

## ALGO -

1) Concevoir deux algo, l'un itératif l'autre récursif qui retournent la longueur d'une liste simplement chaînée (LSC)

INTRO

Concevoir un algo qui affiche le contenu d'une liste

Afficher ( $l$ : liste)

donnée :  $l$ ; la liste à afficher

Début

Tant que  $l \neq \emptyset$

afficher  $l \rightarrow \text{info}$

$l \leftarrow l \rightarrow \text{succ}$

fin tant que

Fin

Comparaison avec tableau.

Afficher ( $t$ : tableau de taille  $n$ )

donnée :  $t$ , tableau de taille  $n$

var :  $i$ , entier compteur

Début

$i \leftarrow 0$

Tant que  $i < n$  faire

Afficher  $t[i]$

$i \leftarrow i + 1$

fin tant que

Fin

1) size-list-it ( $l$ : liste); entrée

donnée  $l$ , entier, liste à étudier

var  $x$ , entier compteur

retourner la longueur de la liste

Début

$x \leftarrow 0$

Tant que  $l \neq \emptyset$  faire

$l \leftarrow l \rightarrow \text{succ}$

$x \leftarrow x + 1$

fin tant que

retourner  $x$

Fin

size-list-rec ( $l$ : liste); entrée

donnée :  $l$ , liste à mesurer

Début

si  $l = \emptyset$  alors retourner 0

retourner  $1 + \text{size-list-rec} (l \rightarrow \text{succ})$

Fin

2) Concevoir un algo qui retourne le nbr d'éléments d'une LSC  
l donné, compris dans un intervalle  $[a, b]$  donné

ex: illustratif  
 $l \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ$   
 $a = 2 \quad b = 3 \quad l$ 'algo retourne 2.

$N(l: \text{liste}, a, b: T)$ : entier (naturel)

donnée:  $l$ : la liste

$a, b$ : les bornes inf et sup de l'intervalle

Variable:  $n$ , un compteur (entier)

Début

si  $l = \circ$  alors retourner 0

si  $a > b$  alors retourner 0

Tant que  $l \neq \circ$  faire

    si  $a \leq l \rightarrow \text{info}$  et  $l \rightarrow \text{info} \leq b$  alors  $n \leftarrow n + 1$

$l \leftarrow l \rightarrow \text{succ}$

Fintant que

retourner  $n$

Tjrs utilisé

$N(l: \text{liste}, a, b: T)$ : entier

donnée:  $l$ : la liste

$a, b$ : les bornes inf et sup de l'intervalle

Déb

si  $a > b$  alors retourner 0

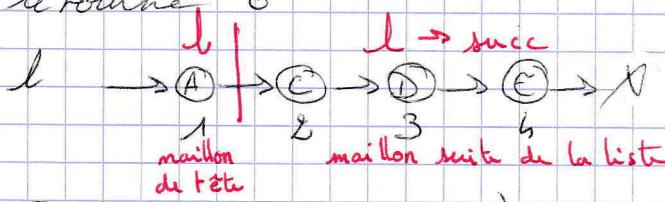
si  $l = \circ$  alors retourner 0

si  $a \leq l \rightarrow \text{info}$  et  $l \rightarrow \text{info} \leq b$  alors retourner  $1 + N$

    retourner  $0 + N(l \rightarrow \text{succ}, a, b)$  ( $l \rightarrow \text{succ}$ ,  $a, b$ )

3) Parcourir une liste jusqu'à un certain point -

Concevoir 2 algos, un récursif / un itératif  
qui retourne la position d'un élément donné en argument,  
dans une liste - Si l'élément n'est pas dans la liste, l'algo  
retourne 0



Exemples. Position  $(l, D) \rightarrow 3$

Position  $(l, F) \rightarrow 0$

Position  $(l, A) \rightarrow 1$

$\text{Position}_it(l: \text{liste}, x: T)$ : entier positif

donnée:  $l$ , la liste ;  $x$ , la clé

Var:  $p$ , entier, compteur qui représente la position courante

Début

$p \leftarrow 0$

Tant que  $l \neq \circ$  faire

    si  $l \rightarrow \text{info} = x$  retourner  $p$

$l \leftarrow l \rightarrow \text{succ}$

$p \leftarrow p + 1$

Fintant que

retourner 0

De l'algo récursif à l'algo itératif  
on gagne une boucle et une variable

Position\_réc ( $l$ : liste,  $x$ : T) : entier positif  
donnée :  $l$ , la liste ;  $x$ , la clé  
Var:  $p$ , position de  $x$  dans  $l \rightarrow \text{succ}$

Début

Si  $l \neq \emptyset$  alors retourner 0  
Si  $l \rightarrow \text{info} = x$  alors retourner 1 maillon de tête  
 $p \leftarrow \text{Position}(l \rightarrow \text{succ}, x)$  récuratif  
Si  $p = 0$  alors retourner 0  
[sinon] retourner  $p + 1$  s'est nul part

Bifz ex1 -

lire voir un algo itératif qui retourne la somme d'un élément sur 2 en partant de la tête de liste -  
Retourner 0 si la liste est vide

Somme\_1-2 ( $l$ : liste) : entier  
donnée :  $l$ , la liste  
Var:  $s$ , la somme  
 $i$ , compteur

Déb si  $l \neq \emptyset$  alors retourner 0

$s \leftarrow 0$

$i \leftarrow 0$

Tant que  $l \neq \emptyset$  faire

si  $i$  est pair alors  $s \leftarrow s + l \rightarrow \text{info}$

$l \leftarrow l \rightarrow \text{succ}$

$i \leftarrow i + 1$ .

Fin tant que  
retourner  $s$

Fin

⊕ Performant

Somme\_1-2 ( $l$ : liste) : entier  
donnée :  $l$ , la liste  
Var:  $s$ , la somme

Déb

$s \leftarrow 0$

Tant que  $l \neq \emptyset$  faire

$s \leftarrow s + l \rightarrow \text{info}$

$l \leftarrow l \rightarrow \text{succ}$

si  $l \neq \emptyset$  alors  $l \leftarrow l \rightarrow \text{succ}$

Fin tant que

retourner  $s$

Fin

## Blitz exo 2

Concevoir un algorithme qui retourne la copie conforme d'une liste.

DUPL(l: liste): liste.

donnée : l, liste source

var : d, référence sur la liste dupliquée

q, référence sur la queue début de la liste dupliquée

Déb

si  $l = \emptyset$  alors retourner  $\emptyset$

$q \leftarrow d \leftarrow \text{réserver faillir}$

$d \rightarrow \text{info} \leftarrow l \rightarrow \text{info}$

$l \leftarrow l \rightarrow \text{succ}$

Tant que  $l \neq \emptyset$  faire :

$q \rightarrow \text{succ} \leftarrow \text{réserver faillir}$

$q \leftarrow q \rightarrow \text{succ}$

$q \rightarrow \text{info} \leftarrow l \rightarrow \text{info}$

$l \leftarrow l \rightarrow \text{succ}$

fin tant que

$q \rightarrow \text{succ} \leftarrow \emptyset$

retourner d

q permet d'effectuer une réservation de mémoire au tableau suivant

Fin

Concevoir d'algo, l'un if - rec qui permet en argument d listes ordonnées l<sub>1</sub>, l<sub>2</sub> et l<sub>3</sub> et qui les fusionnent en une nouvelle liste ordonnée et la retournent.

Important : aucun mailon n'est créé, aucun détruit, ils sont réutilisés pour construire la liste fusionnée

Fusion(l<sub>1</sub>, l<sub>2</sub>: liste): liste.

donnée : l<sub>1</sub>, l<sub>2</sub>: liste

var : l, tête de la liste fusionnée

q, queue de la liste fusionnée

Début

si  $l_1 = \emptyset$  retourner l<sub>2</sub>

si  $l_2 = \emptyset$  retourner l<sub>1</sub>

si  $l_1 \rightarrow \text{info} < l_2 \rightarrow \text{info}$  alors  $l \leftarrow l_1$   $l_1 \leftarrow l_1 \rightarrow \text{succ}$

q  $\leftarrow l_1$  sinon  $l \leftarrow l_2$   $l_2 \leftarrow l_2 \rightarrow \text{succ}$

Tant que  $l_1 \neq \emptyset$  et  $l_2 \neq \emptyset$

si  $l_1 \rightarrow \text{info} < l_2 \rightarrow \text{info}$  alors

$q \rightarrow \text{succ} \leftarrow l_1$

$l_1 \leftarrow l_1 \rightarrow \text{succ}$

sinon  $q \rightarrow \text{succ} \leftarrow l_2$

$l_2 \leftarrow l_2 \rightarrow \text{succ}$

$q \leftarrow q \rightarrow \text{succ}$

fin if q

si  $l_1 = \emptyset$  alors  $q \rightarrow \text{succ} \leftarrow l_2$

sinon  $q \rightarrow \text{succ} \leftarrow l_1$

retourner l

Fin

Fusion rec (l<sub>1</sub>, l<sub>2</sub>: list): list

Début

```
si l1 ≠ Ø alors retourner l1
si l2 ≠ Ø alors retourner l2
si l1 → info < l2 → info alors
    l ← l1
    l ← l → succ
    sinon
        l ← l2
        l ← l → succ
        l → succ ← Fusion rec (l1, l2)
    retourner l
```

Fin

Concevoir un algo it-rec qui inverse une liste simplement chaînée

Reverse (l: liste)

données : l, donner modifier

Début si l = Ø ou l → succ ≠ Ø alors retourner

```
    pre ← Ø
    curr ← l
    succ ← l → succ
    Tq succ ≠ Ø faire
        curr ← succ ← pre
        pre ← curr
        curr ← succ
        succ ← succ → succ
```

fin Tq  
l ← curr

Fin

Bzj Concevoir un algo qui retourne le nbr d'éléments pairs dans un AB<sup>"a</sup> d'enfants  $\Rightarrow$  AB - arbre binaire prototypé imposé

NbPairs (a; AB); entier

Déb si a ≠ Ø alors retourner 0

si a → info est pair alors

retourner 1 + NbPairs (a → sag) + NbPairs (a → sagd)

sinon retourner 0 + NbPairs (a → sag) + NbPairs (a → sagd)

Fin

Bzj Conce

Bzjz concevoir un algo qui transforme une liste non vide en déplacement en position de tête de liste le maillon de queue -

ex avt  $l \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow x$   
ap  $l \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow x$

Tête Queue (l: liste)

donnée modif: l, la liste à modifier

nouveau élément

compte entier naturel // num ciblé par mi

Déb

nouveau  $\leftarrow$  réservé paillot  $\times$   
nouveau  $\rightarrow$  valeur  $\leftarrow$  valeur  
nouveau  $\rightarrow$  suivant  $\leftarrow \emptyset$

pas fondu à la mémoire

si place = 0 alors

nouveau  $\rightarrow$  suivant  $\leftarrow 1$

1  $\leftarrow$  nouveau

sinon mi  $\leftarrow$  taille liste

compte  $\leftarrow 1$

Tant que (compte < place)  $\wedge$  (mi  $\neq \emptyset$ ) faire

compte  $\leftarrow$  compte + 1

mi  $\leftarrow$  mi  $\rightarrow$  suivant

Fina

nouveau  $\rightarrow$  suivant  $\leftarrow$  mi  $\rightarrow$  suivant

mi  $\rightarrow$  suivant  $\leftarrow$  nouveau

Fin si

reformuler l.

Tête Queue (l: liste)

donnée modif: l, la liste

Variante: t, la tête de la liste

p le précédent

q la queue

Début

t  $\leftarrow$  l

Tant que l  $\rightarrow$  succ  $\neq \emptyset$

p  $\leftarrow$  l

l  $\leftarrow$  l  $\rightarrow$  succ

Fina

q  $\leftarrow$  l

q  $\rightarrow$  succ  $\leftarrow$  t

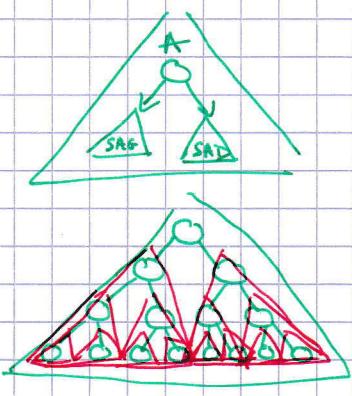
p  $\rightarrow$  succ  $\leftarrow \emptyset$

Fin

## ARBRE

Concevoir un algo pour compter sur un AB.

- A) le nbr d'éléments
- B) le nbr de feuilles.
- C) le nbr de noeuds internes
- D) le nbr de noeuds qui n'ont qu'un enfant.



### A) Nbr d'éléments

$N(a: AB)$ : entier  
donnée: a, autre @Sad (sous arbre droit) sag (sous arbre gauche)

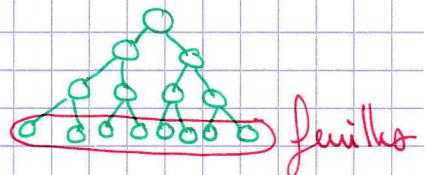
Debut

    |  
    si  $a = \emptyset$  alors retourner 0  
    retourner  $1 + N(a \rightarrow sag) + N(a \rightarrow sad)$

Fin

### B) Nbr de feuilles

$F(a: AB)$ : entier  
donnée: a, autre



Debut

    |  
    si  $a = \emptyset$  alors retourner 0  
    si  $a \rightarrow sag = \emptyset$  et  $a \rightarrow sad = \emptyset$  retourner 1.  
    retourner  $F(a \rightarrow sag) + F(a \rightarrow sad)$

        → arbre vide  
        → arbre singleton  
        noeud racine, 1 feuille

Fin

### C) Nbr de noeuds internes

$I(a: AB)$ : entier

        → noeud racine est noeud interne  
        que si il est seul.  
        → noeud interne ssi  $a \rightarrow sag \neq \emptyset$  et  
 $a \rightarrow sad \neq \emptyset$

Debut

    |  
    si  $a = \emptyset$  alors retourner 0  
    si  $a \rightarrow sag \neq \emptyset$  et  $a \rightarrow sad \neq \emptyset$   
    alors retourner  $1 + I(a \rightarrow sag) + I(a \rightarrow sad)$   
    sinon  
    retourner  $0 + I(a \rightarrow sag) + I(a \rightarrow sad)$

### D) Nbr de noeud qui n'ont qu'un enfant

$PE(a: AB)$ : entier

        → noeud racine  
        est monoséquentaire  
(1 seul enfant)

Debut

    |  
    si  $a = \emptyset$  alors retourner 0  
    si  $(a \rightarrow sag = \emptyset \text{ et } a \rightarrow sad \neq \emptyset)$  ou  $(a \rightarrow sag \neq \emptyset \text{ et } a \rightarrow sad = \emptyset)$   
    alors retourner  $0 + PE(a \rightarrow sag) + PE(a \rightarrow sad)$

Fin

# Blitz

Exo - Concevoir un algo qui retourne la profondeur de la feuille la plus profonde d'un AB, -1 si l'AB est vide -

$P(a: AB)$ : enlever

donnée : a, AB

var : g, d, profondeurs des plus profondes feuilles separé à gauche et à droite

Début

si  $a = \emptyset$  alors retourner 1

si  $a \rightarrow sag = \emptyset$  et  $a \rightarrow sad = \emptyset$  alors retourner 0

fin si

$d \leftarrow P(a \rightarrow sad)$

$g \leftarrow P(a \rightarrow sag)$

si  $g > d$  alors retourner  $1 + g$

sinon retourner  $1 + d$ .

Fin

Exo - Concevoir un algo qui supprime toutes les feuilles d'un AB (sans supprimer les parents qui détiennent feuilles après suppression de leurs enfants) -

→ Si l'arbre est vide rien à faire

→ Si arbre une feuille

$Supp(a: AB)$

donnée modifiée : a, AB

Début si  $a = \emptyset$  alors retourner

si  $a \rightarrow sag = \emptyset$  et  $a \rightarrow sad = \emptyset$  alors

libérer a

$a \leftarrow \emptyset$

sinon

$Supp(a \rightarrow sag)$

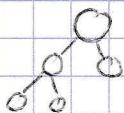
$Supp(a \rightarrow sad)$

fin

Fin

Exo - Concevoir un algo itératif qui retourne le nbr d'élément d'un AB situés à une profondeur p donnée.

$N(a, p)$



$$N(a, 0) = 1$$

$$N(a, 1) = 2$$

$$N(a, 2) = 2$$

$$N(a, 3) = 0$$

# Exemple pour la donnée modifiée

Je veux insérer une nouvelle valeur en tête de liste

Insertfront(l: liste; x: T)

variable: m, nouveau maillon de tête  
donnée modifiée: l, la liste

Début

$m \leftarrow \text{reserver}(\text{taille})$   
 $m \rightarrow \text{info} \leftarrow x$   
 $m \rightarrow \text{succ} \leftarrow l$   
 $l \leftarrow m$

Fin

→ LANGUAGE C.

void insertfront (list \*pl, int x)  
{  
 list m = malloc (sizeof (node));  
 m → info = x;  
 m → succ = \*pl;  
 \*pl = m;

```
typedef struct node
{
    int info;
    struct node *succ;
} node;
typedef node* list;
```

Algo récursif qui inverse une liste

Reverse (l: liste)

donnée modifiée: l, list.

var: q

Début

Si  $l = \emptyset$  ou  $l \rightarrow \text{succ} = \emptyset$  alors ret 0  
 $q \leftarrow l \rightarrow \text{succ}$   
reverse ( $l \rightarrow \text{succ}$ )  
 $q \rightarrow \text{succ} \leftarrow l$   
 $l \leftarrow l \rightarrow \text{succ}$   
 $q \rightarrow \text{succ} \rightarrow \text{succ} \leftarrow \emptyset$

Fin

○ → ○ → ○ → ○

EN C

```
void reverse (list *pl)
{
    if (!(*l & l → succ)) return;
    list q = (*pl) → succ;
    reserver (& (*pl) → succ);
    q → succ = *pl;
    *pl = (*pl) → succ;
    q → succ → succ = NULL;
```

Concevoir un algo qui retourne l'adresse du point de confluence de 2 listes li et lz si elles sont, & non

- $\Rightarrow$  A Trouver une solution naïve en  $O(n^2)$   
 B Trouver une solution + intelligente en  $O(n)$

Confluent (li, lz : listes) liste  
 Données: li, lz : les 2 listes  
 Var : h, la tête de lz

Début

si li =  $\emptyset$  ou lz =  $\emptyset$  alors retourner  $\emptyset$   
 $h \leftarrow lz$

Tant que li  $\neq \emptyset$  faire

Tant que lz  $\neq \emptyset$  faire

| si li = lz alors retourner li  
| lz  $\leftarrow lz \rightarrow \text{succ}$

fin

| li  $\leftarrow la \rightarrow \text{succ}$   
| lz  $\leftarrow h$

fin

retourner  $\emptyset$

Fin

Algo auxiliaire

longueur (l, liste) : entier  
 données l, liste

Var : x, variable -

Déb:

| si (l =  $\emptyset$ ) retourner 0  
| x  $\leftarrow 0$

| tant que l  $\neq \emptyset$  faire

| | x  $\leftarrow x + 1$

| | l  $\leftarrow l \rightarrow \text{succ}$

| fin tant que  
| retourner x

Fin

Algo pré-itératif (l, liste, x, entier)

donnée modifiable liste

Déb:

| si l =  $\emptyset$  retourner

| Tant que (l  $\neq \emptyset$ ) et (x  $> 0$ ) faire

| | l  $\leftarrow l \rightarrow \text{succ}$

| | x  $\leftarrow x - 1$

| fin tant que

Fin

Algo principal

Confluence (li, lz ; liste) liste

Données li, lz, liste

Var : P1, P2, entiers

Déb si li =  $\emptyset$  ou lz =  $\emptyset$  retourner

| P1  $\leftarrow$  longueur (li)

| P2  $\leftarrow$  longueur (lz)

| Si P1 > P2 alors pré-itérer (P1, P1 - P2)

| Sinon Pré-itérer (P2, P2 - P1)

| Tant que li  $\neq \emptyset$  et lz  $\neq \emptyset$  faire

| | si li = lz, retourner li

| | | li  $\leftarrow li \rightarrow \text{succ}$

| | | lz  $\leftarrow lz \rightarrow \text{succ}$

| fin tant que

Fin