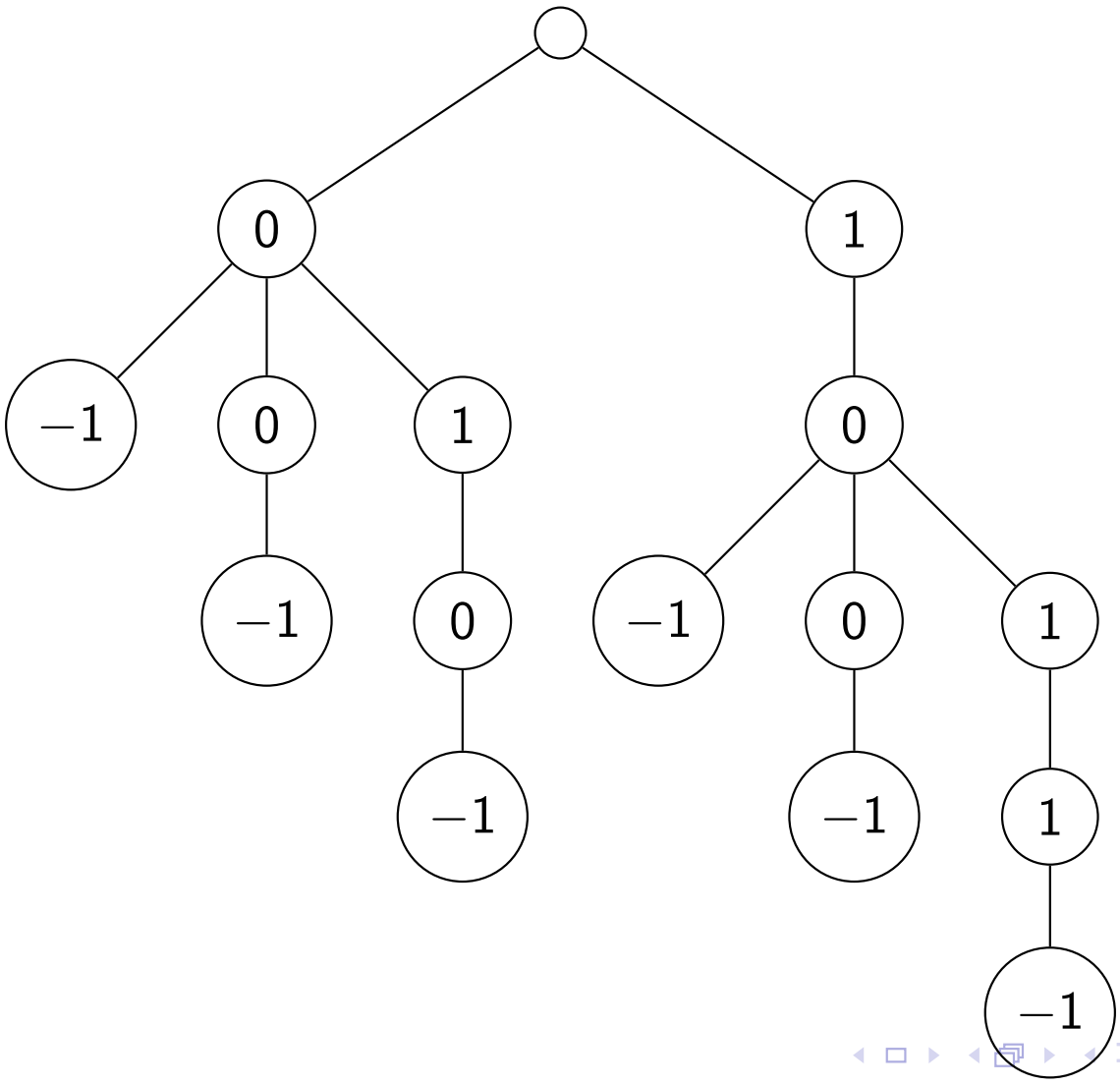
- Un nœud contenant l'étiquette "fin de mot" est nécessairement une feuille de l'arbre préfixe.
- Un mot est codé par le chemin de la racine vers un nœud d'étiquette "fin de mot", donc vers une feuille.

Exemples de caractères "fin de mot" :

- $\rightsquigarrow$ `\0` en `C` si $E = \{\texttt{a}, \texttt{b}, \cdots, \texttt{z}\}$;
- $\rightsquigarrow$ l'espace `' '` ;
- $\rightsquigarrow$ ...

Exemples

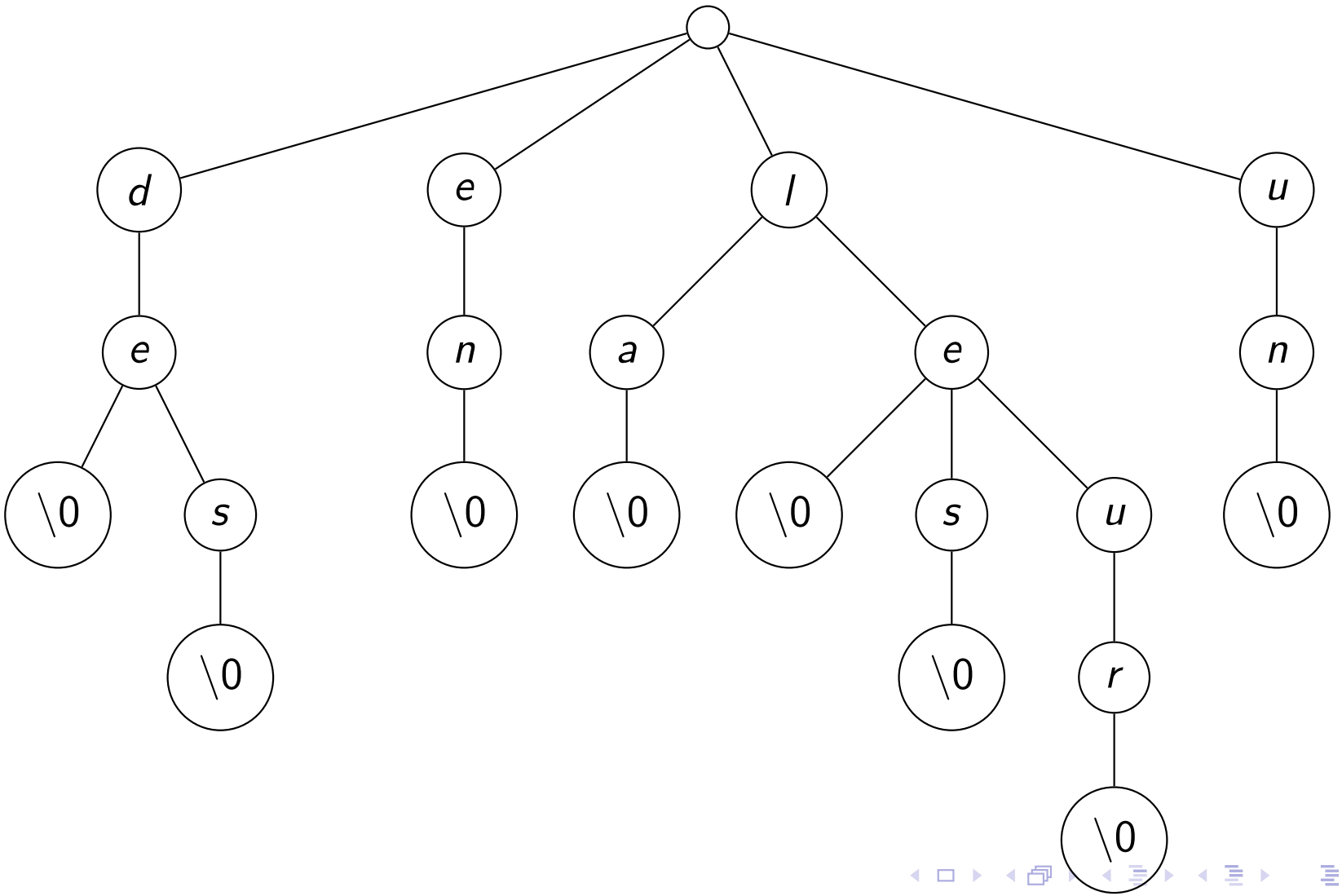
- L'arbre préfixe suivant construit sur $E = \{-1, 0, 1\}$, -1 étant le caractère de "fin de mot", contient les séquences de bits :
  0, 00, 010, 10, 100 et 1011.



Exemples

- L'arbre préfixe suivant construit sur $E = \{\backslash\, 0, \mathtt{a}, \cdots, \mathtt{z}\}$ contient les mots :

`de`, `des`, `en`, `la`, `le`, `les`, `leur`, `un`.



Remarques

1. Dans un arbre préfixe, un nœud pris isolément n'a aucune signification : aucun nœud dans l'arbre ne stocke l'information à laquelle il est associé

2. Seul les chemins de la racine à un nœuds ont une signification :

   ils représentent un PRÉFIXE commun aux suites codés par l'ensemble des feuilles du sous-arbre de racine le nœud considéré.

3. C'est le niveau des nœuds dans l'arbre qui détermine l'indice du terme de la suite codé par le nœud.

Opérations sur les arbres lexicaux

- Recherche d'un mot ;
- Insertion d'une branche ;
- Insertion d'un mot ;
- Suppression d'un mot ;
- Affichage du lexique ;
- Libération de l'arbre.

Applications

- Autocomplétion (T9, barre de navigation google, IDE, $\cdots$)
- Correction orthographique
- $\cdots$

Outline

Représentation d'un lexique avec un tableau statique

Pour représenter un lexique, on peut considérer la structure :

```
1  typedef struct noeudLexico {
2      unsigned char c;
3      struct noeudLexico * enfants[NB_LETTRES + 1];
4  } NoeudLexico, * ArbreLexico;
```

où `NB_LETTRES` est une constante.

- ⇝ Chaque nœud contient une lettre minuscule et possède 27 enfants initialement `NULL` ;
- ⇝ Le premier enfant (indice 0) d'un nœud corresponds au caractère \0 ;
- ⇝ L'enfant situé à l'indice c − `'a'` +1 corresponds au caractère c de l'alphabet ;
- ⇝ La racine contient un caractère qui ne sera pas utilisé (\0, par exemple) ;
- ⇝ Le marqueur de fin de mot \0 est conservé dans l'arbre.

Exemples

Allocation d'un nœud

```
NoeudLexico * alloueNoeud(char c){
    int i;
    NoeudLexico * n = (NoeudLexico *) malloc(sizeof(NoeudLexico)
    if (n != NULL){
        if ('A' <= c && c <= 'Z')
            n->c = c - 'A' + 'a'; // Transformation maj -> min
        else
            n->c = c;
        for (i = 0; i <= NB_LETTRES; i++)
            n->enfants[i] = NULL;
    }
    return n;
}
```

Recherche d'un mot dans l'arbre

Méthode :

On descends le long de la branche devant contenir le mot recherché.
On s'arrête lorsque :

- on a trouvé \0 dans l'arbre : le mot est bien dans le lexique ;
- la branche s'arrête prématurément : un caractère recherché n'est pas dans les enfants d'un nœud et le mot n'est donc pas dans le lexique.

```
int recherche(ArbreLexico A, char * mot){
    if (*mot == '\0')
        return A->enfants[0] != NULL;
    if (A->enfants[*mot - 'a' + 1] != NULL)
        return recherche(A->enfants[*mot - 'a' + 1], mot + 1);
    else
        return 0;
}
```

Ajout d'un mot dans le lexique 1 / 2

Méthode :

On descends le long de la branche de l'arbre contenant le plus long préfixe déjà présent du mot à insérer.
Le suffixe est ajouté caractère par caractère.

```
int insere(ArbreLexico A, char * mot){
    while (*mot != '\0' && A->enfants[*mot - 'a' + 1] != NULL){
        A = A->enfants[*mot - 'a' + 1];
        mot++;
    }
    if (*mot == '\0' && A->enfants[0] == NULL)
        return ajoute_branche(&(A->enfants[0]), mot);
    if (*mot == '\0')
        return 2; // Le mot est déjà présent
    else
        return ajoute_branche(&(A->enfants[*mot - 'a' + 1]),
                              mot);
}
```

Ajout d'un mot dans le lexique 2 / 2

```
int ajoute_branche(ArbreLexico * A, char * mot){
    int prochain_enfant;
    if ((*A = alloueNoeud(*mot)) == NULL)
        return 0;
    if (*mot == '\0')
        return 1;
    else {
        mot++;
        prochain_enfant = (*mot == '\0')? 0: *mot - 'a' + 1;
        return ajoute_branche(&((*A)->enfants[prochain_enfant]),
                              mot);
    }
}
```

$\rightsquigarrow$ `ajoute_branche` s'appelle sur un arbre vide, sinon une partie de l'arbre sera écrasé !

Affichage ordonné de la suite de mot du lexique 1 / 2

Méthode :

Pour chacun des enfants non vide de la racine, on construit le mot dans un tableau en mémorisant les lettres rencontrés lors d'un parcours en profondeur.

Lorsqu'on arrive sur un nœud, on écrit la lettre à l'indice courant :

- ⇝ si cette lettre est `'\0'`, on affiche le mot ;
- ⇝ si on descend vers un enfant non vide, c'est pour compléter le mot ; on avance donc dans le mot.

Attention : Il faut donc transmettre à la fonction un tableau de `char` pour stocker le mot courant et un `int` pour préciser l'indice de la case à remplir.

⇝ Utilisation d'une fonction auxilliaire !

Affichage ordonné de la suite de mot du lexique 2 / 2

```
void affiche_aux(ArbreLexico A, char buffer[], int indice){
    int i;
    if (A != NULL){
        buffer[indice] = A->c;
        if (A->c == '\0')
            printf("%s\n", buffer);
        else {
            for (i = 0; i <= NB_LETTRES; i++)
                affiche_aux(A->enfants[i], buffer, indice + 1);
        }
    }
}
void affiche(ArbreLexico A){
    int i;
    char buffer[MAX_LETTRES];
    if (A != NULL)
        for (i = 0; i <= NB_LETTRES; i++)
            affiche_aux(A->enfants[i], buffer, 0);
}
```

Suppression d'un mot du lexique 1 / 2

Méthode :

La suppression d'un mot du lexique se fait récursivement en remontant la branche stockant le mot et en libèrant les `noeuds` alloués dynamiquement lorsque nécessaire : un nœud est libéré ...

- ⇝ ... lorsque l'on est tout en bas de la branche ;
- ⇝ ... lorsque son seul enfant a été supprimé précédemment et le nœud doit être supprimé.

Suppression d'un mot du lexique 2 / 2

```
int suppression(ArbreLexico * A, char * mot){
    int i, nb_enfants = 0;
    if (*mot == '\0'){
        libere(&((*A)->enfants[0]));
        return 1;
    }
    if (suppression(&((*A)->enfants[*mot-'a'+1]), mot+1) == 1){
        for (i = 0; i <= NB_LETTRES; i++)
            if ((*A)->enfants[i] != NULL){
                nb_enfants++;
            }
        if (nb_enfants == 0){
            libere(A);
            return 1;
        } else
            return 0;
    } else
        return 0;
}
```

Libération d'un arbre

Méthode :

On libère récursivement tous les enfants non vides avant de libérer la mémoire associée à la racine.

```
void libere(ArbreLexico *A){
    int i;
    for (i = 0; i <= NB_LETTRES; i++)
        if ((*A)->enfants[i] != NULL)
            libere(&((*A)->enfants[i]));
    free(*A);
    *A = NULL;
}
```

Avantages / Inconvénients de cette implémentation

- Arbre facile à lire ;
- Facile à implémenter ;
- mais globalement prends énormément de place en mémoire si le lexique est petit

  (stockage de beaucoup de pointeurs `NULL`$\cdots$)
- alors que l'arbre prends raisonnablement de la place pour un *gros* lexique (tous les mots de la langue françcaise, par exemple).

Outline

Représentation d'un lexique par un arbre fils gauche frère droit

Pour représenter un lexique (ensemble de mots), chaque nœud contient une lettre :

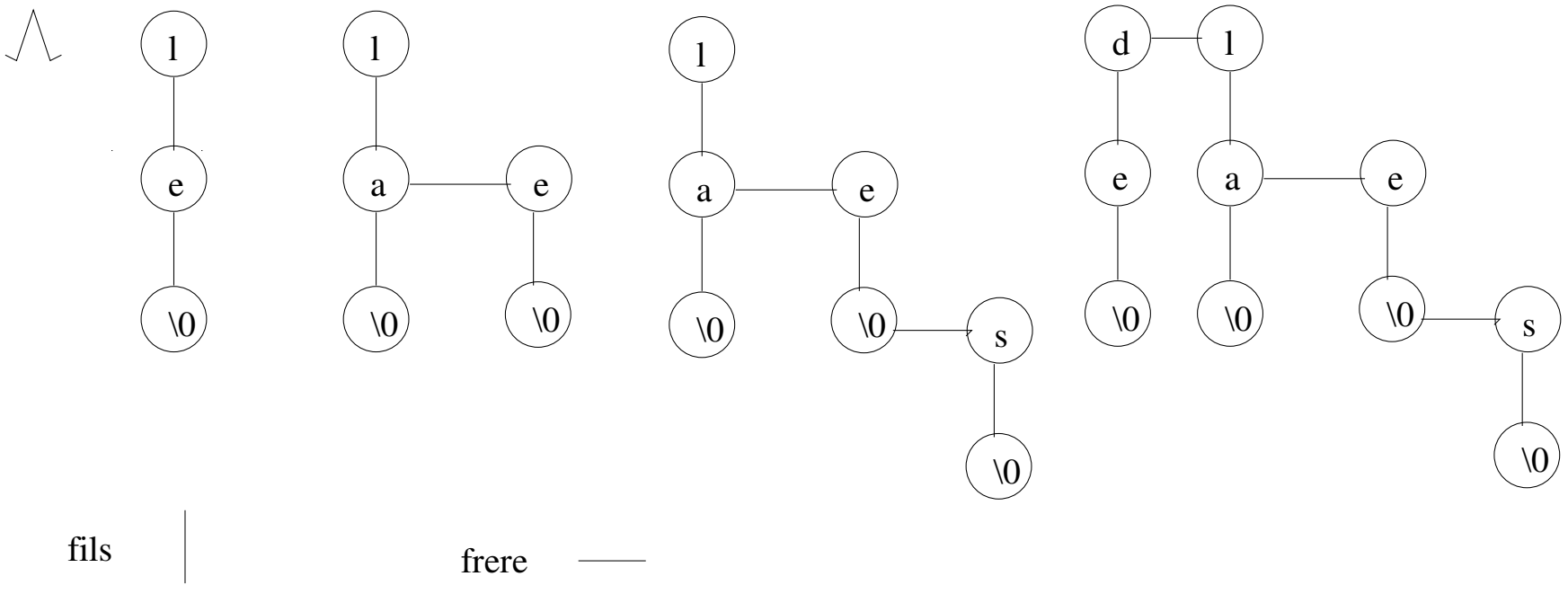
.

```
typedef struct noeud{
  char c;
  struct noeud *filsg, *frered;
} NoeudLexico, *ArbreLexico;
```

On décide d'ordonner les frères en ordre alphabétique.
Le marqueur de fin de mot \0 est conservé dans l'arbre.

Exemples

On ajoute successivement les mots `le,la,les,de,leur,un,des,en` au lexique en partant d'un arbre vide.



fils | frere —

Allocation d'un nœud

```
NoeudLexico * alloue_noeud(char c){
    int i;
    NoeudLexico *n = (NoeudLexico*) malloc(sizeof(NoeudLexico));
    if (n != NULL){
        if ('A' <= c && c <= 'Z')
            n->c = c - 'A' + 'a'; // Transformation maj -> min
        else
            n->c = c;
        n->filsg = n->frered = NULL;
    }
    return n;
}
```

Recherche d'un mot dans un arbre

Méthode :

Si l'arbre n'est pas vide, on cherche la première lettre `l` du mot recherché dans la liste ordonnée des frères en la comparant avec la lettre `c` du nœud courant :

- si `l`<`c`, le mot n'est pas présent ;
- si `l`=`c` :
  - ⇝ si on a trouvé la fin de mot (`'\0'`), le mot est présent ;
  - ⇝ sinon, on cherche la suite du mot dans l'arbre `filsg` ;
- si `l`>`c`, on cherche le mot dans l'arbre `frered`.

```
int recherche(ArbreLexico A, char *mot){
    if (A == NULL || *mot < A->c)
        return 0;
    if (*mot == A->c){
        if (*mot == '\0')
            return 1;
        return recherche(A->filsg, mot + 1);
    }
   return recherche(A->frered, mot);
}
```

Ajout d'un mot dans le lexique 1 / 3

Méthode :

Si l'arbre est vide, le mot est ajouté sous forme d'une liste de fils (une branche).

Sinon l'ajout est basée sur :

- la recherche dans l'arbre du plus long préfixe du mot ;
- l'ajout dans une liste chainée triée (on compare avec la valeur du nœud suivant)
  - ⇝ si la première lettre du mot est plus grande que celle du noeud, on ajoute le mot dans le frère ;
  - ⇝ si la première lettre est trouvée et que la fin du mot n'est pas atteinte, on ajoute le reste du mot dans le fils ;
  - ⇝ si la première lettre est inférieure à celle du nœud, on insère le mot sous forme d'une liste de fils.

Ajout d'un mot dans le lexique 2 / 3

```
void ajoute_branche(ArbreLexico *A, char *mot){
    if ((*A = alloue_noeud(*mot)) != NULL){
        if (*mot != '\0')
            ajoute_branche(&((*A)->filsg),mot + 1);
    }
}
```

Ajout d'un mot dans le lexique 3 / 3

```
/* pas de gestion d'erreur*/
void insere(ArbreLexico *A, char *mot){
    if (*A == NULL)
        ajoute_branche(A,mot);
    else {
    if ((*A)->c < *mot)
        insere(&((*A)->frered), mot);
    else
        if ((*A)->c == *mot && *mot != '\0')
            insere(&((*A)->filsg), mot + 1);
        else
            if (*mot != '\0'){ // Insertion dans la liste
                               // chainée triée des frered
                ArbreLexico tmp = NULL;
                ajoute_branche(&tmp, mot);
                tmp->frered = *A;
                *A = tmp;
        }
    }
}
```

Affichage ordonné de la suite des mots 1 / 2

Méthode :

On construit le mot dans un tableau en mémorisant les lettres rencontrés lors d'un parcours en profondeur (fils gauche d'abord).

Lorsqu'on arrive sur un nœud, on écrit la lettre à l'indice courant :

- ⇝ si cette lettre est `'\0'`, on affiche le mot ;
- ⇝ si on descend vers le fils gauche, c'est pour compléter le mot ; on avance donc dans celui-ci ;
- ⇝ si on passe vers le frère droit, il faudra remplacer la lettre du nœud courant par celle du frère droit : on reste au même indice.

Attention : Il faut donc transmettre à la fonction un tableau de `char` pour stocker le mot courant et un `int` pour préciser l'indice de la case à remplir.

⇝ Utilisation d'une fonction auxilliaire !

Affichage ordonné de la suite des mots 2 / 2

```
void affiche_aux(ArbreLexico A, char buffer[], int indice){
    if (A != NULL){
        buffer[indice] = A->c;
        if (A->c == '\0')
            printf("%s \n",buffer);
        else
            affiche_aux(A->filsg, buffer, indice + 1);
        if (A->frered != NULL)
            affiche_aux(A->frered, buffer, indice);
    }
}

void affiche(ArbreLexico A){
    char buffer[MAX_LETTRES];
    int courant = 0; /*case a remplir*/
    affiche_aux(A, buffer, courant);
}
```

Suppression d'un mot du lexique

- Laissé en exercice !

Libération d'un arbre

Méthode :

S'ils sont non vide, on libère récursivement le fils gauche, puis le frère, avant de libérer la mémoire associée à la racine.

```
void libere(ArbreLexico *A){
    if ((*A)->filsg != NULL)
        libere(&((*A)->filsg));
    if ((*A)->frered != NULL)
        libere(&((*A)->frered));
    free(*A);
    *A = NULL;
}
```

Avantages / Inconvénients de cette implémentation

- Arbre difficile à lire ;
- Implémentation délicate ;
- Aucun stockage de pointeurs `NULL` inutile.
- Nombre d'accès à un nœud légérement plus grand pour la recherche d'un mot que dans l'implémentation précédente.

Outline

Arbre ternaire de recherche

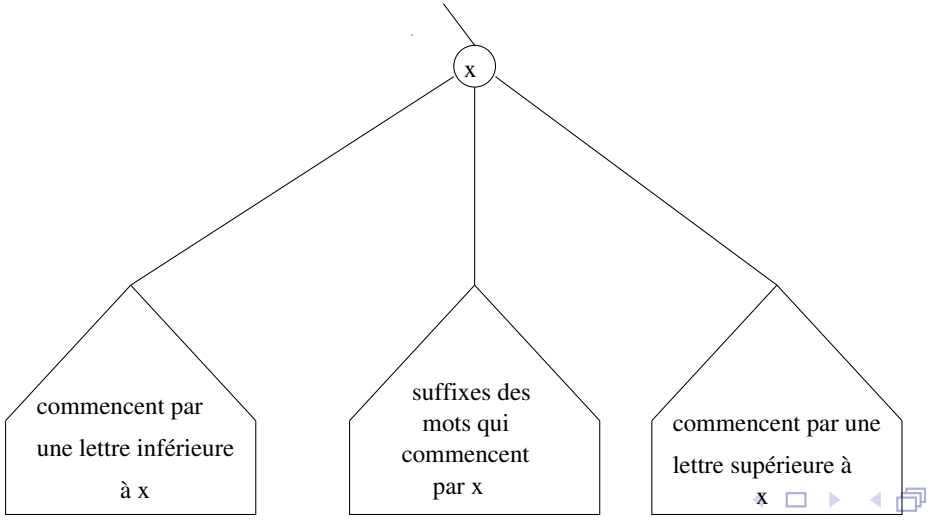
On généralise la notion d'arbre binaire de recherche à un arbre préfixe ternaire :

- TERNAIRE : chaque nœud a au plus trois enfants

  $\rightsquigarrow$ `gauche` $\rightsquigarrow$ `fils` $\rightsquigarrow$ `droit`

- PRÉFIXE :

  - $\rightsquigarrow$ chaque nœud est étiquetté par une lettre.
  - $\rightsquigarrow$ chaque nœud représente un préfixe : la suite des lettres rencontrées sur le chemin de la racine au nœud en empruntant uniquement les nœuds `fils`.



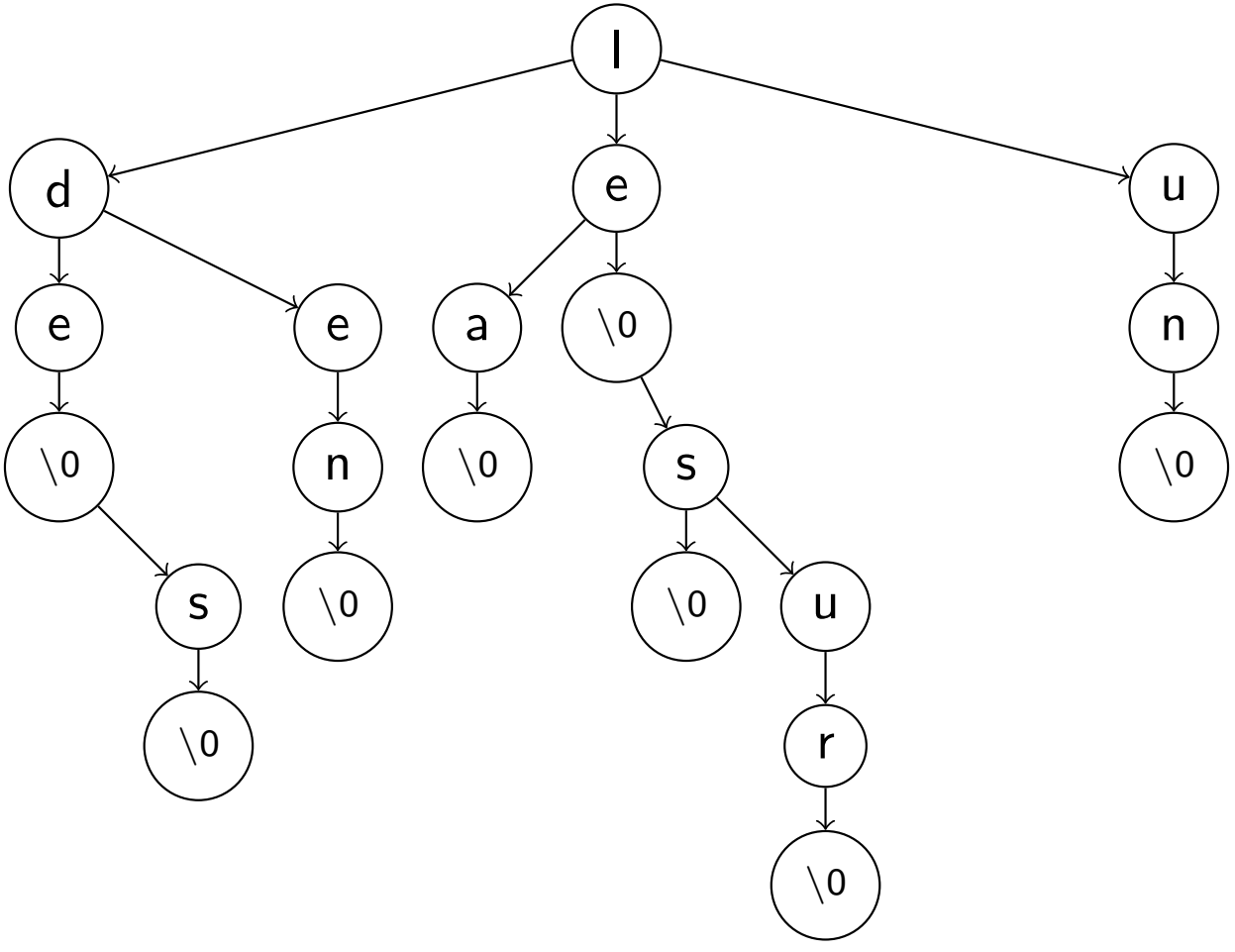
Lecture des préfixe

- La racine représente le préfixe vide
- Si un nœud qui représente le préfixe ω contient la lettre $\mathtt{x}$:
  - le sous arbre `gauche` contient les mots de la forme $\mathtt{a}\upsilon$, avec $\mathtt{a}<\mathtt{x}$
    $\rightsquigarrow$ Ces mots représentent les mots $\omega\mathtt{a}\upsilon$ dans le lexique.
  - le sous arbre `fils` contient les mots de la forme μ
    $\rightsquigarrow$ Ces mots représentent les mots $\omega\mathtt{x}\mu$ dans le lexique
  - le sous arbre `droit` contient les mots de la forme $\mathtt{b}\nu$ avec $\mathtt{b}>\mathtt{x}$
    $\rightsquigarrow$ Ces mots représentent les mots $\omega\mathtt{b}\nu$ dans le lexique,.

Exemple

`Lexique = le, la, les, leur, un, de, des, en,`

$\rightsquigarrow$ Mots ajoutés dans cet ordre dans un arbre initialement vide :



Les lettres liés par la relation *frère* sont à la même position dans les mots.

Sujet de DM pour 2021-2022

Commencer à implémenter les fontionnalités suivantes d'un arbre léxicographique vu comme un arbre ternaire de recherche :

- Recherche d'un mot ;
- Insertion d'un mot ;
- Suppression d'un mot ;
- Affichage du lexique ;
- Libération de l'arbre.

Le reste du sujet arrivera très vite !!