1
STRUCTURES DE DONNEES ELEMENTAIRES

PLAN

- **Types abstraits et structures de données**
- **Piles et files**
- **Listes chaînées**
- **Listes contiguës ordonnées**

2.1 TYPES ABSTRAITS ET STRUCTURES DE DONNEES

hiérarchie des types:

types prédéfinis (integer, boolean,...)

types composés (tableau d'entiers…)

type de données abstrait:

description d'un ensemble organisé d'objets et des opérations sur cet ensemble

structure de données:

impléméntation explicite d'un type de données

2.2 ENSEMBLES DYNAMIQUES

Mathématique → **ensemble statique**

Informatique → **ensemble dynamique**

2-2-1 CLÉS

clé = identificateur d'un élément

(N° SS, noms, N°immatriculation,.)

{clés} totalement ordonné

Classe élément (sera notée Elt)

- **clé (entier ou de la classe C_clé)**
- **contenu (classe C_elt quelconque)**

2-2-2 OPÉRATIONS SUR UN ENSEMBLE DYNAMIQUE

-- requête retourne une information

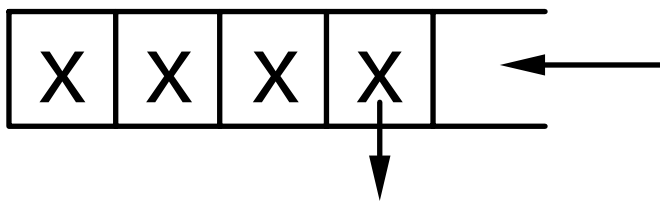
-- modification modifie l'ensemble

- vide
- longueur
- recherche
- insertion
- suppression

…

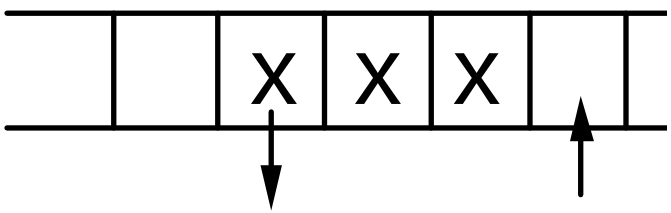
2-2-3 DIFFÉRENTS TYPES DE STRUCTURES LINEAIRES

piles **insertion et suppression d'un même côté**



→ LIFO

files **insertion d'un côté et suppression de l'autre**



→ FIFO

listes **insertion et suppression n'importe où (accès direct** (Indice) **ou par chaînage)**

2-3 PILES ET FILES

structures linéaires dynamiques dont les éléments sont de la classe Elt quelconque

2.3.1 LES PILES

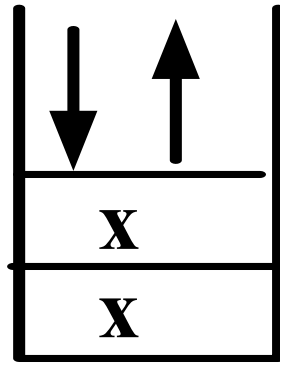
insérer	=	**empiler**
supprimer	=	**dépiler**

- Une pile est un tableau qui contient des éléments (classe Elt).
- On peut ajouter un élément "au-dessus" de la pile si le tableau n'est pas "plein"
- On peut retirer l'élément du "dessus" de la pile si le tableau n'est pas "vide"
- le seul élément de la pile auquel on peut avoir accès est le dernier élément qui y a été ajouté.

On va définir une classe "Pile"

implémentation d'une pile

```
classe Pile{
   entier taille;   //taille du tableau représentant la pile
   Elt[ ] tPile;    //ce tableau contient des objets de la classe Elt
   entier long=0;   //nombre d'éléments dans la pile

   Pile( entier t){
      this.taille=t;
      this.tPile=new Elt[t];
    }
//les méthodes
}
```



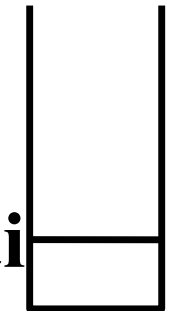
méthodes	**conditions**
sommet retourne Elt	**pile non vide**
empiler (Elt x)	**pile non pleine**
dépiler( )	**pile non vide**

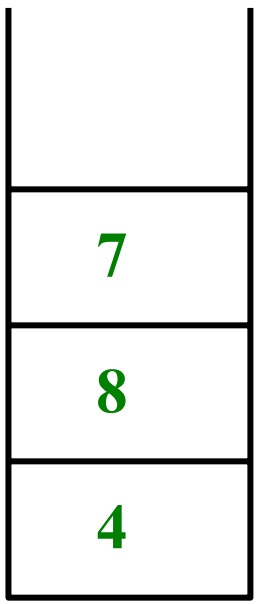
estvide( ) retourne booléen

EXEMPLE

pile=new Pile(10)

pile.estvide( )=vrai





pile.empiler (4)

pile.empiler (8)

pile.empiler (7)

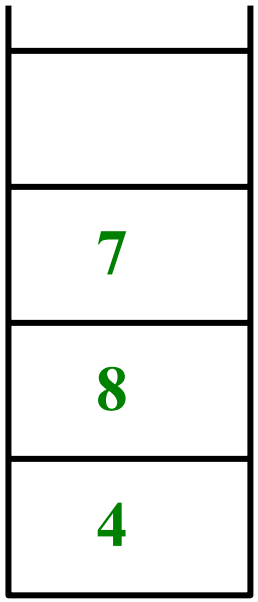
pile.sommet

~~4~~

~~8~~

7

EXEMPLE



pile.empiler (9)

pile.dépiler

pile.sommet

~~7~~

~~9~~

7

pile.estvide( )=faux

```
booléen estvide( )
début
   retourner (this.long==0);
fin
```

fonction en

« temps constant »

(ici = 1 instruction)

Noté O(1)

Lu Ode1

```
Elt  sommet  ( )
début
   si  this.estvide( )   alors  "erreur";
   sinon
        retourner this.tPile[long-1];
   finsi;
fin
```

fonction en temps constant=O(1)

Le temps d'exécution ne dépend pas de la taille de la pile

```
void empiler  (Elt x)
début
  si      this.long==taille   alors  "erreur";
  sinon
        this.long = this.long+1;
        this.tPile[long-1] = x;
  finsi;
fin
```

procédure en O(1)

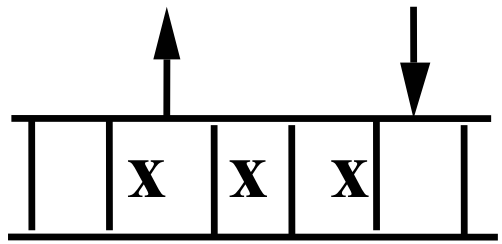
```
void  dépiler  ( )
début
  si       this.long==0   alors  "erreur";
  sinon
           this.long = this.long-1;
  finsi;
fin
```

procédure en O(1)

2-3-2 LES FILES

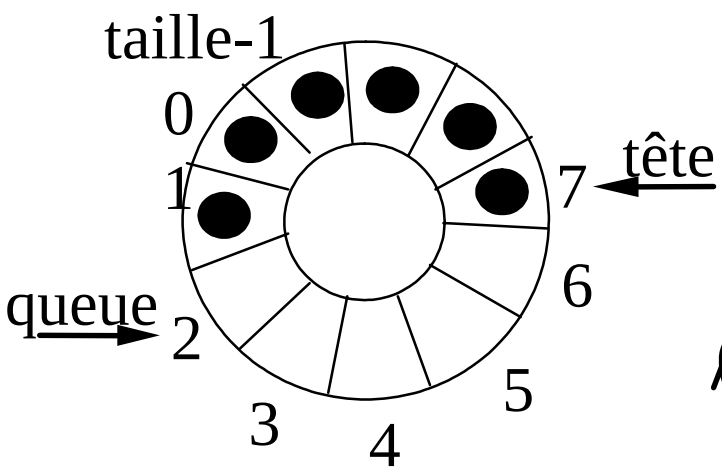
insérer = enfiler

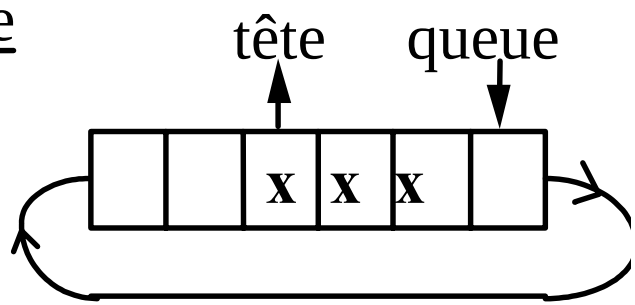
supprimer = défiler



exemples: file d'attente à un guichet, tampon de messages

en général: file circulaire





- Une file est un tableau qui contient des éléments (classe Elt).
- On peut ajouter un élément "à la fin" (queue) de la file si le tableau n'est pas "plein"
- On peut retirer l'élément "au début" (tête) de la file si le tableau n'est pas "vide"
- le seul élément de la file auquel on peut avoir accès est le plus ancien élément qui y a été placé.

On va définir une classe "File"

implémentation d'une file

```
classe File{
   entier taille;
   Elt[ ] tFile;
   entier tête=0; entier queue=0;

File( entier t){
   this.taille=t;
   this.tFile=new Elt[t];
   }
//les méthodes
}
```

méthodes	**conditions**
entête( ) retourne Elt	**file non vide**
enfiler(Elt x)	**file non pleine** **dépôt en queue**
défiler( )	**file non vide** **retrait en tête**

estvide( ) retourne booléen

estpleine( ) retourne booléen

file vide: *tête == queue*

file pleine: *tête == (queue+1) mod taille*

EXEMPLE

file= new File(6)

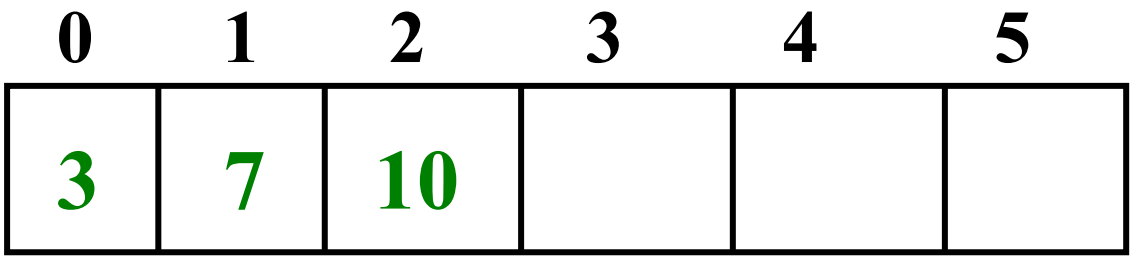
file.tête=
file.queue=0

file.estvide=vrai

file.enfiler(3)

file.enfiler(7)

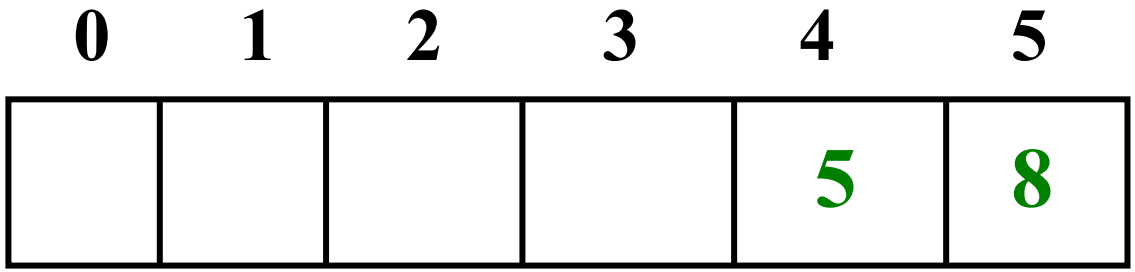
file.enfiler(10)



file.tête= 0 file.queue= ~~1~~
~~2~~
3

file.estvide( )=faux

EXEMPLE



file.défiler()

file.défiler()

file.tête=~~0~~ file.queue=~~3~~ file.enfiler(2)

~~1~~ file.défiler()

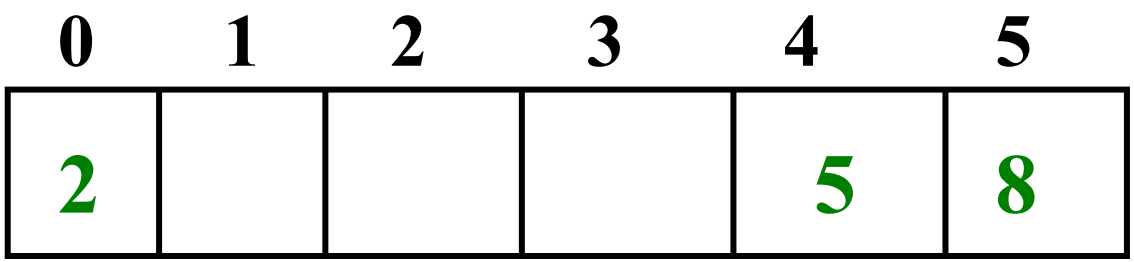
2 file.enfiler(5)

4 file.défiler()

file.tête=4 file.queue=~~5~~ file.enfiler(8)

(5+1) mod 6 = 0

EXEMPLE

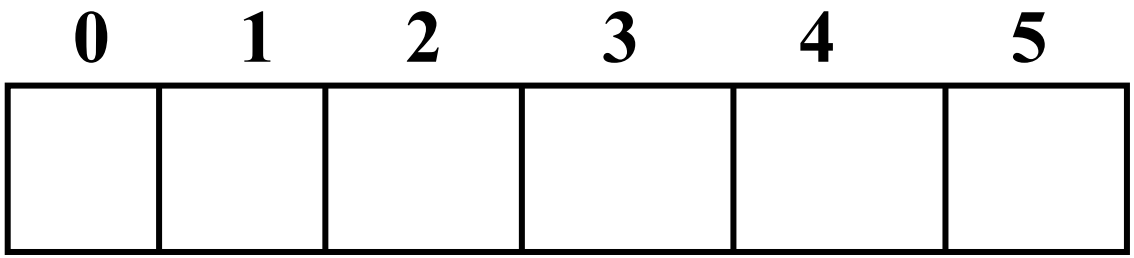


file.tête=4
file.queue=0

file.enfiler(2)

file.tête=4
file.queue=1

EXEMPLE



file.défiler()

file.défiler()

file.tête=~~4~~ ~~5~~ (5+1) mod6=~~0~~ 1 **file.défiler()**

file.queue= 1

File vide car tête= queue (= 1)

booléen **estvide ( )** ;

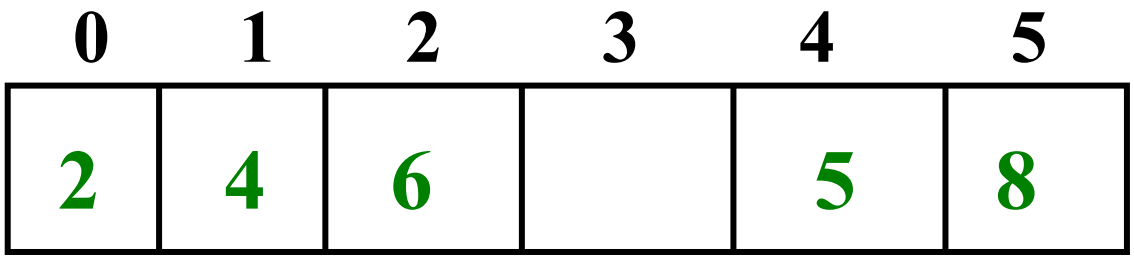
début

retourner (this.**tête**==this.**queue);**

fin

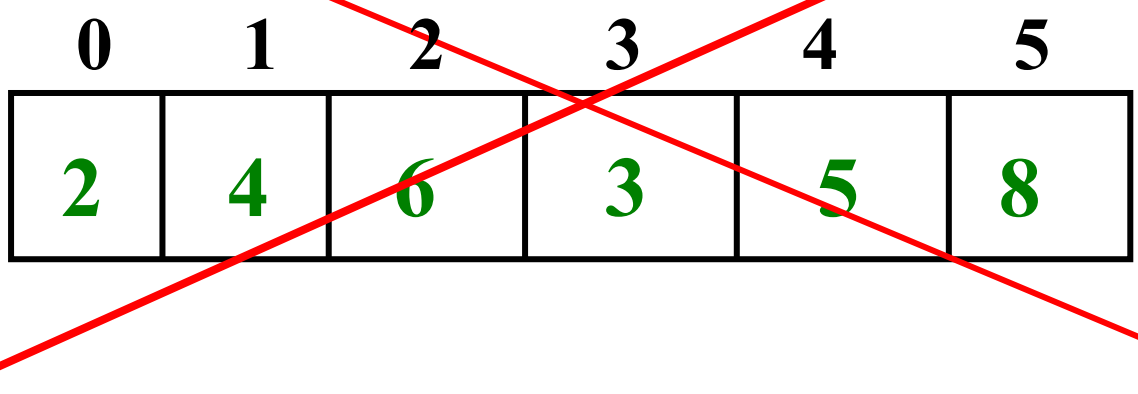
fonction en O(1)

EXEMPLE



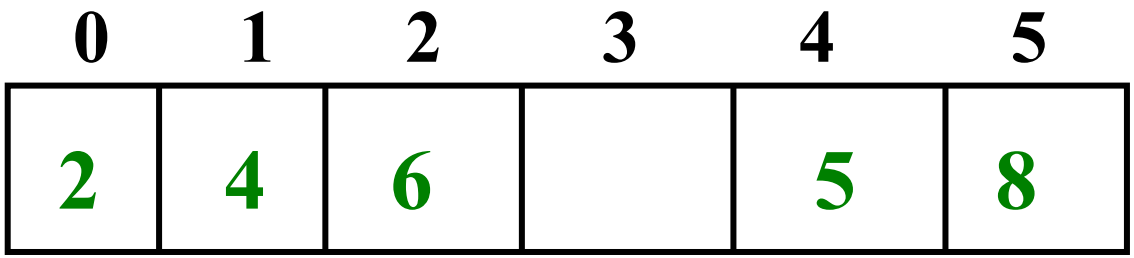
file.tête=4
file.queue=3

file.enfiler(3)



file.tête=4
file.queue=4

EXEMPLE



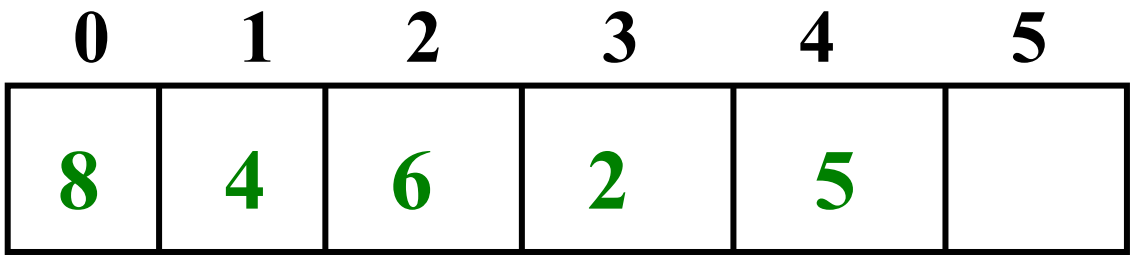
file.tête=4
file.queue=3

La file est pleine **file.enfiler(3) est interdit**

tête = queue+1

une file est "pleine" si
il reste une seule place non occupée dans le tableau

EXEMPLE



file.tête=0
file.queue=5

La file est pleine **file.enfiler(3) est interdit**

tête = (queue+1) mod 6

Remarque:
si queue<5 alors (queue+1) mod 6 = queue + 1
si queue=5 alors (queue+1) mod 6 = 0

LES METHODES

booléen estpleine ( )

début

retourner

(this.**tête** == (this.**queue** +1) **mod** this.**taille**);

fin

Elt entête ( )

début

si this.**estvide( )** alors "erreur";

sinon **retourner** this.**tFile[tête]**;

finsi;

fonctions en O(1)

fin

```
void  enfiler  (Elt x)
début
   si       this.estpleine( )   alors  "erreur";
   sinon
        this.tFile[queue] = x;
        this.queue = (this.queue+1) mod this.taille;
    finsi;
fin
```

procédure en O(1)

```
void  défiler  ( )
début
  si       this.estvide( )   alors  "erreur";
  sinon
        this.tête = (this.tête +1)mod this.taille;
  finsi;
fin
```

procédure en O(1)

2-4 LISTES CHAÎNEES

2-4-1 LES POINTEURS

Intérêt:

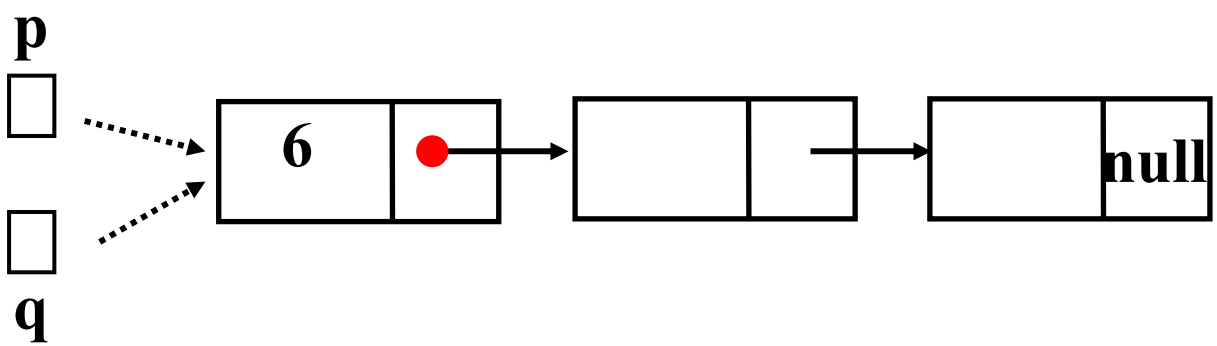
- créer dynamiquement des objets

- réaliser des structures chaînées

Une "place" contient un élément et un pointeur

valeur **élément de la classe Elt**

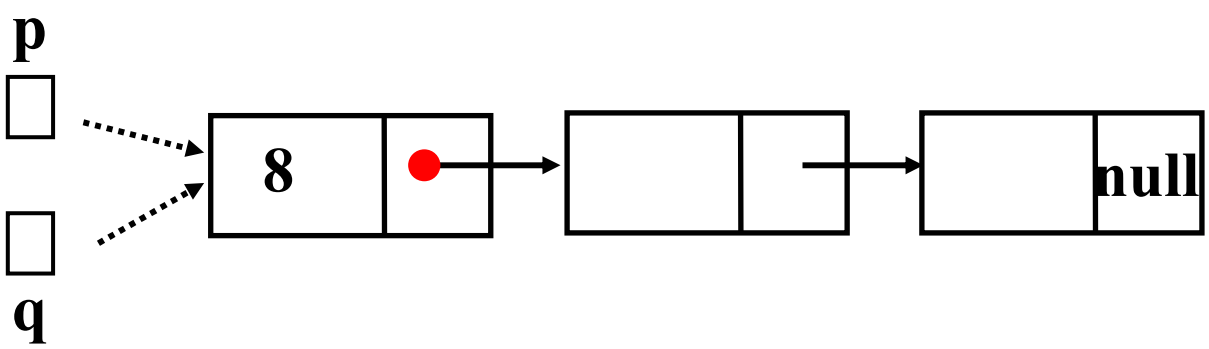
suivant **"pointe" sur la**
place suivante (null si elle n'existe pas)



p,q //pointeurs sur une "place"

p.valeur=6;

q=p; //alors q.valeur=6 et q.suivant=p.suivant



p,q //pointeurs sur une "place"

p.valeur=6;

q=p; //alors q.valeur=6 et q.suivant=p.suivant

q.valeur=8; //alors p.valeur=8 aussi

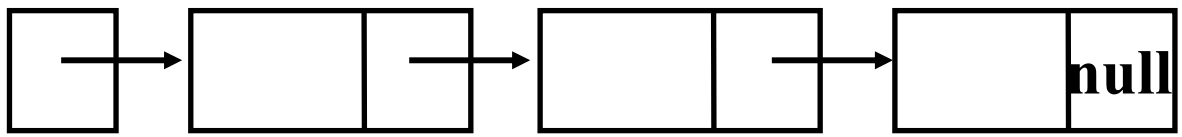
p = null; //p ne pointe sur rien

//null : fin de structure chaînée

2-4-2 LES LISTES CHAÎNEES

structure dynamique souple dont les éléments sont de la classe Elt quelconque

insérer et supprimer en n'importe quel point



une "place" =

valeur: Elt élément

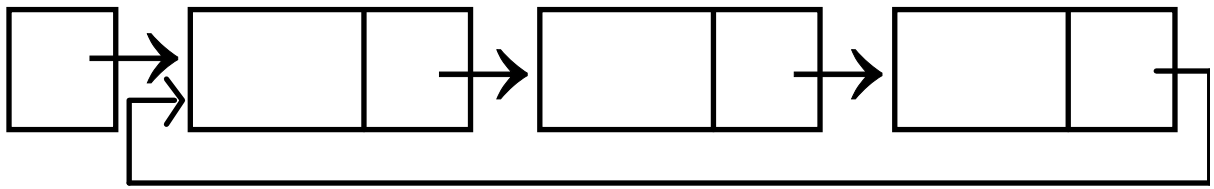
suivant: Liste référence

une liste: référence sur la première place

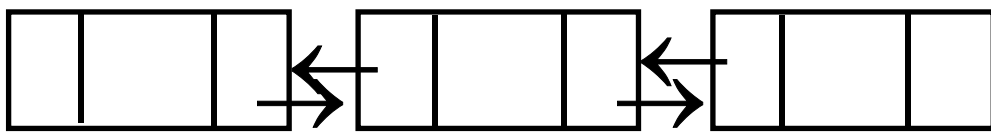
- Une liste chaînée est une structure souple qui contient des éléments (classe Elt).
- On peut ajouter un élément, sans condition, à n'importe quel endroit de la liste
- On peut retirer un élément à n'importe quel endroit de la liste si elle n'est pas "vide"
- on accède à un élément seulement après avoir accédé successivement à tous ceux qui le précèdent.

On va définir une classe "Liste" (chaînée)

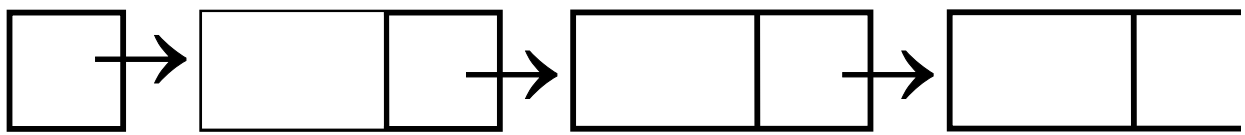
liste circulaire



liste doublement chaînée



Implémentation d'une liste chaînée simple



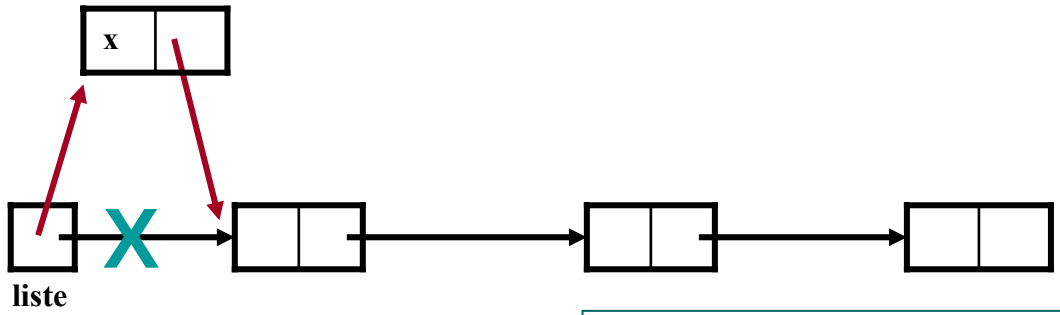
```
classe Liste {
    Elt  valeur;
    Liste suivant;

    Liste(Elt element, Liste reste){
        this.valeur = element;
        this.suivant = reste;
        }
//les méthodes
}
```

création d'une liste avec un élément x :

```
Liste liste = new Liste(x,null);
```

Dans tout le paragraphe 2.4.2 on suppose qu'un appel d'une méthode sur liste vide (liste=null) lève une exception que nous ne préciserons pas ici.

LES METHODES



liste = liste.insérerentête(x)

Liste **insérerentête (Elt x)**

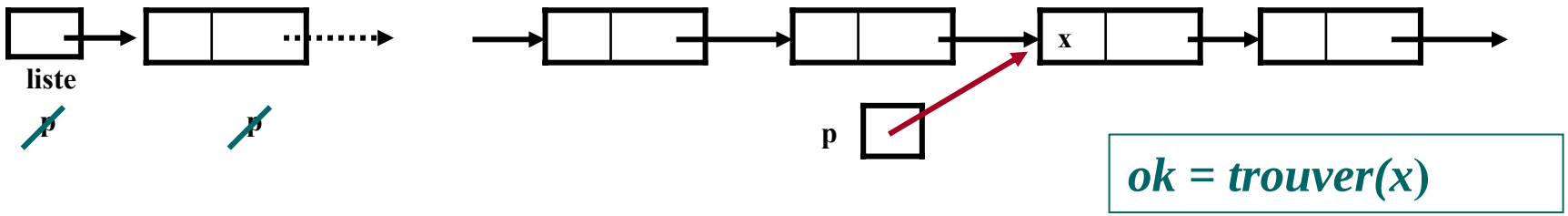
//insère x en tête de la liste

début

retourner new Liste(x,this)

fin

procédure en O(1)



booléen **trouver (Elt x)**

//cherche l'élément x dans la liste; renvoie vrai si il y est, renvoie faux sinon

début

Liste p = this;

tant que **p≠ null** faire

si **p.valeur==x** alors **retourner vrai**;

sinon **p = p.suivant**;

finsi;

fait;

retourner faux;

fin

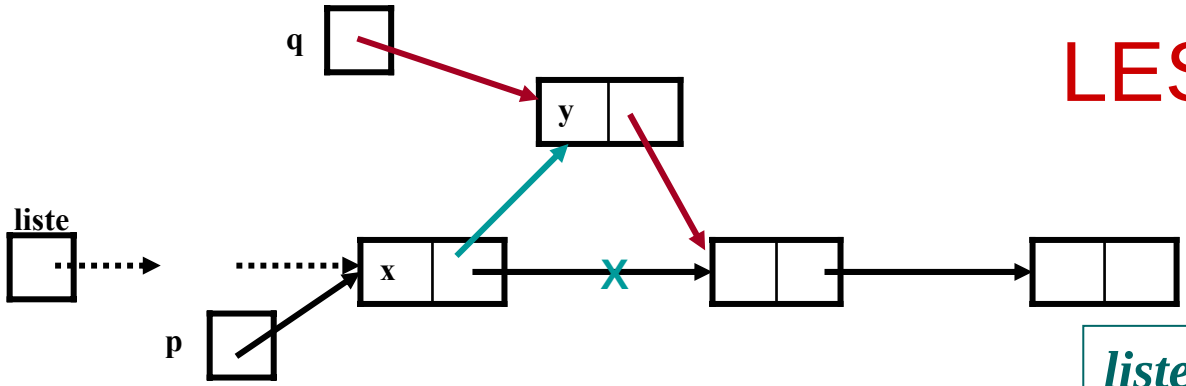
LES METHODES

```
booléen trouver (Elt x)
//cherche l'élément x dans la liste; renvoie vrai si il y est, renvoie
    faux sinon
début
 Liste p = this;
 tant que   p≠ null faire
    si  p.valeur==x    alors  retourner vrai;
    sinon  p = p.suivant;
    finsi;
fait;
retourner faux;
fin
```

Le temps d'exécution "dépend" du nombre *n* d'éléments dans la liste

Procédure en O(*n*)

LES METHODES



liste = liste.insérerapres(x,y)

Liste insérerapres (Elt x; Elt y)

// ajoute l'élément y après x, si x est présent;

début

Liste p = this;

tant que **p ≠ null** et **p.valeur ≠ x** faire **p = p.suivant**;fait;

si **p==null** alors **"erreur"** *//élément x absent de la liste*

sinon **Liste q = new Liste(y,p.suivant)**;

p.suivant = q ;

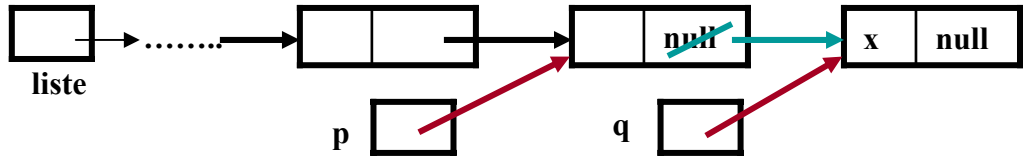
finsi;

Retourner p;

fin

procédure en O(*n*)

LES METHODES



liste = liste.insérerenqueue(x)

Liste insérerenqueue (Elt x)

//insère x en queue de la liste (en tête si la liste est vide)

début

Liste p = this;

Liste q= new Liste(x,null);

tant que **p.suivant ≠ null** faire

p = p.suivant;

fait;

p.suivant = q;

retourner this;

finsi;

fin

procédure en O(*n*)

Liste supprimer(Elt x)

//supprime l'élément x de la liste;

//erreur si x n'est pas présent

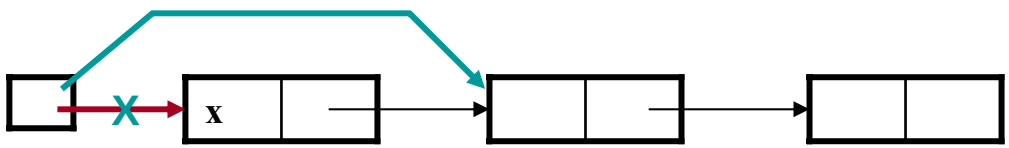
début

Liste p = this; *//sera la place de x*

Liste q = this; *//sera la place précédant p*

si **p.valeur==x** alors **q = p.suivant**; **retourner q**; *//x est le premier*

finsi;



liste
p=q

liste = liste.supprimer(x)

//cas où x est le premier

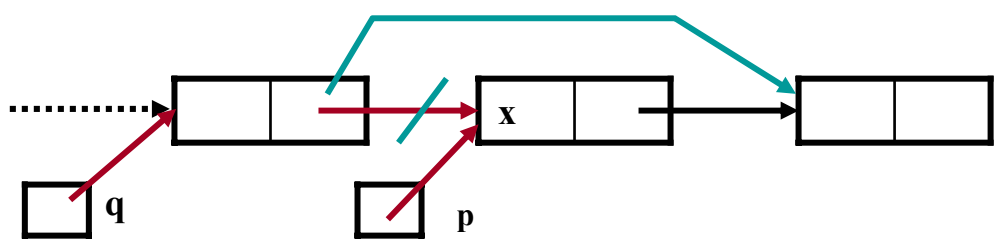
Liste supprimer(Elt x)

//supprime l'élément x de la liste;

//erreur si x n'est pas présent

procédure en O(*n*)

```
début
    Liste p = this;                 //sera la place de x
    Liste q = this;                 //sera la place précédant p
    si  p.valeur==x   alors  q = p.suivant;  retourner q; finsi;
    tant que  p ≠ null faire
          si  p.valeur==x   alors  q.suivant = p.suivant;  retourner this;
          sinon     q = p;
                    p = p.suivant;
          finsi;
    fait;
    "erreur";  //élément absent
fin
```



liste = liste.supprimer(x)

remarque

supprimer ne rend pas les emplacements mémoire au système; il y a un "ramasse-miettes" qui récupère les places libérées.

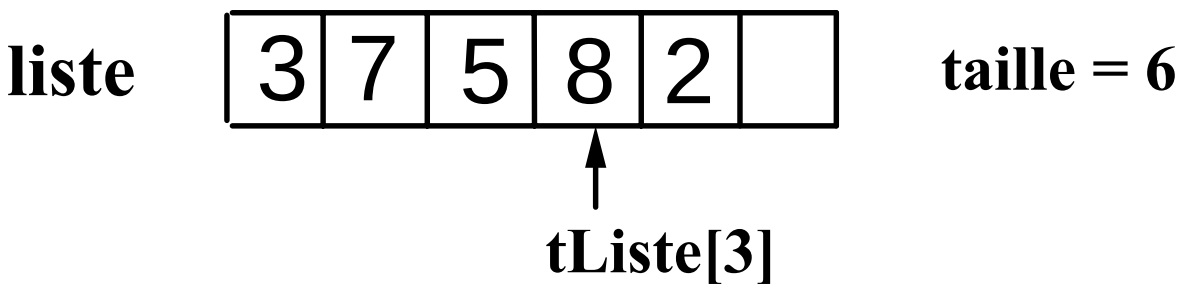
trouver, supprimer, insérerenrès et insérerenqueue en O(n)

insérerentête en O(1)

concaténation et scission simples

2-5 LISTES CONTIGUËS
LISTES ORDONNEES

2-5-1 LISTE CONTIGUËS

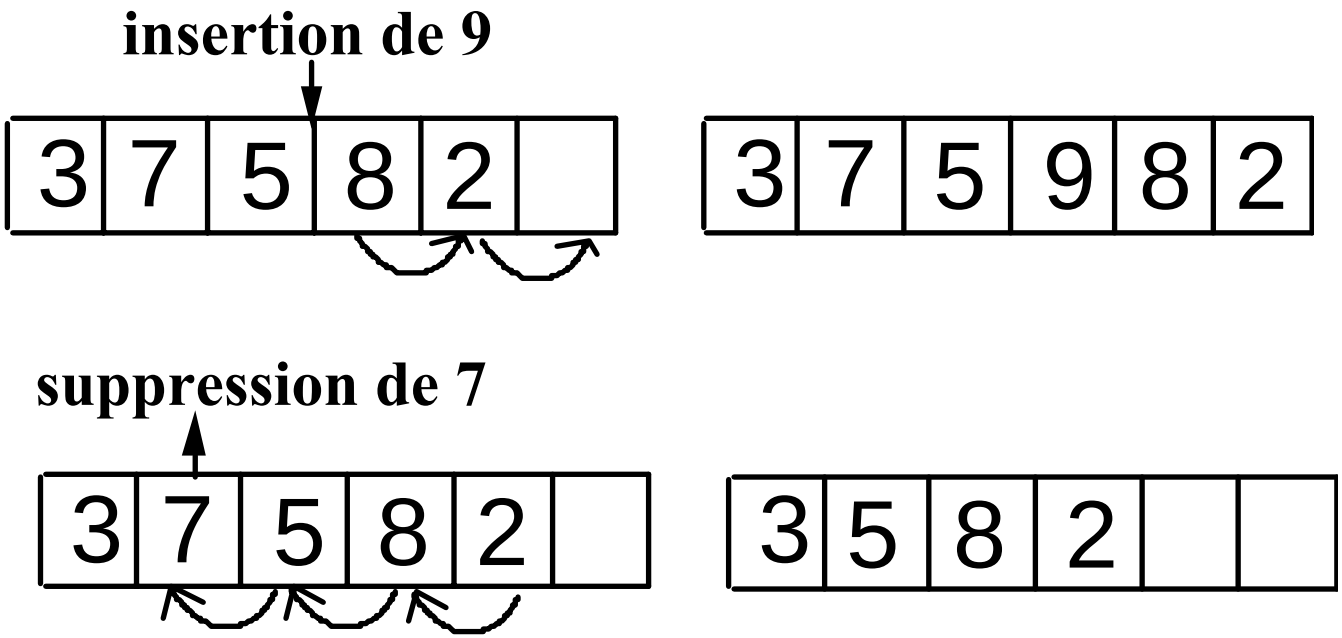


Longueur de la liste: 5

- Une liste contiguë est un tableau qui contient des éléments (classe Elt).
- On peut ajouter un élément à n'importe quel endroit si le tableau n'est pas "plein"
- On peut retirer un élément à n'importe quel endroit si le tableau n'est pas "vide"
- On peut accéder à tout élément directement à partir de sa place (Indice) dans le tableau

accès facile au ième élément: en O(1)

insertion (si liste non pleine) **et suppression** (si liste non vide) **plus difficiles : en O(longueur)**



2-5-2 LISTE CONTIGÜES ORDONNEES

un élément → une clé

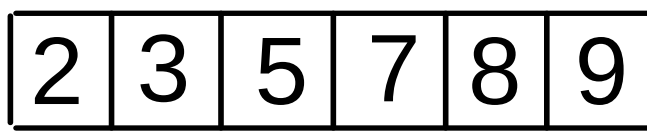
éléments rangés dans une liste dans l'ordre croissant des clés

Classe Elt {

entier clé

T_elt contenu

}



Classe LO //Liste ordonnée;

- Une liste ordonnée est un tableau qui contient des éléments (classe Elt) classés par ordre croissant de leurs clés.
- On peut ajouter un élément après un élément de clé inférieure et avant un élément de clé supérieure (ou =) si le tableau n'est pas "plein"
- On peut retirer un élément à n'importe quel endroit si le tableau n'est pas "vide"
- On peut accéder à tout élément directement à partir de sa place (Indice) dans le tableau

On va définir une classe "LO": "Liste ordonnée"

implémentation d'une liste ordonnée

```
classe LO{
   entier taille;  //constante
   Elt[ ] tab;
   entier longueur = 0;

LO ( entier t){
   this.taille=t;
   this.tab=new Elt[t];
   }
// les méthodes
}
```

On définit une classe Indice pour les indices du tableau tab (entier entre 0 et taille-1)

Dans la suite on omettra d'écrire "this" pour simplifier.

recherche dans une liste ordonnée

-pour chercher un élément e

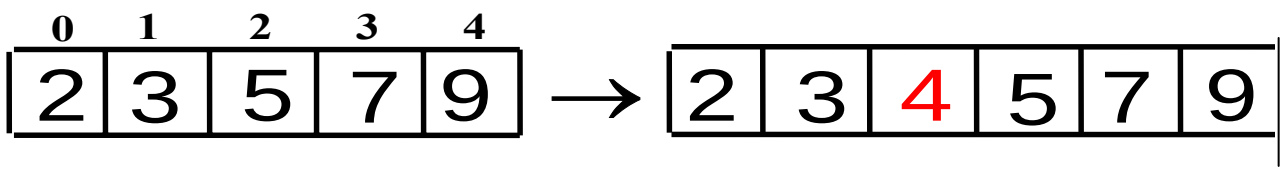
-pour insérer un nouvel élément e

***recherche de l'indice i du premier élément de clé ≥ à la clé de e**

***si tab[i].clé = e.clé, e est dans tab en i (trouvé=vrai)**

***sinon, e n'est pas dans tab et on peut l'insérer en i (trouvé=faux)**

EXEMPLE ***où insérer 4?***



k=0 tab[0].clé<4, k=1 tab[1].clé <4, k=2 tab[2].clé>4

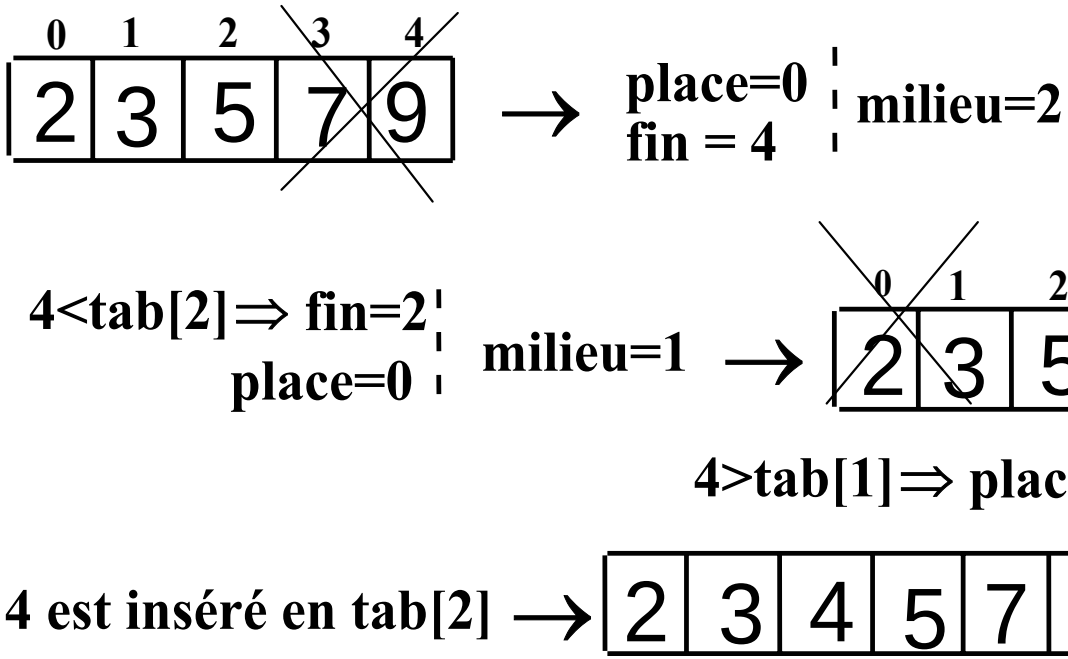
recherche séquentielle

```
void rech_séq (Elt x,   Indice place,
Booléan trouvé)
// indique vrai et place si l'élément est
//présent; la place où l'insérer sinon
début
Indice i = new Indice (0);
int val=x.clé;      trouvé = faux;
//il s'agit d'abord de la place
tant que  non trouvé et
                          i ≤ longueur -1 faire
          si  tab[i].clé ≥ val
            alors  place = i;  trouvé = vrai;
          sinon  i = i+1 ;
          finsi;
fait;
si  non trouvé  //la liste est vide ou bien
//val est > au plus grand élément
alors  place = longueur;
sinon       si  tab[place].clé ≠ val
            alors  trouvé = faux;
            //élément absent
            finsi;
finsi;
          //si trouvé reste à vrai,
          //l'élément est présent
fin
```

procédure en O(*n*)

recherche dichotomique

EXEMPLE



void **rech_dic** **(Elt x, Booléan trouvé, Indice place)**

//renvoie vrai et place si l'élément est présent;

//renvoie faux et indique la place où l'insérer sinon

LES METHODES

```
début
place = 0; trouvé = faux;
Indice fin=new Indice (longueur-1);
int val=x.clé;
si  fin ≠ -1 alors
//sinon, x absent à insérer en place 0
  tant que  place < fin  faire
    //val est dans le paquet d'au moins 2
    //éléments situé entre  place et fin;
    milieu = (place+fin)/2 ;
     si  tab[milieu].clé < val   alors
    //val est dans la partie de droite
          place = milieu+1 ;
     sinon  //val est à gauche
          fin = milieu ;
     finsi;
  fait;
   si tab[place].clé== val
        alors trouvé = vrai;
   sinon          //clé absente;
        //place sert à l'insérer
     si   place==tab.longueur-1
          et
          tab[place].clé < val
                    //clé > que tous
     alors place = place+1;
                    //à la fin
     finsi;
   finsi;
finsi;
fin
```

Complexité en O(log n)

nombre k de passages dans la boucle:

on divise le nombre d'éléments restants par 2

jusqu'à ce qu'il n'en reste qu'un (k divisions)

$((((n/2)/2)/2)/../2)=1$

soit $n/2^k = 1$ et donc $k=\log_2 n$

insertion dans une liste ordonnée tab

insertion de e de clé c en tab[i] autorisée seulement si

-longueur==0 et i==0 *//liste vide*

ou

-0 < longueur == n < taille *//liste non pleine*

et [0 < i ≤ n-1

et tab[i-1].clé < c <tab[i].clé *//insertion en milieu de liste*

ou i==n et tab[n-1].clé < c] *//insertion en fin de liste*

Remarque: les deux procédures de recherche précédentes autorisent les "doublons"

insertion dans une liste ordonnée tab sans doublons

void **insérer (Elt x)**

//insère l'élément x dans la liste si x n'est pas déjà présent

booléen **trouvé;** Indice **place;**

début

si longueur==taille alors "erreur" finsi; //liste pleine

rech_dic (x.clé,trouvé,place);

si trouvé==vrai alors erreur; //élément déjà présent

sinon

pour **i=this.longueur-1** à **i=place** pas **-1**

faire **tab[i+1]=tab[i]** fait;

tab[place] = x;

longueur=longueur+1;

finsi;

fin

procédure en O(*n*)

2-7 CONCLUSION

- **liste chaînée**

  - accès par le rang en O(n)

  - adjonction et suppression

  en O(1) si on connaît la place; en O(n) sinon

  - recherche en O(n)

- **liste contiguë**

  - accès par le rang en O(1)

  - adjonction et suppression en O(n)

  - listes ordonnées → recherche dichotomique en O(log n)

- **file ou pile**

  - liste contiguë particulière efficace mais peu souple