



SDA TD

Exercices

TD 1

1. Gestion d'un « stock » de marchandises nommées par une « référence » dans une certaine « quantité » pour lesquelles on souhaite pouvoir ajouter, supprimer, réajuster ladite quantité. Les références peuvent être nouvelles ou être supprimées. (max. 1000 références).
lors de l'inventaire

Spéc STOCK étend BASE

sous STOCK

opérations

gén.
de
vare (créer_stock : _ → Stock

new_ref : Stock Réf Nat → Stock

erase_ref : Stock Réf → Stock

ajout_qty : Stock Réf Nat → Stock

retrait_qty : " "

reajust_qty : " "

exist_ref : Stock Réf → bool

quant_ref : Stock Réf → Nat

stockvide : Stock → bool

nombre_ref : Stock → Nat

préconditions s; stock; R : référence i : q Nat

tnes

pré : new_ref (s, r) = exist_ref (s, r) ∧ nombre_ref < 1000

pré : erase_ref (s, r) = exist_ref (s, r)

pré : quant_ref (s, r) = exist_ref (s, r)

pré : ajout_qty (s, r, q) = exist_ref (s, r)

axiomes s; stock; R : référence i q1, q2, q3 : Nat

erase_ref (new_ref (s, r1, q), r2) si r1 = r2 alors s

sinon new_ref (erase_ref (s, r2), r1, q) fsi

quant_ref (new_ref (s, r1, q), r2) = si r1 = r2 alors q

sinon quant_ref (s, r2) fsi

ajout_qty (new_ref (s, r1, q1), r2, q2) = si r1 = r2

alors new_ref (s, r1, q1 + q2)

sinon new_ref (ajout_qty (s, r2, q2), r1, q1)

fsi

exist-ref(créerstock(), r) = FAux

exist-ref(new_ref(S, R₁, Q₁), R) = R == R₁ \vee exist-ref(S, R)

nombre-ref(créerstock()) = 0

nombre-ref(new_ref(S, R₁, Q₁)) = 1 + nombre-ref(S)

Programmation

Fonctionnelle `typedef struct { Nat tab[1000]; ... } Stock;`

Mutation `typedef Nat stock[1000];`

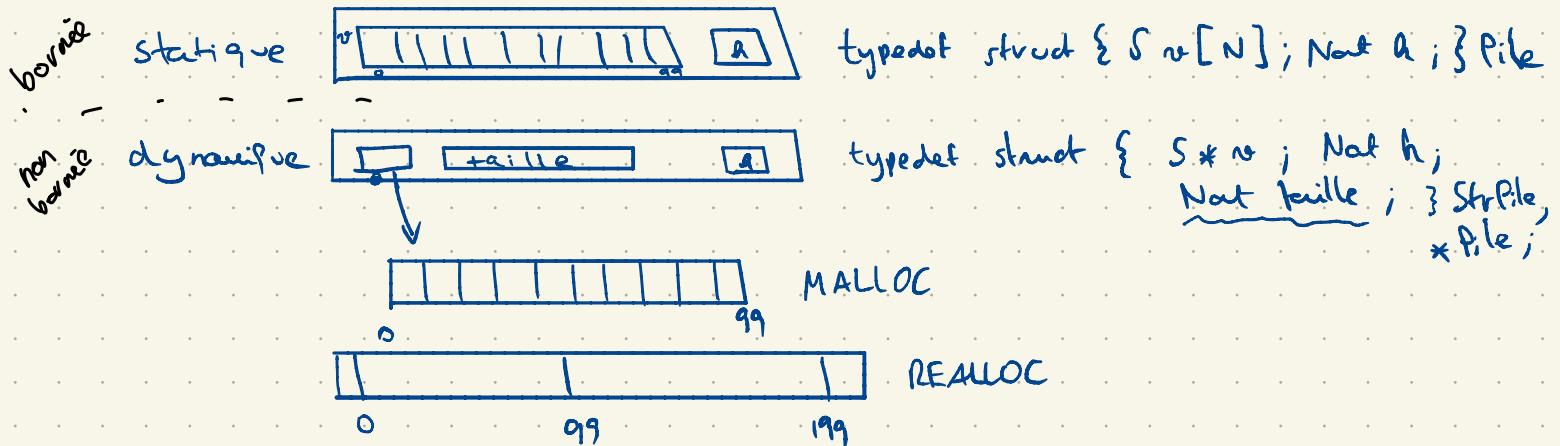
rappel tableau, passage par adresse

TP 2

24-09-19

Tableaux dynamique

piles non bornées avec des tableaux dynamiques



→ reécrire l'implantation des piles par mutation.

`typedef struct { S *v; Nat h; Nat taille; } StrPile, *Pile; /* version 3 */`

`#define TAILLE 100`

```
Pile pilenouv()
{
    Pile p;
    p.h=0;
    p.taille = TAILLE;
    p.v = malloc = MALLOCN( S, TAILLE );
    return p;
}
```

```
Pile pilenouv() /* version 3 */
{
    Pile p = MALLOC (StrPile);
    p->h = 0;
    p->taille = TAILLE;
    p->v = MALLOCN( S, TAILLE );
    return p;
}
```

```

#define REALLOC (p,t,n) ( (t*) realloc (p, n*sizeof(t));

Pile empiler ( Pile S, S x) /* version 3 */
{
    S *temp = p->v;
    if (( p->h )==( p->taille ))
    {
        p->v = REALLOC ( p->v, S, p->taille+TAILLE);
        if ( p->v=NULL)
        {
            printf("Complet");
            p->v = temp;
        }
        p->taille = p->taille+TAILLE
        p->v[p->h] = x;
        p->h++;
    }
    else
    {
        p->v[p->h] = x;
        p->h++;
    }
    return p;
}

```

Comparatif de 3 SD pour un même objet.

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n \quad n = \text{degré}$$

$$d(p) = a_1 + 2a_2 x + \dots + n a_n x^{n-1} \quad a_0, \dots, a_n \text{ réels}$$

• SD polynomial

degré ?

1 ou 2 opérations

< poly nomme dérivé

```
#define Reel polynome1[N]; // SD tableau des coeff a0a1a2...
```

```
typedef struct // SD2 structure avec un tab. statique des coeff.
{
    Nat degre;
    Reel coef[N+1];
} polynome2;
```

```
typedef struct // SD3 structure avec un tab. dynamique des coeff.
{
    Nat degre;
    Reel *coef;
    Nat taille;
} polynome3;
```

```
/* SD1 = Tableau */
```

```
Nat degre (Polynome1 p)
{
    Nat deg,i;
    for (i=0; i<N; i++)
        if ( P[i] != 0)
            deg=1
    return deg;
}
```

En C, passage des arguments par valeur.

Copie du tableau P,
En C, copie de l'adresse

Polynôme dérivé (polynôme P')

```
/* Définir le polynôme d sans détruire le polynôme p (récursivité fonctionnelle) */
```

```
{
    Nat i = N;
    while (i>=0 && (p[i] == 0))
        i = i-1;
    return i;
}
```

8. oct. 19.

→ Reprise cours 1.3 // (chapitre 1) (p. 10).

À faire parcours de gauche à droite

récursif avec app1 et itératif
idem pour la fonction minimum.

- spéciat. - récursive - récurs. de droite à gauche.
(version itérative faite en cours).

```
Bool app1 (Ens e, Nat x, Nat h)
{
    if (h == e.n) return faux;
    else if (e.v[h] == x) return vrai;
    else return app1 (e, x, h+1);
}
```

```
Bool app (Ens e, Nat x)
return app1 (e, x, 0)
```

Bool app (Ens e, Nat x) {

Nat K;

```
for (K=0; e.n > K && e.v[K] != x; K++)  
    return (K == e.n);  
}
```

recherche de la position d'un élément r: Ens Nat → Nat ou est

position: 1 à n / pas trouvé / ex0 chez vous

position: 0 à n-1 / pas trouvé -1 // maintenant.

① spécification par rapport aux générateurs de base

② recopie de la spécf. en C: version récursive.

③ version itérative.

$$r(\emptyset, x) = -1$$

$$r((e, x), y) =$$

...

Ent r(Ens e, Nat y)

```
{ if (e.n == 0)  
    return -1;  
else if (e.v[e.n-1] == y)  
    return e.n-1;  
else {  
    e.n--;  
    return r(e, y);  
}
```

Ent r(Ens e, Nat y)

~~```
{ for (int i=0; e.v[i] != y; i++)
 if (e.v[i] == y)
 return i+1;
else
```~~

Ent r(Ens e, Nat x)

```
{ Ent K = e.n - 1
 while (K >= 0) {
 if (e.v[K] == x) return K;
 K ...;
 }
 return -1;
```

Ent K:

```
for (K=e.n-1; K >= 0 &&
 e.v[K] != y; K--);
return K;
```



## Spec MULTIENS (EG) étend BASE

Sorte Multiens

### Opérations

|                                                                  |                                                    |
|------------------------------------------------------------------|----------------------------------------------------|
| $\emptyset : \_ \rightarrow \text{Multiens}$                     |                                                    |
| $i : \text{Multiens } S \rightarrow \text{Multiens}$             |                                                    |
| $\wedge : \text{Multien } S \rightarrow \text{Multien}$          | /* suppression d'une occurrence */                 |
| $s' : \text{Multien } S \rightarrow \text{Multien}$              | /* supprime toutes les occurrences d'un élément */ |
| $s^n : \text{Multien } S \text{ Nat} \rightarrow \text{Multien}$ | /* supprime n occurrences */                       |
| $v : \text{Multien} \rightarrow \text{Bool}$                     |                                                    |
| $e : S \text{ multien} \rightarrow \text{Bool}$                  |                                                    |
| $\circ : \text{Multien } S \rightarrow \text{Nat}$               | /* nombre d'occurrences */                         |
| $ _1 : \text{Multien} \rightarrow \text{Nat}$                    | /* cardinal du multien ensemble */                 |
| $c : \text{Multien} \rightarrow \text{Nat}$                      | /* nombre d'éléments distincts */                  |

### pas de préconditions

axiomes :  $e : \text{Ens}; x, y : S; n : \text{Nat}$

$$(e0) i(i(e, x), y) = i(i(e, y), x) \quad /* \text{pas d'ordre */}$$

$$(e1) \circ(i(e, x), y) = \begin{cases} \infty & \text{si } x == y \\ 0 & \text{sinon} \end{cases} + \circ(e, y)$$

$$(e2) c(i(e, x), y) = \begin{cases} \infty & \text{si } e == \text{false} \\ c(e) + 1 & \text{sinon} \end{cases}$$

$$s'(i(e, x), y) = \begin{cases} v(e) & \text{si } e == \text{true} \\ s'(x, y) & \text{sinon} \\ s'(e, y) & \text{si } x == y \end{cases}$$

$$(e3) s'(\emptyset, x) = \emptyset$$

$$(e4) s'(i(e, y), x) = \begin{cases} s'(e, x) & \text{si } x == y \\ i(s'(e, x), y) & \text{sinon} \end{cases}$$

$$(e5) s''(\emptyset, x, n) = \emptyset$$

$$(e6) s''(i(e, y), x, n) = \begin{cases} \emptyset & \text{si } n == \emptyset \\ s''(e, x, n - 1) & \text{sinon si } x == y \\ i(s''(e, x, n - 1), y) & \text{sinon} \\ \emptyset & \text{si } e == \text{false} \end{cases}$$

Specification.

Spec CPILES (EG) étend PILE

sorte Cpiles

Opérations

x :  $\text{Cpiles}$  /\* couple de piles vides \*/

$\rightarrow (-, -)$  : Pile Pile  $\rightarrow$  Cpiles /\* couple de piles créé à partir de 2 piles \*/

e : Cpiles Nat S  $\rightarrow$  Cpiles /\* empilement sur une des 2 piles \*/

d : Cpiles Nat  $\rightarrow$  Cpiles /\* dépiler une des deux piles \*/

et.vide ev : Cpiles  $\rightarrow$  Bool  $\begin{cases} \text{vrai} & \text{si les 2 piles sont vides} \\ \text{faux} & \text{sinon} \end{cases}$  n : Cpiles  $\rightarrow$  Nat

somme s : Cpiles, Nat  $\rightarrow$  S

hauteur h : Cpiles, Nat  $\rightarrow$  Nat  $\begin{cases} \text{hauteur une pile de l'atm} & \text{si une pile est vide} \\ \text{hauteur une pile de l'atm} & \text{sinon} \end{cases}$  somme : Cpiles, Nat  $\rightarrow$  S

préconditions x : S cp : Cpiles i : Nat

p1 (p1, p2) = hauteur (p1) + hauteur (p2)  $\leq N$

p2 2 .. . droit.

p1 h ((p1, p2), i) =  $i \in \{1, 2\}$

p1 ev ((p1, p2)), i) =  $x \in 4 \cup x \in 2$ .

axiomes cp : Cpiles ; p1, p2 : Pile ; i : Nat , x, y : S

1)  $\text{v}( ) = (\text{pilev}, \text{pilev})$

2)  $(e((p1, p2), 1, x)) = (\text{empiler}(p1, x), p2)$

3)  $e((p1, p2), 2, x) =$

4)  $d((p1, p2), b) =$

si b == 1 alors  $(\text{dépiler}(p1), p2)$

sinon si b == 2 alors  $(p2, \text{dépiler}(p1))$

sinon  $(p1, p2)$  fini

5)  $s((p1, p2), b) = \text{si } b == 1 \text{ alors somme}(p1)$

sinon somme(p2)

6)  $n((p1, p2)) = \text{hauteur}(p1) + \text{hauteur}(p2)$

7)  $\text{somme}((p1, p2), 1) = \text{si vider}(p1) \text{ alors } \emptyset$ .

sinon somme( $(p1, p2)$ , 1) fini

8)  $\text{h}((p1, p2), i) = \text{si } i == 1 \text{ alors hauteur}(p1)$

sinon hauteur(p2) fini.

9)  $\text{ev}((p1, p2), z) = \text{si } x == 1 \text{ alors ev}(p1) \text{ sinon ev}(p2)$ .

Prog par mutation en utilisant une structure avec un tableau.

by def struct {

    Nat  $\alpha[N]$ ;  
    Nat  $h_1$ ;  
    Nat  $h_2$ ;  
}

    StrPiles;

alternative.

$\Rightarrow \{ \times C\text{piles};$

$C\text{piles} . \text{new}()$

{

$C\text{piles } cp = \text{MALLOC } (\text{Str} C\text{pile})$ ;  
     $cp \rightarrow h_1 = 0$ ;    $cp \rightarrow h_2 = 0$ ;  
    return  $cp$ ;

}

$C\text{piles } e (C\text{piles } cp, \text{Nat } n, S \_x)$  {

if ( $n == 1$ ) {

$cp \rightarrow v [cp \rightarrow h_1] = \infty$ ;  
     $cp \rightarrow h_1 ++$ ;

}

else {

$cp \rightarrow v [N - 1 - cp \rightarrow h_2] = \emptyset$ ;  
     $cp \rightarrow h_2 ++$ ;

}

return  $cp$ ;

$C\text{piles} d (C\text{pile } cp, \text{Nat } n)$  {

if ( $n == 1$ )

$cp \rightarrow h_1 --$ ;

if ( $n == 2$ )

$cp \rightarrow h_2 --$ ;

return  $cp$ ;

}

$S . s (C\text{piles } cp, \text{Nat } i)$

{ if ( $i == 1$ )

    return  $cp \rightarrow \alpha [cp \rightarrow h_1 - 1]$ ;

if ( $i == 2$ )

    return  $cp \rightarrow v [N - cp \rightarrow h_2]$ ;

}

Nat somme (Cpiles Cp) {

Nat m=0;

if (b == 1)

for (int i=0; i < Cp → h1; i++)

m += Cp → v[i];

else

for (i=N-1; i >= N-Cp → h2; i--)

m += Cp → v[i]

return m;

}

Nat h (Cpiles, Nat i)

{

if (i == 1)

return Cp → h-1;

if (i == 2)

return Cp → h2;

}

Bool ev (Cpiles Cp; Nat z) {

if (z == 1)

return (Cp → h1 == 0);

if (z == 2)

return (Cp → h2 == 0);

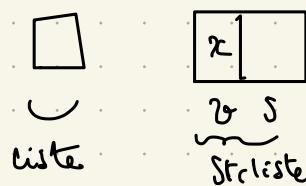
}

# Listes

## Représentation par chaînage

LISTE $\emptyset$  ( listenou  
 adjt  
 supt  
 tête  
 vide  
 lgr

### liste simple



### LISTE1. - app

- opérations sur la 1ère occurrence.
- opérations sur la queue de la liste
- op sur la i-ème : SPECIF

typedef struct str liste {

    S n;

    struct str liste \* s;

} str liste , \* liste ;

liste listenou () { return NULL ; }

liste adjt ( liste l , S a ) {

    // pas de précédent

    liste l1 = MALLOC ( str liste );

    l1 → a = a ;

    l1 → s = l ;

    l = l1 ;

    return l ;

}

liste supt ( liste l ) {

\* / liste non vide \*/

    liste save = l ;

    l = l → s ;

    free save ;

    return l ;

}

bool nicle ( liste l ) {

    return l == NULL ;

}

s tête ( liste l ) {

    return l → v ; }

Nat lgr ( liste l ) {

for ( nat h = 0 ; !nicle ( l ) ; h++ ; l = l → s )

    return h ;

}

```

Bool app (liste l, S x) {
 /* pas de freind */
 {
 Bool trouve = false;
 while ((l != NULL) && (trouve == false))
 if (l->v == x) trouve = true;
 l = l->s;
 return trouve;
 }
}

```

Variante

les deux marchent pareillement

sup1 : liste S → liste /\* suppression de la 1ère occurrence de x \*/

chg1 : liste S S → liste /\* remplacement de la 1ère occurrence de x \*/

rech1 : liste S → Nat /\* position dans la liste \*/

rech2 : liste S → liste /\* liste convergeant à la 1ère occur. \*/

### Quesions :

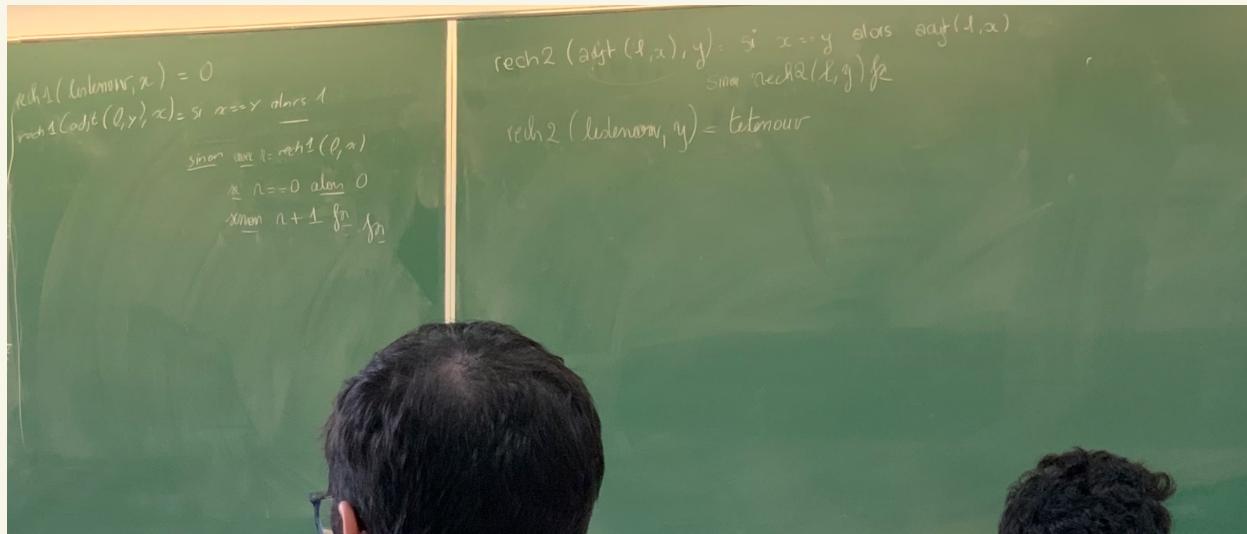
$$(a1) \sup(\emptyset, x) = \emptyset$$

$$(a2) \sup(i(l, y), x) = \begin{cases} \text{si } x = y \text{ alors } \sup(l, x) \\ \text{sinon } i(\sup(l, x), y) \\ \text{finsi} \end{cases}$$

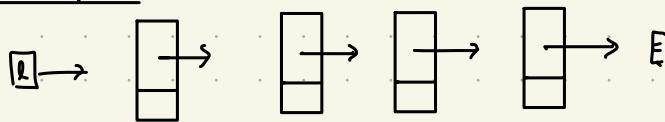
```

Nat sup (liste S, S x) {
 int h;
 if S == v

```



Continuation  
prochainne  
classe

Listes Simples

listenour  
adjt  
Supt  
lgr  
vide  
tete  
app

1ère occurrence /\* sans préconditions \*/

rech1 : liste S → Nat /\* position \*/

rech2 : liste S → liste /\* liste commençant à l'elt \*/

sup1 : liste S → liste

chg1 : liste S . S → liste

Opérations

```

Liste rech2 (Liste l, S x) {
 while (l != NULL) {
 if (l->v == x)
 return l;
 else
 l = l->s;
 }
 return l;
}

```

```

while ((l != NULL) && (l->v != x))
 l = l->s;
return l;

```

```

Liste chg1 (Liste l, S x, S y) {
 Liste l1 = rech2 (l, x);
 if l1 := NULL
 l1->v = y;
 return l;
}

```

axiome:

chg1 (listenour, x, y) = listenour()

chg1 (adjt(l, x), y, z) =

si  $x = y$  alors adjt (l, z)

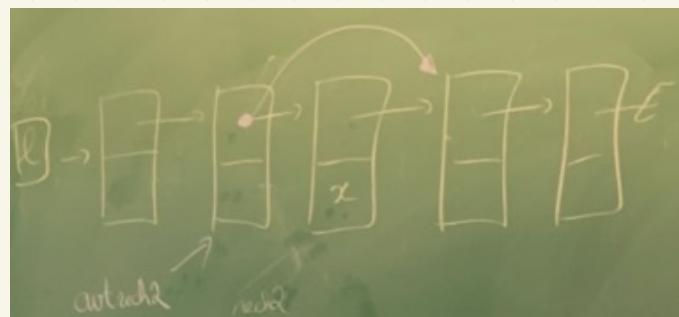
sinon adjt (chg1 (l, y, z), x) fsi

```

Liste autorech2 (Liste l, S x) {
 while ((l->n != NULL) && (l->s->v != x))
 l = l->s;
 return l;
}

```

/\* précondition : (l->NULL) et (l->v != x) \*/



```

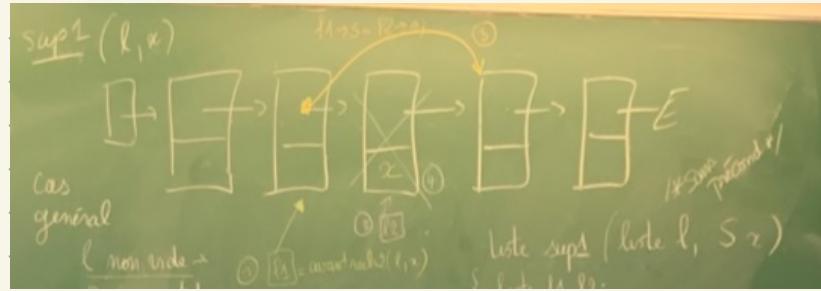
Liste sup1 (Liste l , s x) {
 Liste l1 , l2 ;
 if (l != NULL) {
 if (l->v == x)
 l = sup1 (l);
 else {
 l1 = autrecherche (l , x);
 if (l1 != NULL) {
 l2 = l1->s;
 l1->s = l2->s;
 free (l2);
 }
 }
 }
 return l;
}

```

axiome :

$\text{sup1} (\text{adit} (l, y), z) = \begin{cases} z & \text{si } x = y \\ \text{adit} (\text{sup1} (l, x), y) & \text{sinon} \end{cases}$

$\text{sup1} (\text{listenouv} (l, x)) = \text{listenouv} ()$



### Opérations sur le i-ème élément

\* i-ème :  $\text{Liste Nat} \rightarrow \text{Liste} / * \text{sous-liste } x /$

$\begin{cases} \cdot \text{ins} : \text{Liste Nat } s \rightarrow \text{Liste} \\ \cdot \text{sup} : \text{Liste Nat} \rightarrow \text{Liste} \\ \cdot \text{chg} : \text{Liste Nat } s \rightarrow \text{Liste} \end{cases}$ 
 $\} \text{ cours}$

$\text{lième} (\text{listenouv}, i) = \text{listenouv} / * \text{necess. comme ass d'arrêt récursif} /$  lorsque  $i > \lg r(l) + 1$

$\text{lième} (\text{adit} (l, x), i) = \underline{\text{si }} i = 1 \text{ alors adit} (l, x)$

sinon  $\text{lième} (l, i-1)$  fin

$/ * \text{ pré: } i \geq 1 *$

```

Liste lième (Liste l , Nat i) {
 while ((l != NULL) && (i != 1)) {
 l = l -> s;
 i = i - 1;
 }
 return l;
}

```

pré:  $i < \lg r (L)$ ,  $L > 0$

$\text{ins} (L, i, x) = \text{adit} (L, x)$

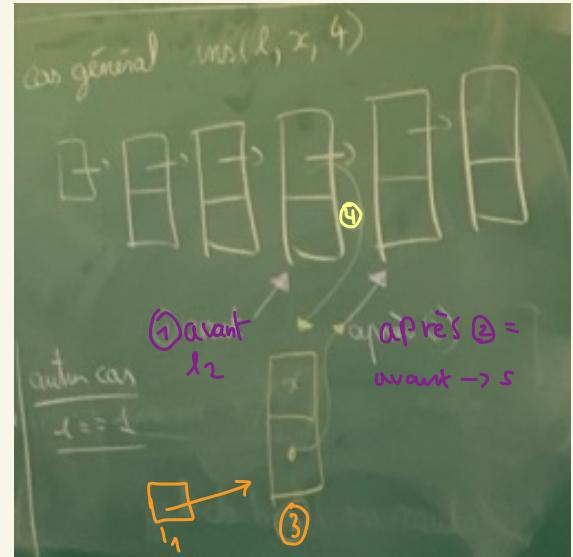
$\text{ins} (\text{adit} (L, x), i, y) = \text{adit} (\text{ins} (L, i-1, y), x)$

cas général  $\text{ins} (l, x, 3)$

```

liste ins (Liste l, Nat x, Symbole y) {
 if (l == NULL)
 l = adjt (l, y);
 else {
 ① Liste l1, l2 = reme (l, x - 1);
 l1 = MALLOC (Str liste);
 ③ l1 → v = y;
 l1 → s = l2;
 ④ l2 → s = l1;
 }
 return l;
}

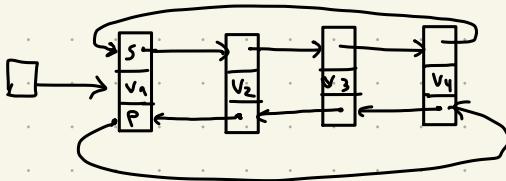
```



Faire les fcts pour lième ...

26-Nov-19

### Liste doublement chaînée



1 / typedef ...

2 / à faire op du noyau listnouv  
adjt ...

3 / opérations en lième position

|       |      |  |
|-------|------|--|
| lième |      |  |
| del   | sup. |  |
| ins   |      |  |

```

typedef struct sldc {
 S *v;
 struct Strlistedc *s;
 struct Strlistedc *p;
} *Liste, Strlistedc;

```

```

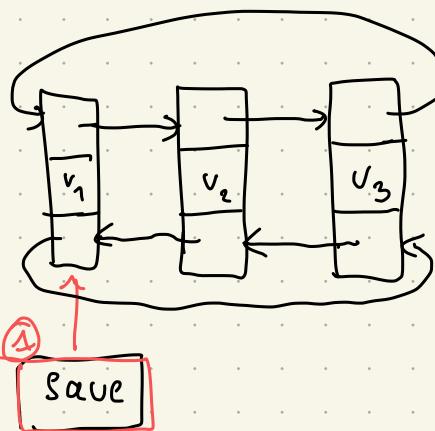
liste sup (liste l)
Liste save = l; ①
if (l → s == l → p) { free (l);
 return NULL; }
l → p → s = l → s; ②
l → s → p = l → p;
free (save);
return l;
}

```

```

liste listnouv () {
 return NULL;
}

```



liste supt ( liste l ) {

```
if (l == NULL) return NULL;
if (l == l -> s) { free (l); return NULL; }

l -> p -> s = l -> s;
l -> s -> p = l -> p;
liste save = l;

l = l -> s;
free save;
return l;
}
```

liste addit ( liste l , S x ) {

```
liste l1 = MALLOC (Strliste);
if (l == NULL) // listenode
{
 l1 -> v = x;
 l1 -> s = l1;
 l1 -> p = l1;
}
else
{
 l1 -> s = l;
 l -> p -> s = l1;
 l1 -> p = l -> p;
 l1 -> v = x;
 l -> p = l1;
}
return l1;
}
```

Nat lg. ( liste l ) {

```
if (l == NULL) return 0;
else {
 liste cp = l -> s
 Nat cpt = 1;
 while (cp != l) {
 cp = cp -> s;
 cpt++;
 }
 return cpt;
}
```

Bool nvide ( liste l ) {

```
return l == NULL;
```

S tete ( liste l ) {

```
return l -> v;
```

```
}
```

```

Nat ienne (Liste l , Nat x) {
 if (l != NULL)
 while (x > 1) {
 l = l -> s;
 x--;
 }
 return l -> v;
}

```

```

liste liene (liste l , Nat i) {
 while ((l != NULL) && (i != 1)) {
 l = l -> s;
 }
}

```

*/\* pré:  $i \geq 1$  \*/*

```

Liste liene (liste l , Nat i) {
 while ((l != NULL) && (i != 1)) {
 l = l -> s;
 i--;
 }
 return l;
}

```





- Preuve pour les Piles:

hauteur ( $p$ ) = hauteur (dépiler ( $p$ )) + 1  
 pour  $p$  non-vide

## • Preuve sur les files:

$$\text{lgr}(f) = \text{lgr}(\text{rupt}(f)) + 1.$$

pour  $f$  non vide.

forme canonique

$p = \text{empiler}(\text{empiler}(\dots \text{empiler}(\text{pileenav}, a_1), \dots), a_{n-1}), a_n) = \text{empiler}(p', a_n)$

$$F = \text{adj}_g (\underbrace{\text{adj}_g (\dots \text{adj}_g (\text{filenour}, a_1), \dots, a_{n-1}), a_n}_{f'})$$

forme canonique (suite)

d'après (p4) en substituant  
 $\begin{cases} p' \text{ à } p \\ q \text{ à } x \end{cases}$

$\text{empiler}(t, an)) + 1$  d'après (P) dans l'autre sens

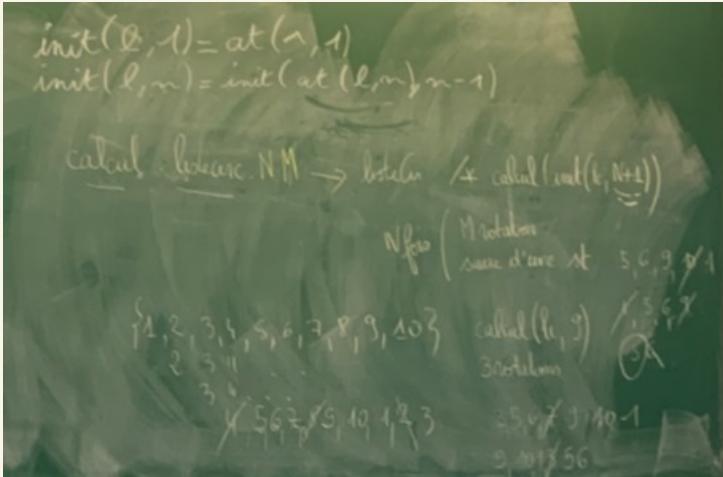
## Lutes circulaires

## Sorte liste og operations

  $\rightarrow$  hitecne  
 at sq: hitecne  $S \rightarrow$  hitecne  
 st, sq: hitecne  $\rightarrow$  hitecne  
 ev: hitecne  $\rightarrow$  hitecne  
 $\downarrow$   $\rightarrow S$

RotN : bidean Nat  $\rightarrow$  bidean /\* N rotations \*/

mais liste Nat  $\rightarrow$  liste circulaire /\* liste circulaire constituée d'entiers de 1 à N \*/



## TD 10 - 12 : Dernier TD 4h

listes: op sur la 1<sup>ère</sup> occurrence

$$\{ 1, 2, \cancel{X}, 2, 5, 6 \}$$

$$\begin{array}{c} \text{sup} \\ \text{ang} \end{array} (\underline{l}, \underline{y}, \underline{z})$$

- $\text{sup}(\text{listenouv}, \text{pos}) = \text{listenouv}$
- $\text{sup}(\text{adjt}(l, x), \text{pos}) = \begin{cases} \text{si } \text{pos} \leq 1 \text{ alors } \text{sup}(l) \\ \text{sinon si } \text{pos} > \text{lgr}(l)+1 \\ \text{sinon adjt}(\text{sup}(l, \text{pos}-1), x) \end{cases}$
- $\text{ins}(\text{listenouv}, \text{pos}, x) = \text{adjt}(\text{listenouv}, \text{pos})$
- $\text{ins}(\text{adjt}(l, y), \text{pos}, z) = \begin{cases} \text{si } \text{pos} \leq 1 \text{ alors adjt} \\ \text{sinon adjt}(\text{ins}(l, \text{pos}-1, y), z) \end{cases}$

$$\bullet \text{Sup}(\text{listenouv}, \text{pos}) = \text{listenouv}()$$

$$\begin{aligned} \text{Sup}(\text{adjt}(l, x), \text{pos}) = & \begin{cases} \text{si } (\text{pos} \leq 0) \text{ alors} \\ \text{si } (\text{pos} = 1) \vee (\text{pos} < 1) \text{ alors } l \\ \text{sinon si } (\text{pos} > (\text{lgr}(l)+1)) \text{ alors supg}(l) \\ \text{sinon adjt}(\text{sup}(l, \text{pos}-1), x) \end{cases} \end{aligned}$$

inclus cas générales

$$\begin{aligned} \text{sup}(\text{listenouv}(), \text{pos}) &= \text{listenouv}() && (\text{cas } n > \text{lgr}(l)+1) \text{ tout invalide} \\ \text{sup}(\text{adjt}(l, x), n) &= \begin{cases} \text{si } n < 1 \text{ alors } \text{adjt}(l, x) \\ \text{sinon si } n = 1 \text{ alors } l \\ \text{sinon adjt}(\text{sup}(l, n-1), x) \end{cases} \end{aligned}$$

$$\text{sup}(\text{adjt}(\text{listenouv}(), 3), 3) = \text{adjt}(\text{sup}(\text{listenouv}(), 2), 3)$$

$$= \text{adjt}(\text{listenouv}(), 3)$$

$\text{lig}(\text{listenouv}, n) = \text{listenouv}$

$\text{lig}(\text{adj}, t(l, x), n) = \begin{cases} \text{listenouv} & \text{si } n <= 1 \\ \text{adj}(\text{listenouv}, x) & \text{si } n > 1 \end{cases}$

$\text{lid}(\text{listenouv}, n) = \text{listenouv}$   
 $\text{lid}(\text{adj}(l, x), n) = \begin{cases} \text{si } n \leq 1 \text{ alors adj}(l, x) \\ \text{non si } n = 1 \text{ alors } l \\ \text{non si } n > 1 \text{ alors } \text{adj}(l, \text{listenouv}) \\ \text{non si } n > 1 \text{ alors } \text{adj}(l, \text{listenouv}), \text{ lid}(l, n-1) \end{cases}$

$\text{lid}(\text{adj}(\text{adj}(\text{listenouv}, x_1), x_2), 1) = \text{adj}(\text{adj}(\text{listenouv}, x_1), x_2)$   
 $\text{lid}(\text{adj}(\text{adj}(\text{listenouv}, x_1), x_2), 3) = \text{adj}(\text{adj}(\text{listenouv}, x_1), 2)$   
 $= \text{listenouv}$

/ sans précondition /

$\text{lig}(\text{listenouv}, n) = \text{listenouv}$

$\text{lig}(\text{adj}(l, x), n) = \begin{cases} \text{si } n \leq 1 \text{ alors listenouv} \\ \text{non si } n > 1 \text{ alors adj}(l, \text{listenouv}) \\ \text{non si } n > 1 \text{ alors adj}(\text{lig}(l, n-1), x) \end{cases}$

{a, b, c, d, e, f}



1

prouver par induction que

$$|s(e, x)| = \boxed{\text{si } x \in e \text{ alors } |e|-1} \text{ sinon } |s(e, x)|$$

prendre en raisonnement la longueur de la forme canonique

$$e = i(e', n_k) = i(i(\dots i(i(\phi, n_1), \dots, n_{k-1}), n_k))$$

dans le cas où  $x \in e$  montrons que  $|s(e, x)| = |e|-1$

cas simple  $k=1$  montrons que

$$|s(i(\phi, n_1), x)| = |i(\phi, n_1)| - 1$$

$$\text{sous l'hypothèse } x \in i(\phi, n_1) \stackrel{(e2)}{\Rightarrow} x = n_1 \text{ ou } x \in \phi \stackrel{(e1)}{\Rightarrow} (x = n_1)$$

parlons de  $|s(i(\phi, n_1), x)|$   
comme  $x = n_1 \stackrel{(e1)}{\Rightarrow} |\phi| \stackrel{(e5)}{=} 0$

parlons de  $|i(\phi, n_1)| - 1$   
 $\stackrel{(e6)}{=} |\phi| + 1 - 1 \stackrel{(e5)}{=} 0$

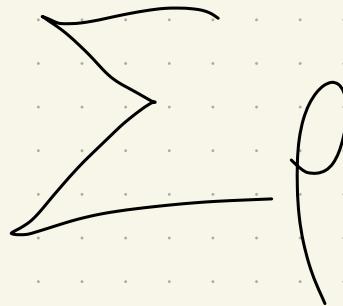
Cas général  $k \geq 1$  avec  $e' = i(-\iota(\phi, m_1), \dots, m_{k-1})$

hypothèse de récurrence vraie pour la longueur  $k-1$  cad  $|S(e', x)| = \underline{x} \in e'$  alors  $|e'| - 1 \leq |S(e', x)|$  montrons que c'est vrai pour la longueur  $k$  cad  $|S(e, x)| \leq |S(i(e', m_k), x)| = |i(e', m_k)| - 1 = |e| - 1$

- d'après (et)  $S(i(e', m_k), x) = \underline{\underline{m_k}} = x$

Deux cas sont à examiner

(cas 1)  $x = m_k = \underline{\underline{m_k}} = x$  alors  $S(i(e', m_k), x) = e'$  par déf de  $|S(i(e', m_k), x)| = |i(s(e', x), m_k)|$   
par déf de  $|i(e', m_k)| - 1 \stackrel{(e)}{=} |e'| + 1 - 1 = |e'|$   $\stackrel{\text{def}}{=} |S(i(e', m_k), x)| + 1$  applique !  
par déf  $|e| - 1 = |i(e', m_k)| - 1 \stackrel{(e)}{=} |e'| + 1 - 1$



```
Void proc (int n)
{
 int i, j, k;
 for (i=1 ; i <= n ; i++)
 for (j = i+1 ; j <= n ; j++)
 for (k = 1 ; K <= j ; K++) {
 instruction;
 }
}
```

---