XII. Piles, files, listes chaînées

1. Généralités
2. Les Piles
3. Les Files
4. Les listes chaînées

1. Généralités sur les piles, files et listes chaînées

Suite d'éléments **ordonnée** :

- en fonction de l'ordre d'arrivée (pile, file)
- ou d'une relation d'ordre spécifique (liste chaînée)

Trois "supports" classiques :

- tableaux
- structures avec allocation dynamique
- fichiers

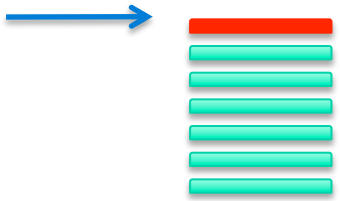
Représentation de l'ordre :

- ordre physique (tableaux, fichiers),
- ordre logique via une table d'accès (indirection)
- chaque élément désigne son suivant (champ spécial)

Principes des piles et des files

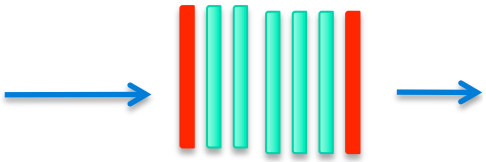
Les **piles** *(dernier arrivé, premier traité)* :

- ajout/suppression au **sommet** de pile



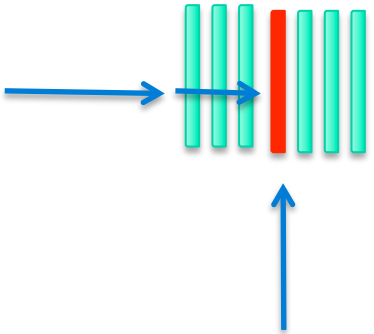
Les **files** *(premier arrivé, premier traité)* :

- ajout en **queue** de file
- suppression en **tête** de file



Les **listes chaînées** *(ordre non chronologique)* :

- accès par une **tête** de liste
- ajout/suppression n'importe où dans la liste



Cinq opérations de base sur ces suites :

1. **initialiser** la suite
2. **tester** si la suite est **pleine** (plus de place)
3. **tester** si la suite est **vide**
4. **ajouter** un élément (si place disponible)
5. **enlever** un élément (si suite non-vide)

➔ Cinq fonctions

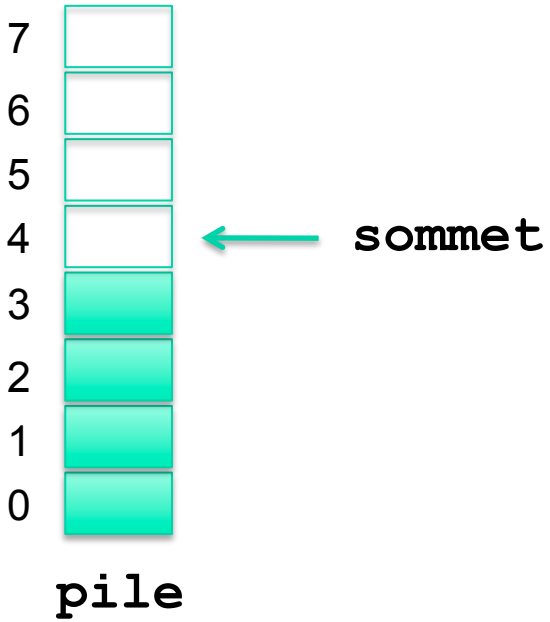
2. Les Piles

last in, first out (dernier arrivé, premier traité)

Aisément représentable par :

- un tableau
- et une variable désignant le sommet

```
typedef … typeInfo ; /* à définir */
#define MAX 100
typeInfo pile[MAX];
int sommet ;
```



Pile dans un tableau :

```
#define MAX 100
typeInfo pile[MAX];
int sommet ;
```

1. initialisation d'une pile
2. pile vide *(exp logique)*
3. pile pleine *(exp logique)*
4. empiler *(ajout)*
5. dépiler *(suppression)*

Ex : lecture et évaluation d'une expression avec une pile de calcul

```
float val[10];          pile opérande
char op[10];            pile opérateur
int sommetVal=0, sommetOp=0 ;

int priorite(char operateur);
```

opérateurs : `= + - * / ^`

priorités : `0 1 1 2 2 3`

Gestion des piles pendant l'évaluation : (ex. avec `4+2*3^2-5=`)

si l'opérateur courant est prioritaire sur celui du sommet de pile

- *empiler l'opérateur courant et l'opérande qui suit*
- *lire l'opérateur suivant*

sinon

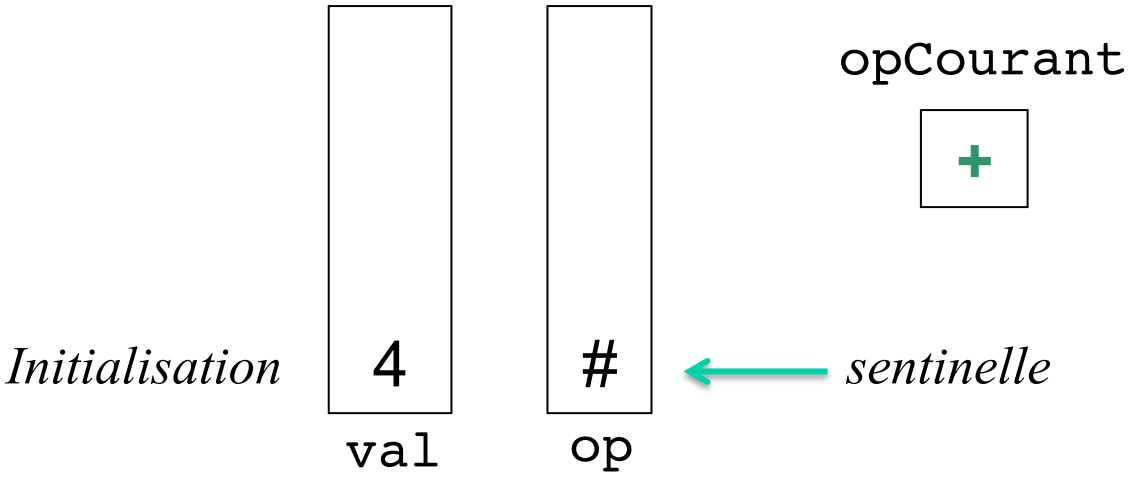
- *dépiler un opérateur et les 2 derniers opérandes*
- *exécuter l'opération et empiler le résultat*

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

4+2*3^2-5 = ?



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

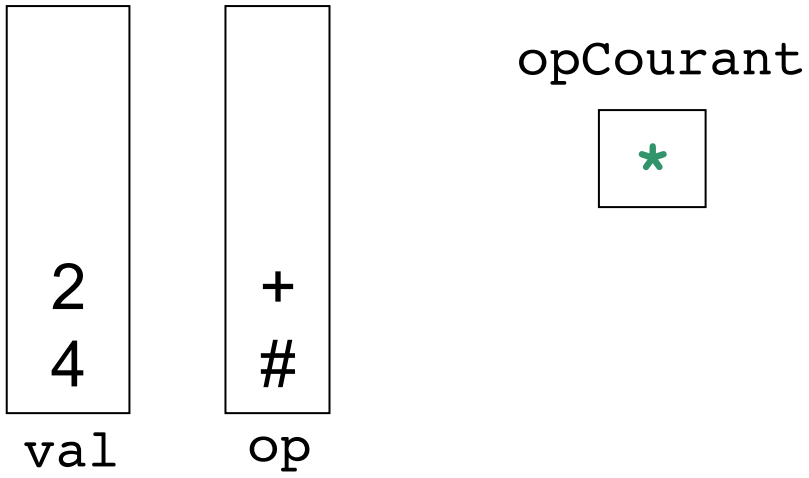
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

4+2*3^2-5 = ?



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

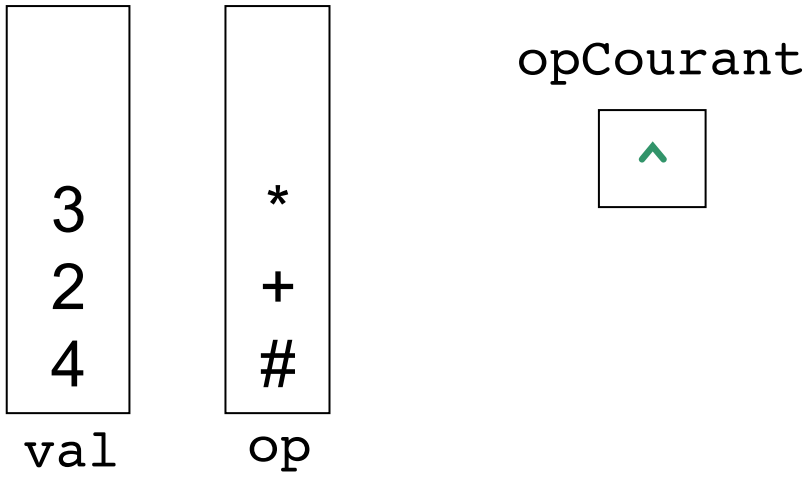
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

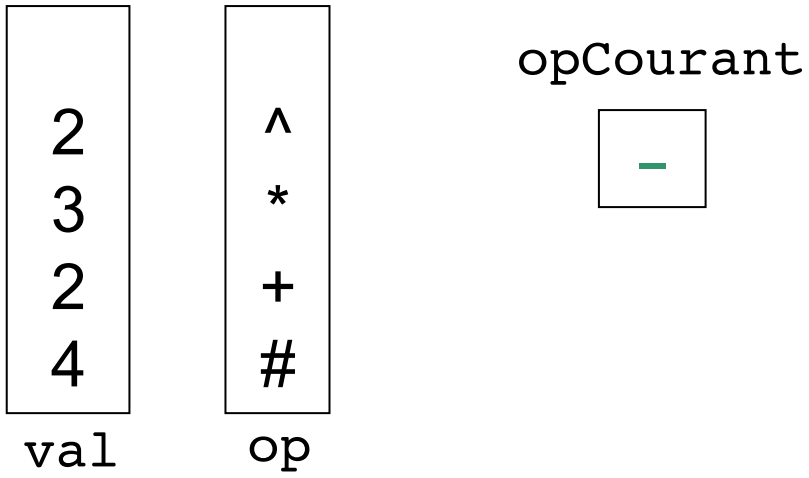
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

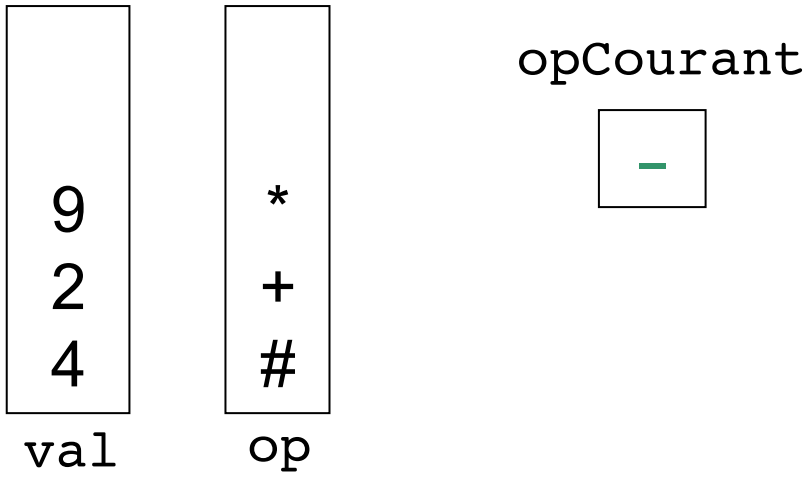
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

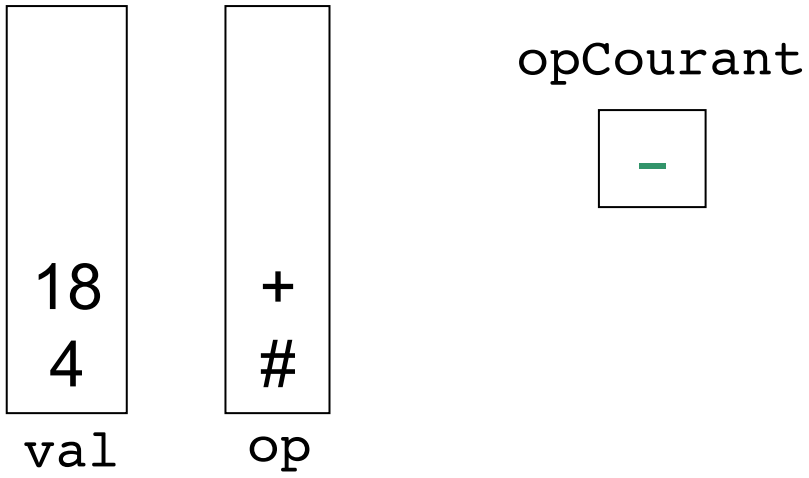
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

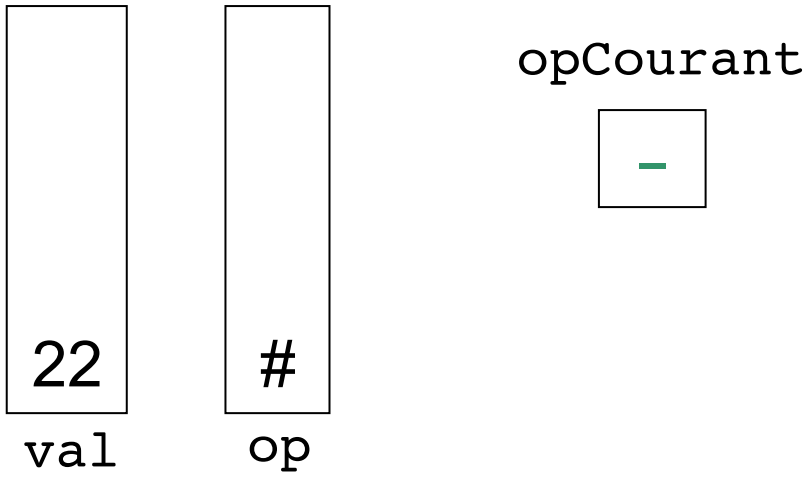
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

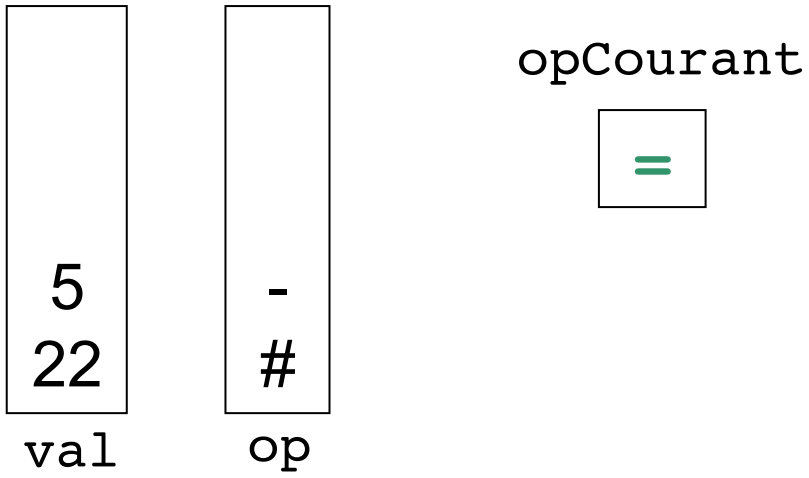
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

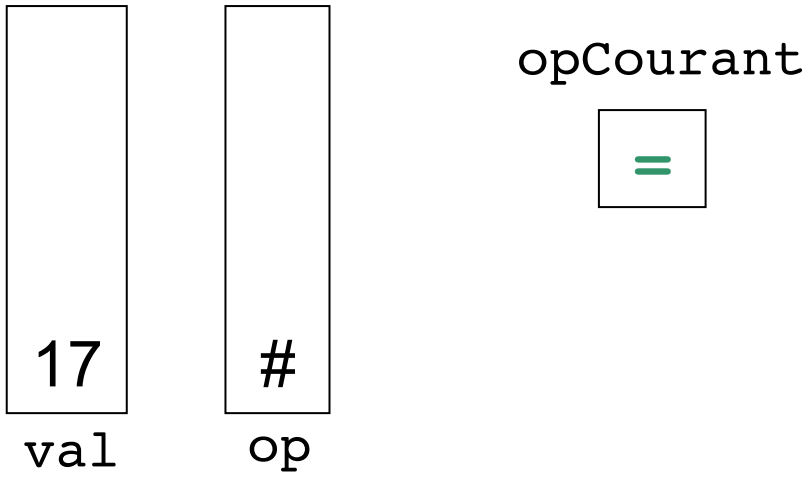
- exécuter l'opération et empiler le résultat

fin si

opérateurs : `= + - * / ^` *# (sentinelle)*

priorités : `0 1 1 2 2 3` `0`

`4+2*3^2-5 = ?`



si l'opérateur courant est prioritaire sur celui du sommet de pile

- empiler l'opérateur courant et l'opérande qui suit

- lire l'opérateur suivant

sinon

- dépiler un opérateur et les 2 derniers opérandes

- exécuter l'opération et empiler le résultat

fin si

Pile avec allocation dynamique :

```
typedef struct doublet
{ typeInfo info;
  struct doublet *suiv; // structure récursive
} Doublet,*Pdoublet ;

Pdoublet sommet;
```

```
Pdoublet allouer(typeInfo i, Pdoublet s);
void initPile(void);
int pileVide(void);
void empiler(typeInfo i);
typeInfo depiler(void);
```
(retourne l'information stockée)

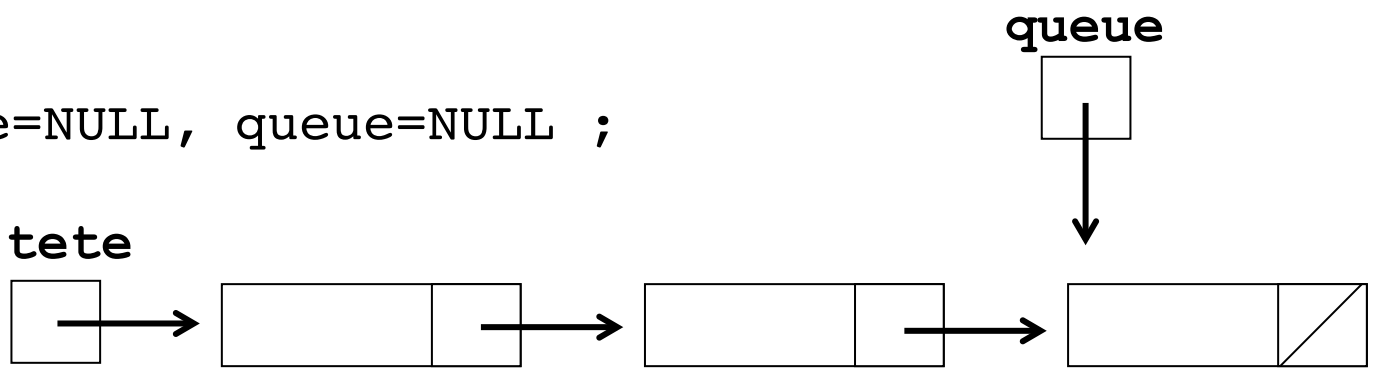
3. Les files

first in, first out (premier arrivé, premier traité)

Aisément représentable par des structures allouées dynamiquement :

- une **tête** (adresse du premier élément)
- une **queue** (adresse du dernier élément)

```
Pdoublet tete=NULL, queue=NULL ;
```




```
void initFile(void);
int fileVide(void);
void ajouter(typeInfo i);
typeInfo supprimer(void);    // retourne l'information stockée
```

```
void initFile(void)
{ tete = queue = NULL;
}

int fileVide(void)
{ return tete == NULL ;
}

void ajouter(typeInfo i)
{ if (fileVide())
   tete=queue=allouer(i, NULL);
  else
  { queue->suiv=allouer(i, NULL);//acrocher le doublet
    queue=queue->suiv; // désigner le doublet créé
  }
}
```

```
int supprimer(typeInfo *i)
{ Pdoublet p;
  if (fileVide()) return 0;
  *i = tete->info ;
  if (tete == queue) // 1 seul élément
  { free(tete);
    tete=queue=NULL;
  }
  else
  { p = tete;
    tete = tete->suiv;
    free(p); // restitution en dernier
  }
  return 1 ;
}
```

4. Les listes chaînées

Aisément représentable par des structures allouées dynamiquement :

- une **tête** (adresse du premier élément) `Pdoublet tete;`
- une relation d'ordre `infoCmp(info1, info2)`
- un pointeur courant désignant un maillon `Pdoublet Pcourant;`

Remarques: prenons le cas de fiches étudiants classées dans l'ordre alpha

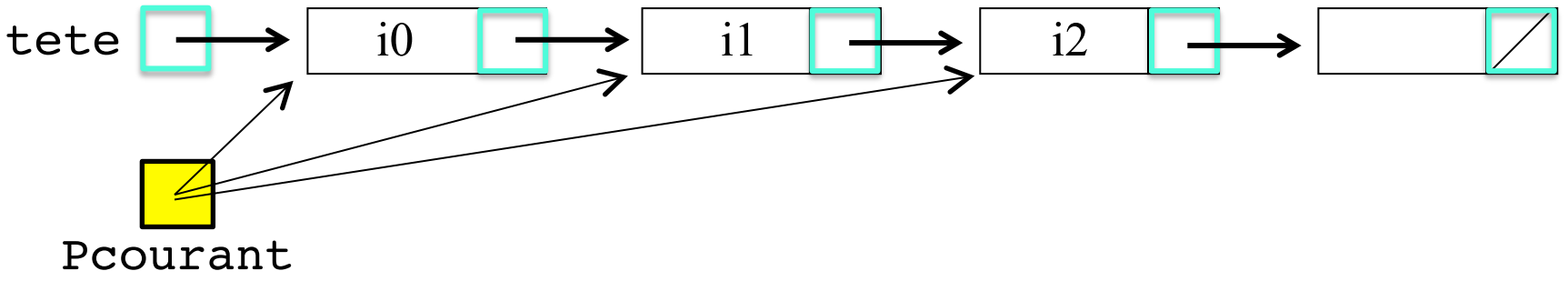
- La relation d'ordre peut ne porter que sur une partie de l'information
- On proposera une fonction **`infoCmp`** qui retourne -1, 0 ou 1 suivant que **`info1`** est inférieur à **`info2`**, égal ou plus grand.
- Toutes les opérations d'ajout/suppression se feront dans la liste en respectant cette relation d'ordre
- La recherche d'un élément se fera sur l'égalité du même critère qui a servi à la relation d'ordre (le nom pour nos fiches étudiants)

Variable globale :

- `Pdoublet tete;`

- Fonction définissant la relation d'ordre

  `int infoCmp(typeInfo info1, typeInfo info2)`




```
Pdoublet allouer(typeInfo i, Pdoublet suivant);
void initListe(void);
int listeVide(void);
void ajouter(typeInfo i);
int supprimer(typeInfo *i);
```

Initialiser :

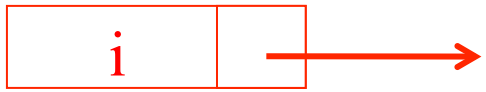
```
void initListe(void)
{… }
```

Vérifier si la liste est vide :

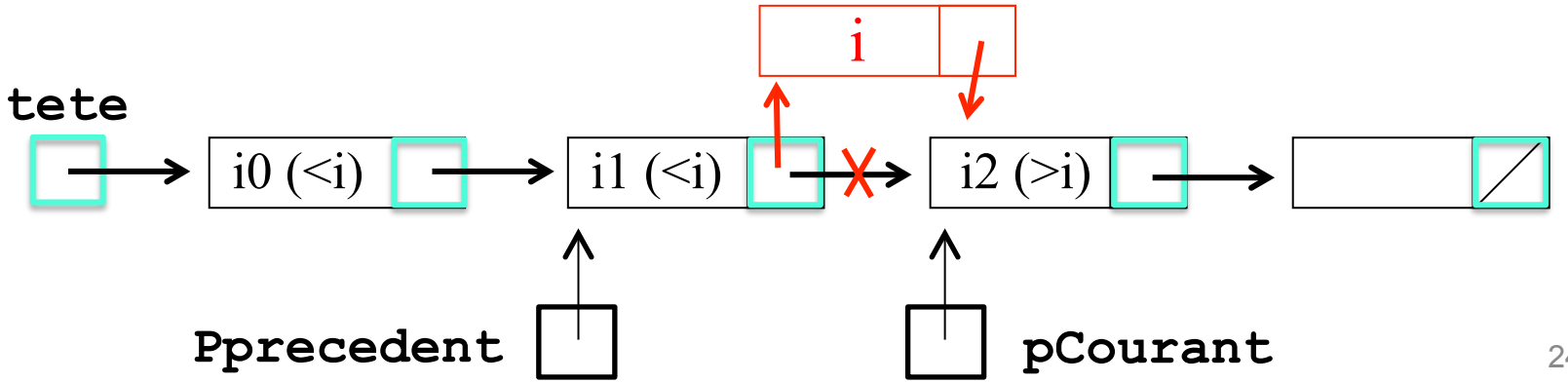
```
int listeVide(void)
{… }
```

Allouer un maillon, affecter ses champs et retourner son adresse :

```
Pdoublet allouer(typeInfo i, Pdoublet suivant)
{… }
```



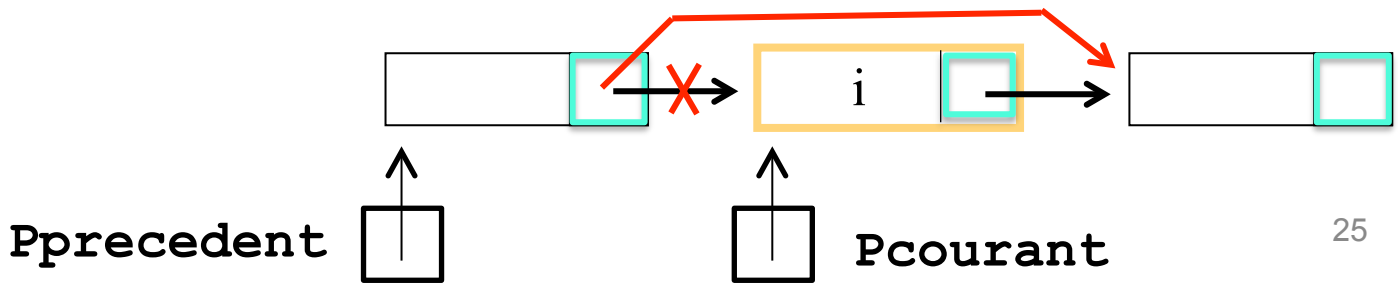
```
void ajouter(typeInfo i)
{ Pdoublet Pcourant, Pprecedent ;
  Pcourant= tete;
  // recherche du point d'insertion
  while ((Pcourant!=NULL)&&infoCmp(Pcourant->info, i)<0)
  { Pprecedent = Pcourant ;
    Pcourant= Pcourant->suiv;
  }
  if (Pcourant == tete) // insertion en tête
    tete = allouer(i, tete);
  else // insertion au milieu ou en queue
    Pprecedent->suiv = allouer(i, Pcourant);
}
```



Suppression :

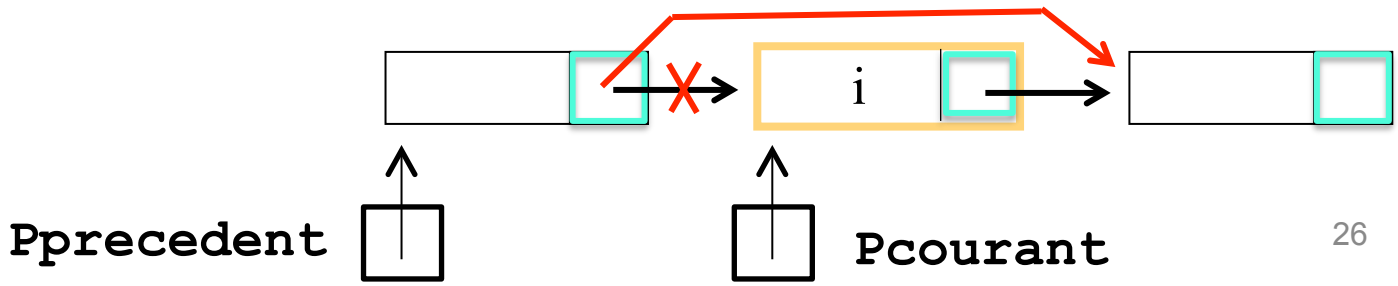
- Le champ info est passé par adresse si il est incomplet et qu'il faut le compléter avec le doublet à supprimer.
- Seule la valeur nécessaire à la relation d'ordre est obligatoire. Par exemple le nom pour une fiche étudiant.

```
int supprimer(typeInfo *i)
{ Pdoublet Pcourant, Pprecedent ;
  // recherche du doublet
  Pcourant = tete;
  while ((Pcourant!=NULL)&&infoCmp(Pcourant->info,*i)<0))
  { Pprecedent = Pcourant ; Pcourant= Pcourant->suiv; }
  // cas où il n'existe pas
  if (Pcourant == NULL) return 0;
  if (infoCmp(Pcourant->info, *i) > 0) return 0;
```

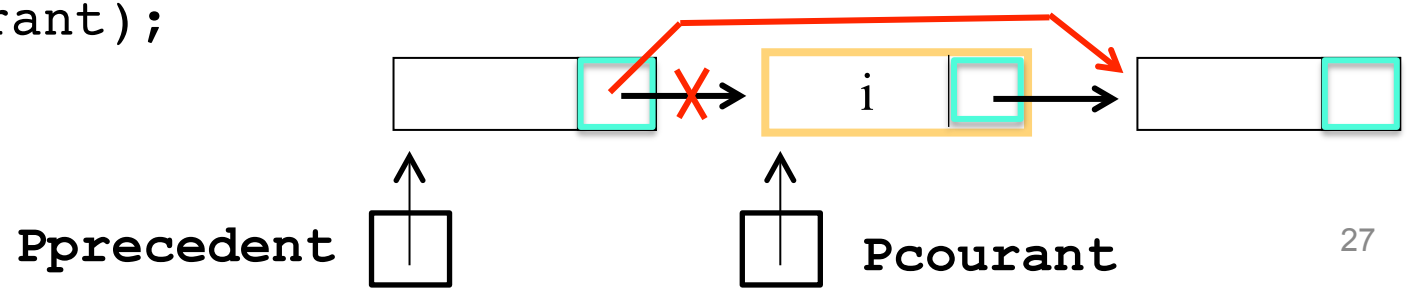


Suite :

```
   // décrocher le doublet
   if (Pcourant == tete) tete = tete->suiv; // en tête
   else Pprecedent->suiv = Pcourant->suiv;  // au milieu
   // Si *i est incomplet, par exemple ne contient que le nom de l'étudiant
   // pour la recherche,
   // on peut compléter les infos avec Pcourant->info avant de restituer
   *i = Pcourant->info;
   free(Pcourant);
   return 1 ;
}
```

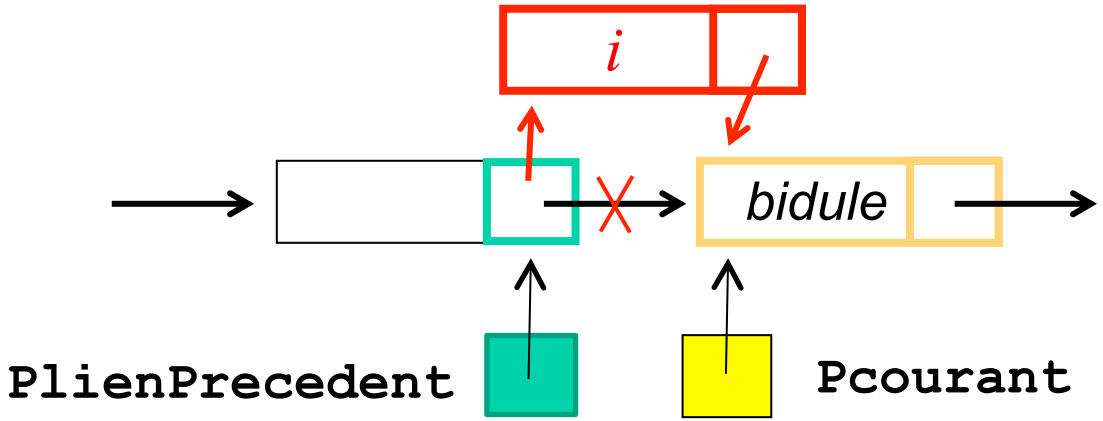


```
int supprimer(typeInfo *i)
{ Pdoublet Pcourant, Pprecedent ;
  Pcourant = tete;
  while ((Pcourant!=NULL)&&infoCmp(Pcourant->info,*i)<0))
  { Pprecedent = Pcourant ; Pcourant= Pcourant->suiv; }
  if (Pcourant == NULL) return 0;
  if (infoCmp(Pcourant->info, *i) > 0) return 0;
  if (Pcourant == tete) // suppression en tête
    tete = tete->suiv;
  else
    Pprecedent->suiv = Pcourant->suiv;
  // Si *i est incomplet, par exemple ne contient que le nom de l'étudiant,
  // on peut récupérer des infos dans Pcourant->info avant de restituer
  *i = Pcourant->info;
  free(Pcourant);
  return 1 ;
}
```



Autre solution

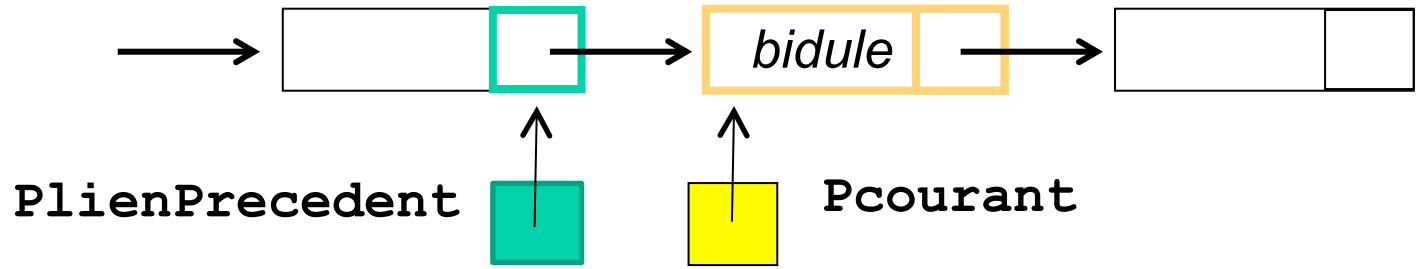
- Pointer sur le champ `suiv` du prédécesseur :
- Insérer un maillon avant le maillon courant :



Il faut garder l'adresse du lien précédent !!!

```
Pdoublet *PlienPrecedent; // pointeur de pointeur de doublet
//insertion entre le précédent et le courant
*PlienPrecedent = allouer(i, Pcourant) ;
```

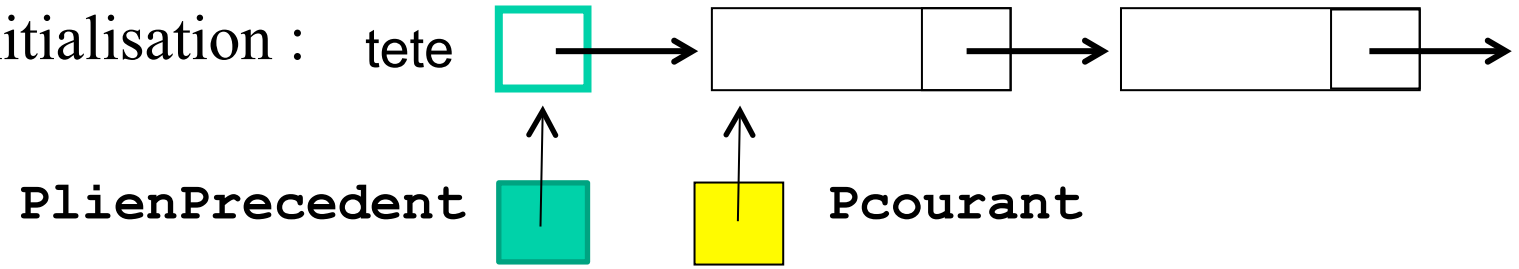
```
Pdoublet *PlienPrecedent; // pointeur de pointeur de doublet
Pdoublet Pcourant; // pointeur de doublet
```



- Se déplacer sur le maillon suivant :

```
PlienPrecedent = &(Pcourant->suiv);
Pcourant = Pcourant->suiv;
```

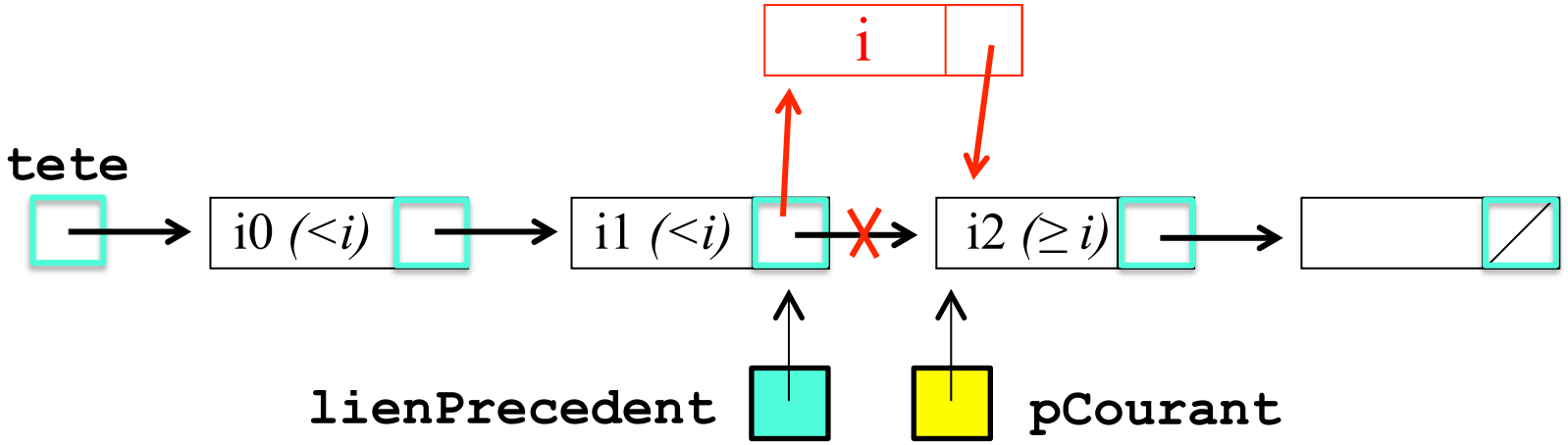
- Initialisation :




```
PlienPrecedent = &tete;
Pcourant = tete;
```

Ajouter une information dans la liste :

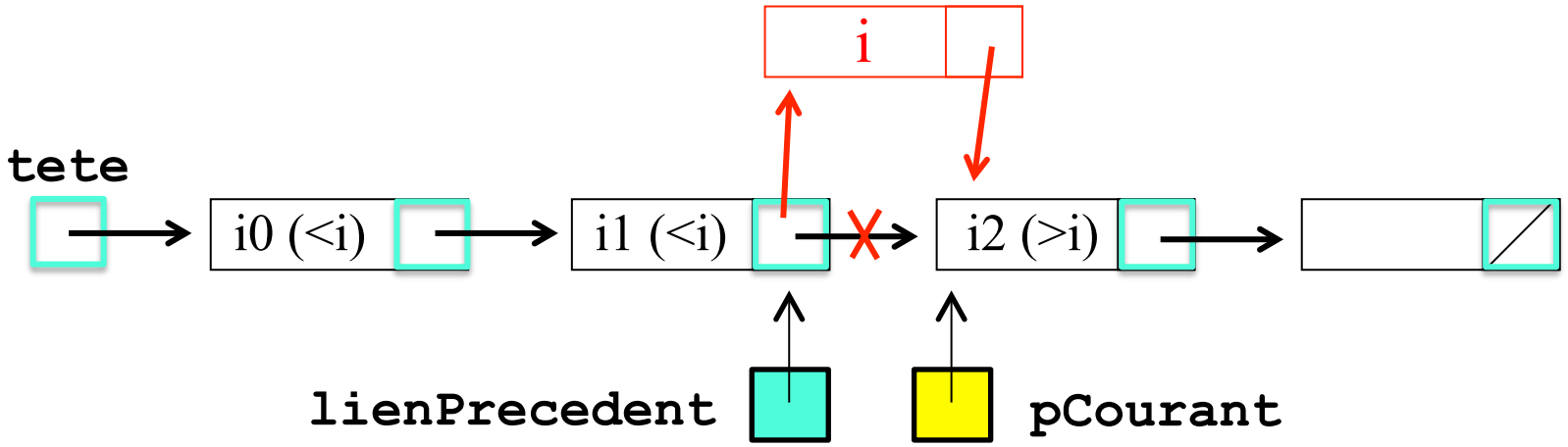
```
void ajouter(typeInfo i)
{ … }
```



- recherche de l'emplacement où insérer :
  - en partant de la tête : `lienPrecedent = &tete;`
  - en testant la fin de liste
  - et en respectant l'ordre croissant
- insérer

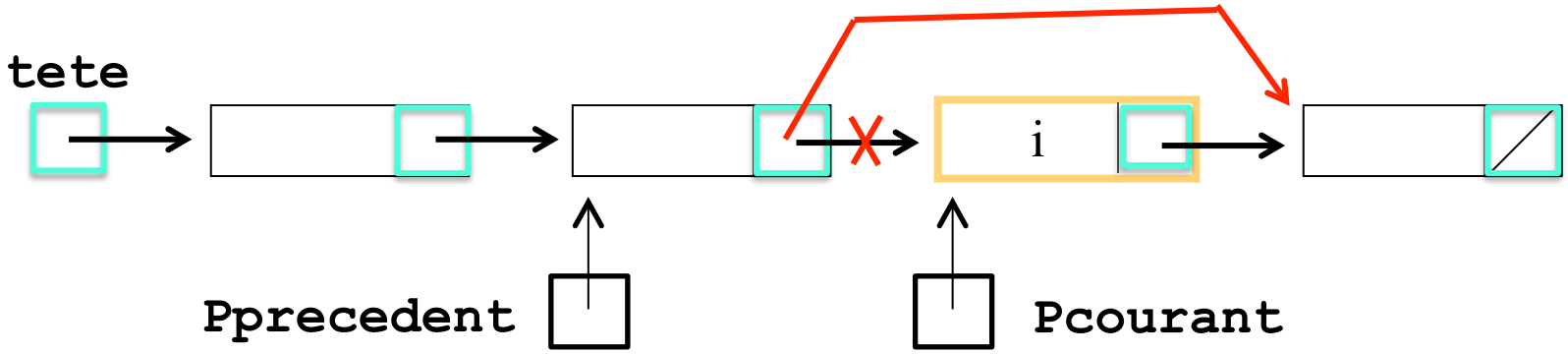
Ajouter une information dans la liste :

```
void ajouter(typeInfo i)
{ Pdoublet Pcourant, *PlienPrecedent ;
  Pcourant= tete;
  PlienPrecedent = &tete;
  while ((Pcourant!=NULL) && inf(Pcourant->info, i))
  { PlienPrecedent = &(Pcourant->suiv);
    Pcourant= Pcourant->suiv;
  }
  *PlienPrecedent = allouer(i, Pcourant);
}
```



Retirer une information de la liste :

```
int supprimer(typeInfo i)
{ … }
```



- recherche le maillon à supprimer
  - en partant de la tête
  - en testant la fin de liste (si absent)
  - et en tenant compte de l'ordre croissant
- retirer le maillon : `Pprecedent->suiv = Pcourant->suiv;`
- restituer le maillon au système : `free(Pcourant);`