On représente les ensembles d'entiers par des listes. Définir les fonctions de manipulation des ensembles :

- appartient ?
- est inclus ? $A \subseteq B \Leftrightarrow \forall x \in A \Rightarrow x \in B$
- union $A \cup B = \{x / x \in A \vee x \in B\}$
- intersection $A \cap B = \{x / x \in A \wedge x \in B\}$
- différence $A - B = \{x / x \in A \wedge x \notin B\}$

```
(define appartient?
  (lambda (e l)
    (if (null? l)
        #f
        (if (= (car l) e)
            #t
            (appartient? e (cdr l))))))

(define estInclus?
  (lambda (A B)
    (if (null? A)
        #t
        (if (appartient? (car A) l)
            (estInclus? (cdr A) l)
            #f))))
```

```
(define union
  (lambda (E1 E2)
    (if estVide? E1)
        E2
        (if (appartient? (car E1) E2)
            (union (cdr E1) E2)
            (cons (car E1)(union (cdr E1) E2)

(define intersection
  (lambda (A B)
    (if (null? A)
        '()
    (if (null? B)
        '()
        (if (appartient? (car A) B)
            (cons (car A)(intersection (cdr A) B)))
            (intersection (cdr A) B))))))

(define difference
  (lambda (A B)
    (if (null? A) '()
      (if (null? B) (copie (A))
        (if (appartient? (car A) B)
            (difference (cdr A) B)
              (cons (car A) (difference (cdr A) B)))))))
```

Définir une fonction qui construit la liste (1 2 3 ... n) pour n donné.
Même chose pour obtenir (n (n-1) (n-2) ... 1)

```
(define croissant
  (lambda (n)
    (croissant2 n 1)))

(define croissant2
  (lambda (n i)
    (if (<= i n)
        (cons i (croissant2 n (+ i 1)))
        '())))

(define decroissant
  (lambda (n)
    (if (>= n 1)
        (cons n (decroissant (- n 1)))
        '())))
```

Définir une fonction récursive qui renvoie une liste renversée

```
(define renversee
  (lambda (l)
    (if (null? l)
        '()
        (concatenation (renversee (cdr l)) (list (car l))))))
```

pas optimal
on parcourt trop de fois la liste

Pour le rendre optimal, on devrait empiler la liste dans une pile

n n n + 1 . . . d'un arbre

```
(define renversee1
  (lambda (l)
    (renversee2 l '())))

(define renversee2
  (lambda (l p)
    (if (null? l)
        p
        (renversee2 (cdr l)(cons (car l) p)))))
```

Définir une fonction qui calcule toutes les sous listes d'une liste donnée (sans tenir compte de l'ordre)

```
(define sslistes
  (lambda (l)
    (if (null? l)
        '(())
        (concatenation (sslistes (cdr l))(ajout (car l)(sslistes (cdr l)))))))

(define ajout
  (lambda (e l)
    (if (null? l)
        '()
        (cons (cons e (car l))(ajout e (cdr l))))))
```

(let ((s2 (sslistes (cdr l))))
(concatenation s2 (ajout (car l) s2)))

Plus simple :

```
(define ajout
  (lambda (e l)
    (map (lambda (x) (cons e x)) l)))
```

Programmation fonctionnelle TD

Définir une fonction qui calcule la hauteur d'un arbre

```
(define hauteur
  (lambda (a)
    (if (null? a)
        0
        (+ 1 (max (hauteur (SG (A))) (hauteur (SD (A))))))))
```

```
(define SD
  (lambda (A)
    (caddr (A))))
```

```
(define SG
  (lambda (A)
    (cadr (A))))
```

Définir une fonction qui calcule le nb de noeuds d'un arbre

```
(define nbNoeuds
  (lambda (A)
    (if (estVide? A)
        0
        (+ 1 (nbNoeuds (SG (A))) (nbNoeuds (SD (A)))))))
```

Définir une fonction qui calcule le nb de feuilles d'un arbre

```
(define nbFeuilles
  (lambda (A)
    (if (null? A)
        0
        (if (estUneFeuille? (A))
            1
            (+ (nbFeuilles (SG (A))) (nbFeuilles (SD (A))))))))

(define estUneFeuille?
  (lambda (A)
    (and (estVide? (SG (A))) (estVide (SD (A))))))
```

Définir une fonction qui détermine si l'arbre donné en argument est binaire ordonné.

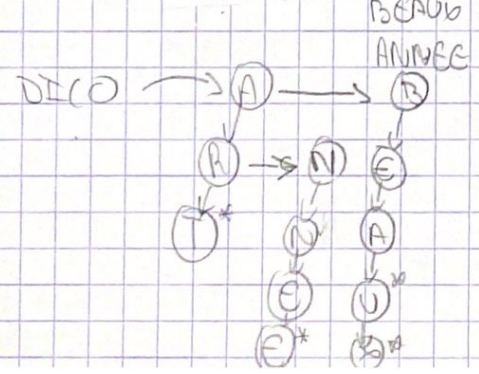
```
(define estOrdonné?
  (lambda (A)
    (if (estVide? (A))
        #t
        (if (estVide? (SG(A)))
            (if (estVide? (SD(A)))
                #t
                (if (estOrdonné? (SD(A)))
                    (> (min (SD(A)) (valeur (A))))
                    #f))
            (if (estOrdonné? (SG(A)))
                (if (<= (max (SG(A)) (valeur (A))))
                    (if (estVide? (SD(A)))
                        #t
                        (if (estOrdonné? (SD(A)))
                            (> (min (SD(A)) (valeur (A))))
                            #f)))))))))
```

```
(define min
  (lambda (A)
    (if (not (null? (SG(A))))
        (min (SG(A)))
        (valeur (A)))))

(define max
  (lambda (A)
    (if (not (null? (SD(A))))
        (max (SD(A)))
        (valeur (A)))))
```

Définir la fonction present? qui détermine si un mot M est présent dans un dictionnaire DICO

```
(define marqueur
  (lambda (A)
    (caddr(A))))

(define valeur
  (lambda (A)
    (car (A))))

(define frere
  (lambda (A)
    (caddr(A))))

(define fils
  (lambda (A)
    (cadr(A))))
```

(Petite fils frere marqueur)



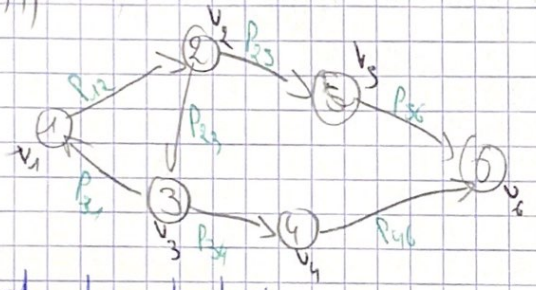
```
(define creationNoeud
  (lambda (lettre fils frere marqueur)
    (list (lettre fils frere marqueur)))

(define copieDICO
  (lambda (D)
   (if (null? D)
     '()
     (creationNoeud (valeur D) (copieDico (fils D)) (copieDico (frere D)) (marqueur D)))))

(define present?
  (lambda (M DICO)
    (if (null? DICO)                    TESTER L'ALGO
      #f
      (if (null? M)
        #f
        (if (marqueur DICO)
          (if (null? (cdr M))
            (if (eq? (car M) (valeur DICO))
              #t
              (present? (M (frere DICO))))
            (if (eq? (car M) (valeur DICO))
              (present? (cdr M) (fils DICO))
              (present? (M (frere DICO)))))
          (if (eq? (car M) (valeur DICO))
            (present? (cdr M) (fils DICO))
            (present? (M (frere DICO))))))))
```

Définir une fonction qui ajoute un mot dans un DICO

MOT → M(0 (T')(#t)(#)(#))

```
(define ajout
  (lambda (M DICO)
    (if (null? DICO)
      (if (null? M)
        '()
        (creationNoeud (car M)(ajout (cdr M)'())'()(null? (cdr M))))
      (if (eq? (car M)(valeur DICO))
        (creationNoeud (valeur DICO)(ajout (cdr M)(fils DICO))(copieDico (frere DICO))
                       (or (marque DICO)(null? (cdr M))))
        (creationNoeud (valeur DICO)(copieDico (fils DICO))(ajout M (frere DICO))(marque DICO)))))
```

Graphes :



1) Proposer une structure de données
2) Définir une fonction qui renvoie la liste des successeurs directs dans un graphe donné G pour un noeud connu par son nom.
3) " de tous les successeurs "
4) Définir une fonction qui détermine s'il existe un chemin entre deux noeuds.

1) G=((1 (2))
(2 (3 5))
(3 (1 4))
(4 (6))
(5 (6))
(6 ()))

2) (define successeurD
(lambda (n G)
(if (null? G)
'()
(if (eq? n (car (car G)))
(cadar G)
(successeurD n (cdr G))))))