Les structures séquentielles chaînées

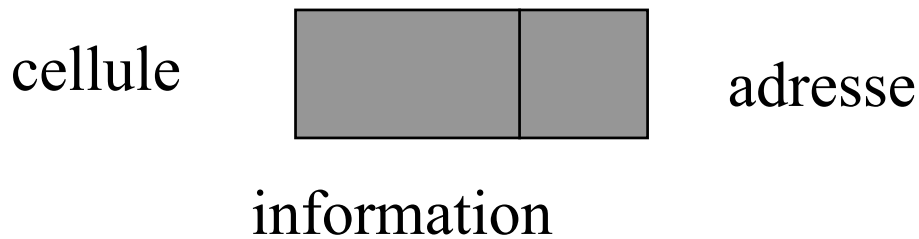
Isabelle Comyn-Wattiau

- Définitions
- Création d'une liste
- Parcours d'une liste
- Accès à un élément d'une liste
- Mises à jour d'une liste
- Conclusion sur les listes chaînées

Définitions

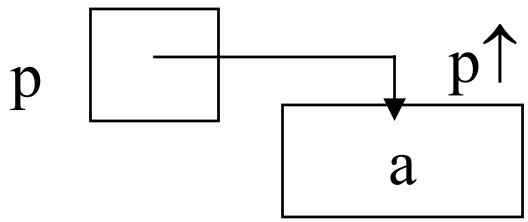
Définition

- Chaque élément indique l'adresse de l'élément suivant



- L'information peut être simple ou structurée

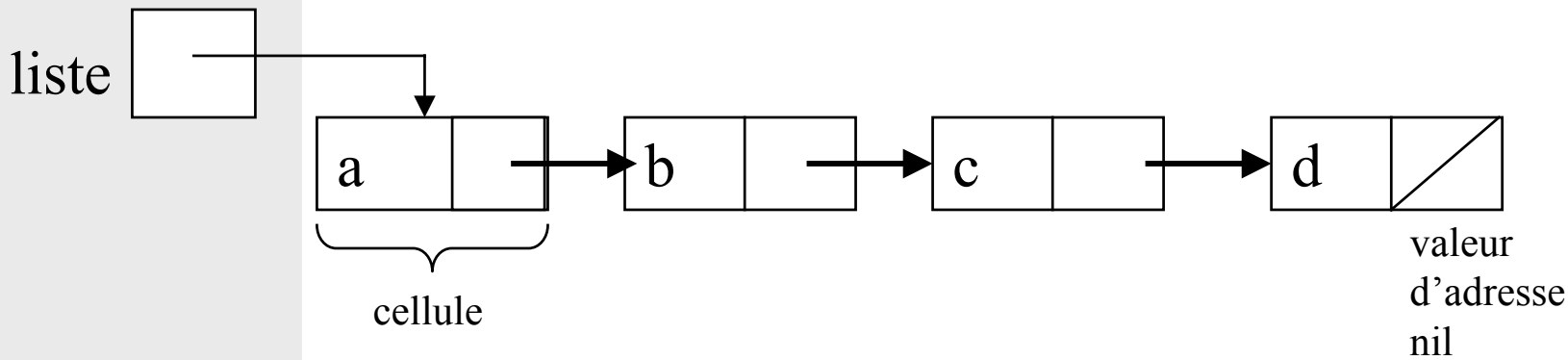
Variables de type pointeur



p est une variable de type pointeur

p↑ est l'objet dont l'adresse est rangée dans p

Liste linéaire chaînée

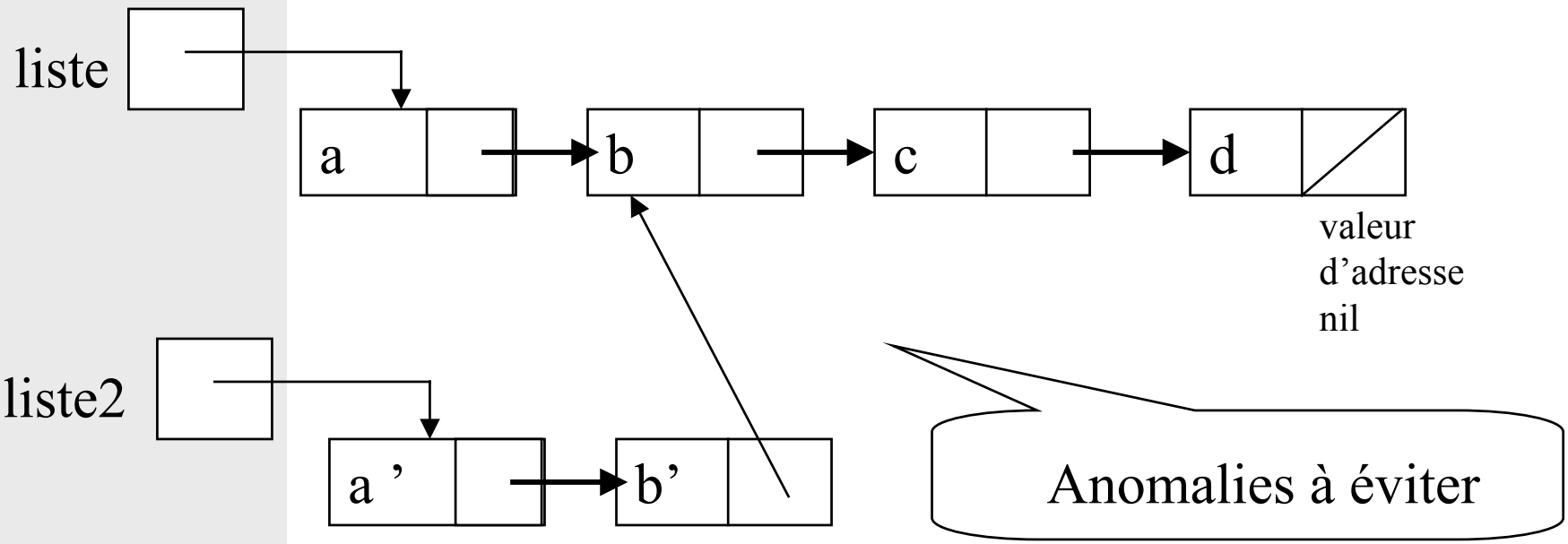


La liste est définie par l'adresse de sa première cellule

```
Type cellule is record
                info:element;
                suivant:pointeur;
end record;
Type pointeur = ↑ cellule;
```

Si p <> nil, p↑.info et p↑.suivant permettent d'accéder à la cellule sur laquelle pointe p

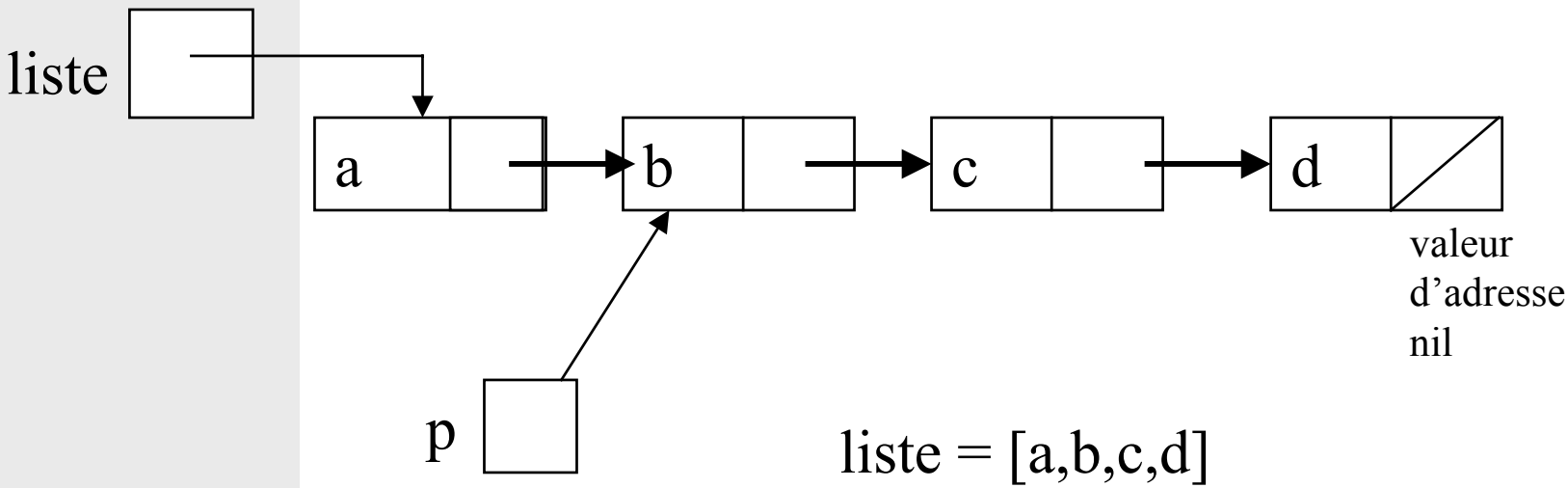
Attention aux ambiguïtés



Allocation dynamique de la mémoire

- L'organisation chaînée permet d'utiliser les procédures d'allocation et libération dynamique de mémoire
  - allouer(p) : allouer un espace mémoire à une cellule sur laquelle pointe p
  - libérer(p) : libérer l'espace mémoire occupée par la cellule sur laquelle pointe p

Représentation récursive des listes



liste = [a,b,c,d]

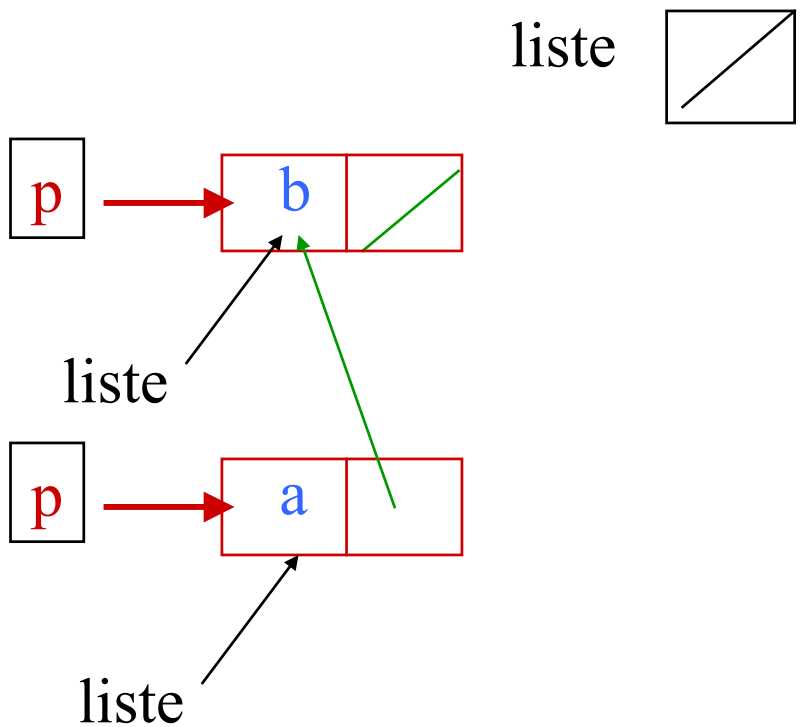
liste = [a | p]

Notation récursive : liste= [↑liste.info | ↑liste.suivant]

Création de liste

Algorithme de création d'une liste [a,b]

```
liste:=nil;
allouer(p);
p↑.info:=b;
p↑.suivant:=nil;
liste:=p;
allouer(p);
p↑.info:=a;
p↑.suivant:= liste;
liste:=p;
```



Algorithme de création d'une liste à partir du contenu d'un fichier

```
procedure creerliste(f : fichier de element in; liste : pointeur out);
p:pointeur; c:element;
début
liste:=nil; ouvrir(f);
tant que non eof(f) faire
        allouer(p); lire(f,c);
        p↑.info:=c; p↑.suivant:=liste;liste:=p;
fin tant que;
fin
```

Parcours de liste

Parcours d'une liste chaînée

- Une liste chaînée est :
  - soit vide : liste = nil
  - soit composée d'une cellule suivie d 'une liste (définition récursive)

Parcours récursif de gauche à droite

```
procedure parcoursgd (liste:pointeur);
début
        si liste < > nil alors
                traiter(liste);
                parcoursgd(liste↑.suivant);
        finsi;
fin
```

Parcours non récursif de gauche à droite

```
procedure parcoursgdnonrec (liste:pointeur);
liste1: pointeur;
début
        liste1:= liste;
        tant que liste1 < > nil faire
                traiter (liste1);
                liste1:=liste1↑.suivant);
        fintantque;
fin
```

Parcours récursif de droite à gauche

```
procedure parcoursdg (liste:pointeur);
début
        si liste < > nil alors
                parcoursdg(liste↑.suivant);
                traiter(liste);
        finsi;
fin
```

Les accès à une liste

Accès itératif au k-ième dans une liste (par position)

```
procedure accesposition (liste:pointeur in; k:entier in;
      pointk : pointeur out; trouvé: booléen out);
{(trouvé et pointk=adresse du k-ième élément) ou (trouvé faux et pointk indéfini)}
liste1:pointeur; i:entier;
début
liste1:=liste; i:=1;
tant que (i<k) et (liste1< > nil) faire
            i:=i+1; liste1 := liste1↑.suivant
fin tant que
trouvé:= (i=k) et (liste< >nil);
pointk:= liste1;
fin
```

Accès récursif au k-ième dans une liste (par position)

```
fonction accesposrec (liste:pointeur; k:entier):pointeur;
      {résultat=adresse du k-ième élément s 'il existe, nil sinon}
début
si liste=nil
          alors retourner(nil)
          sinon
          si k=1
                    alors retourner(liste)
                    sinon retourner accesposrec(liste↑.suivant,k-1)
          finsi
finsi
fin
```

OU BIEN
si (liste=nil) ou (k=1)
alors retourner (liste)

Principe : le k-ième dans la liste [T|Q] est le(k-1)-ième dans la liste Q

Accès itératif à la place contenant val (par association)

```
procedure accesassociter (liste:pointeur in; val:element in;
      point : pointeur out; trouvé: booléen out);
{(trouvé et point=adresse du 1er val) ou (trouvé faux et point nil et val n'est pas
dans liste)}
liste1:pointeur;
début
liste1:=liste;
tant que (liste1< > nil)et (liste1↑.info < > val) faire
            liste1 := liste1↑.suivant
fin tant que;
trouvé:=liste1< >nil;
point:= liste1;
fin
```

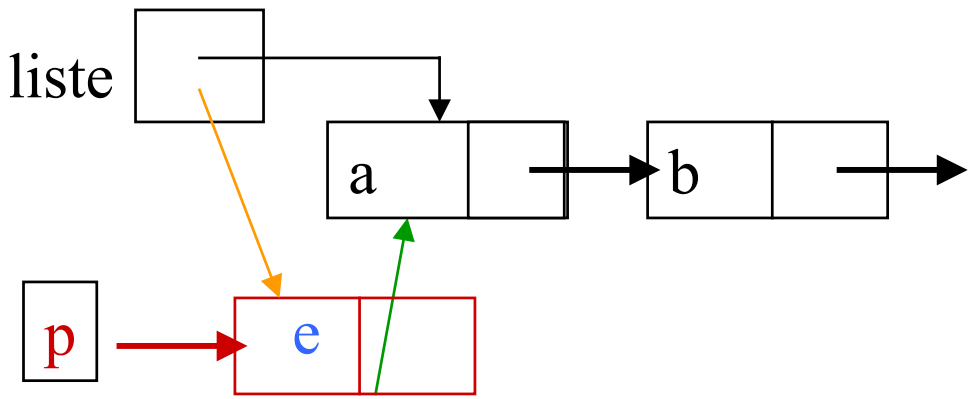
Accès récursif dans une liste par association

```
fonction accesassocrec (liste:pointeur; val:element):pointeur;
      {résultat=adresse de la 1e occurrence de val et val est dans la liste ou
       résultat=nil et val n'est pas dans la liste}
début
si liste=nil
            alors retourner(nil)
            sinon
            si liste↑.info = val
                        alors retourner(liste)
                        sinon retourner accesassocrec(liste↑.suivant,val)
            finsi
finsi
fin
```

Mises à jour d’une liste

Insertion de l'élément en tête de la liste

```
allouer(p);
p↑.info:=e;
p↑.suivant:=liste;
liste:=p;
```



insertiontete(liste, e)

Elément « e » à insérer dans la liste « liste »

Insertion d'un élément en fin de liste

Elément « e » en fin de liste « liste »

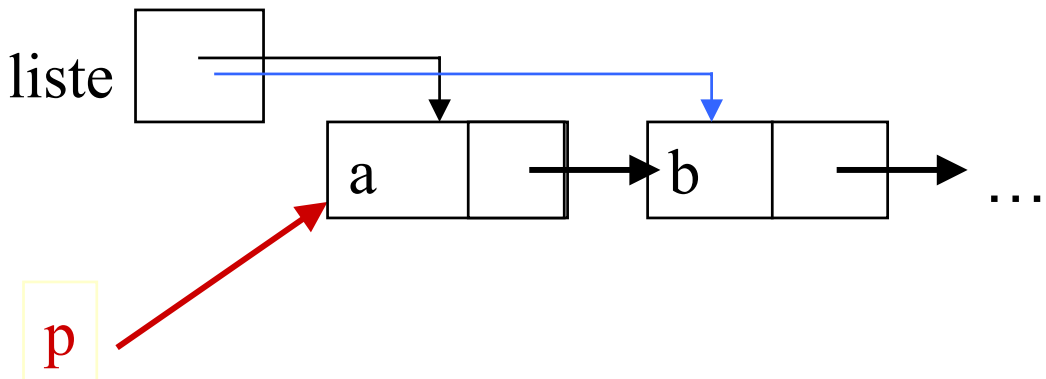
```
si liste = nil
        alors insertiontete (liste,e)
        sinon
                der:=dernier(liste);
                allouer(p);
                p↑.info:=e;
                p↑.suivant := nil;
                der↑.suivant :=p
finsi
```

```
fonction dernier
(liste:pointeur):pointeur;
      p, précédent : pointeur;
début
p:=liste;
tant que p < > nil faire
            précédent :=p;
            p:= p↑.suivant;
fin tant que
retourner(précédent);
fin
```

Suppression de l'élément en tête de liste

supprimetete(liste)

```
p:=liste;
liste:=liste↑.suivant;
libérer(p);
```

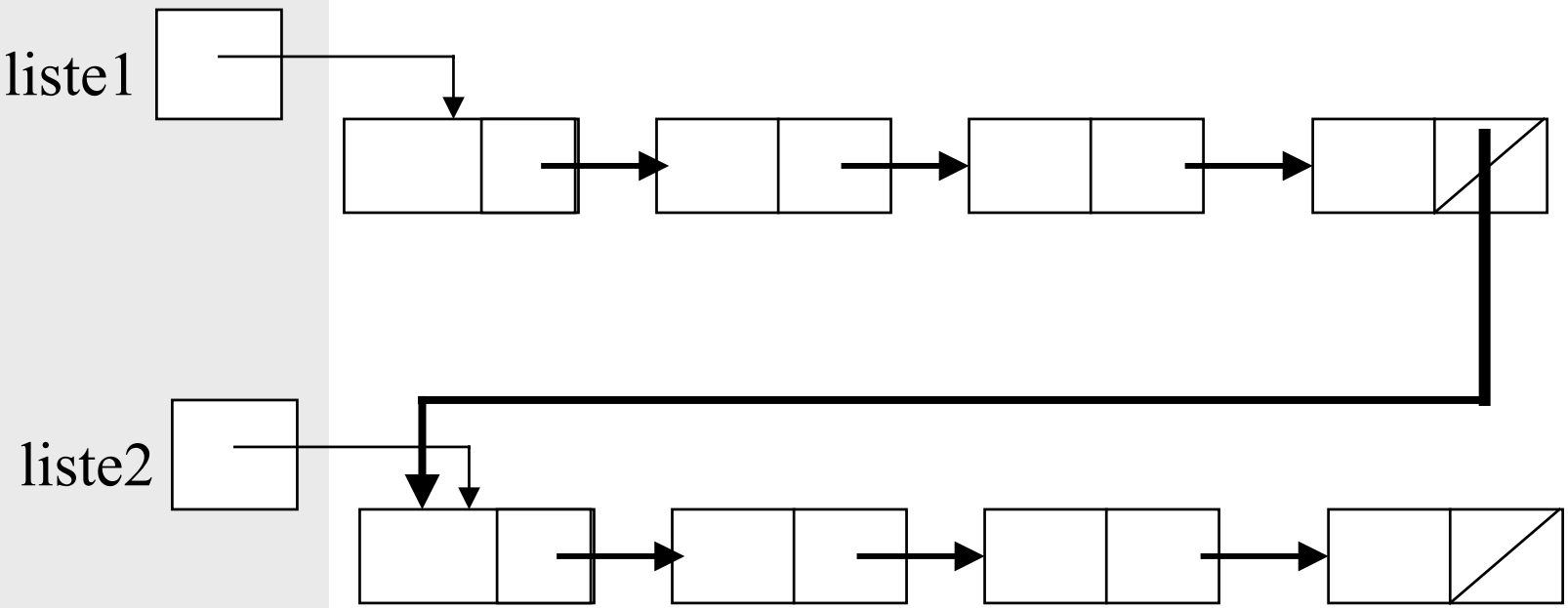


Il faut préserver l'adresse de la tête
pour pouvoir ensuite libérer l'espace

Suppression du k-ième élément d'une liste chaînée

```
procedure supprimerk (liste:pointeur in out; k:entier in; possible: booléen out);
{(possible et on supprime le k-ième) ou (possible faux et on ne supprime rien)}
pointk,précédent:pointeur;
si (liste < > nil) et (k=1)
        alors possible := vrai; supprimetete(liste);
        sinon
                possible:=faux; précédent := accesposrec(liste, k-1);
                si précédent < > nil
                alors   pointk := précédent↑.suivant;
                        si pointk < > nil
                                alors
                                  possible := vrai;
                                  précédent↑.suivant := pointk↑.suivant;
                                  libérer(pointk);
                        finsi
                finsi
finsi
fin
```

Concaténation de deux listes



Concaténation de deux listes

```
procédure concat (liste1:pointeur in; liste2 pointeur in ; liste:pointeur out);
      der:pointeur;
début
si liste1=nil
            alors liste:=liste2
            sinon
            liste:=liste1;
            si liste2< > nil alors
                              der:=dernier(liste1);
                              der↑.suivant:=liste2;
            finsi
finsi
fin
```

Conclusion

Listes chaînées particulières

- On peut insérer une tête fictive (ou sentinelle) pour faciliter l'écriture des opérations de manipulation de la liste
- On peut introduire un double chaînage (liste bidirectionnelle) pour autoriser un parcours dans les deux sens
- Dans certains cas, une liste circulaire ou anneau est recommandée

Conclusion sur les listes chaînées

- Structure orientée vers les traitements séquentiels
- Peut être manipulée de façon itérative ou de façon récursive
- L'implantation chaînée permet des ajouts et des suppressions sans déplacement
- Mais l'accès par position est proportionnel à la longueur de la liste
- Le coût d'un algorithme est évalué en place occupée et en nb de pointeurs parcourus ou affectés
- A utiliser chaque fois que les mises à jour sont plus importantes que les consultations

Deux listes particulières : les piles et les files

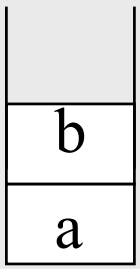
Isabelle Comyn-Wattiau

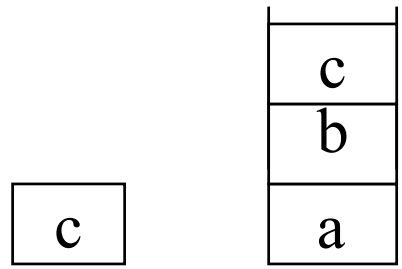
Les piles

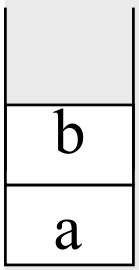
- Ce sont des listes particulières
- Le dernier élément ajouté est au sommet
- C'est le seul accessible immédiatement : LIFO
- Stockage très temporaire
- Analogie avec la pile de livres

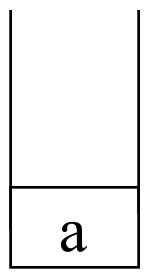
Primitives de manipulation

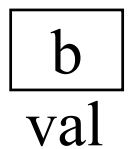
- On peut manipuler une pile avec seulement deux primitives :
  - empiler(val) ajoute l'élément val au sommet de la pile
  - dépiler(val) supprime l'élément au sommet de la pile et range sa valeur dans val











Représentation contiguë d'une pile

- La pile est un vecteur
- Elle a une taille maximale
- L'accès au premier élément est un indice
- Pour tester les limites de la pile, on teste la valeur de l'indice
- En général, la tête de liste est l'indice le plus élevé, la base est à 1

Représentation contiguë (suite)

Tableau pile de taille maximum pilemax
On suppose qu'on a 3 variables globales pile, pilemax et sommet
sommet est l'indice de remplissage de la pile

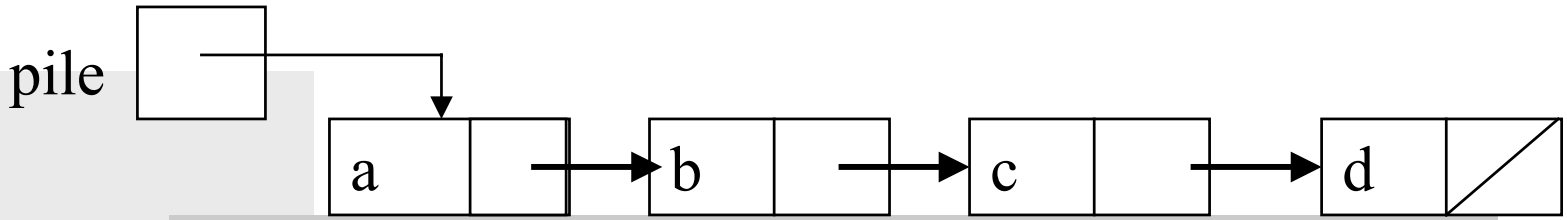
```
fonction pilevide:booléen;
début
        retourner(sommet=0);
fin;
```

```
fonction pilepleine:booléen;
début
        retourner(sommet=pilemax);
fin;
```

Représentation contiguë (suite)

```
procédure empiler(val:élément in);
 {val est empilé sauf si la pile est pleine}
début
si pilepleine
            alors message d'erreur
            sinon sommet:=sommet + 1;
                        pile[sommet]:=val;
finsi;
fin
```

```
procédure dépiler(val:élément out);
 {val est dépilé sauf si la pile est vide}
début
si pilevide
            alors message d'erreur
            sinon sommet:=sommet - 1;
                        val:=pile[sommet];
finsi;
fin
```



Représentation chaînée d'une pile

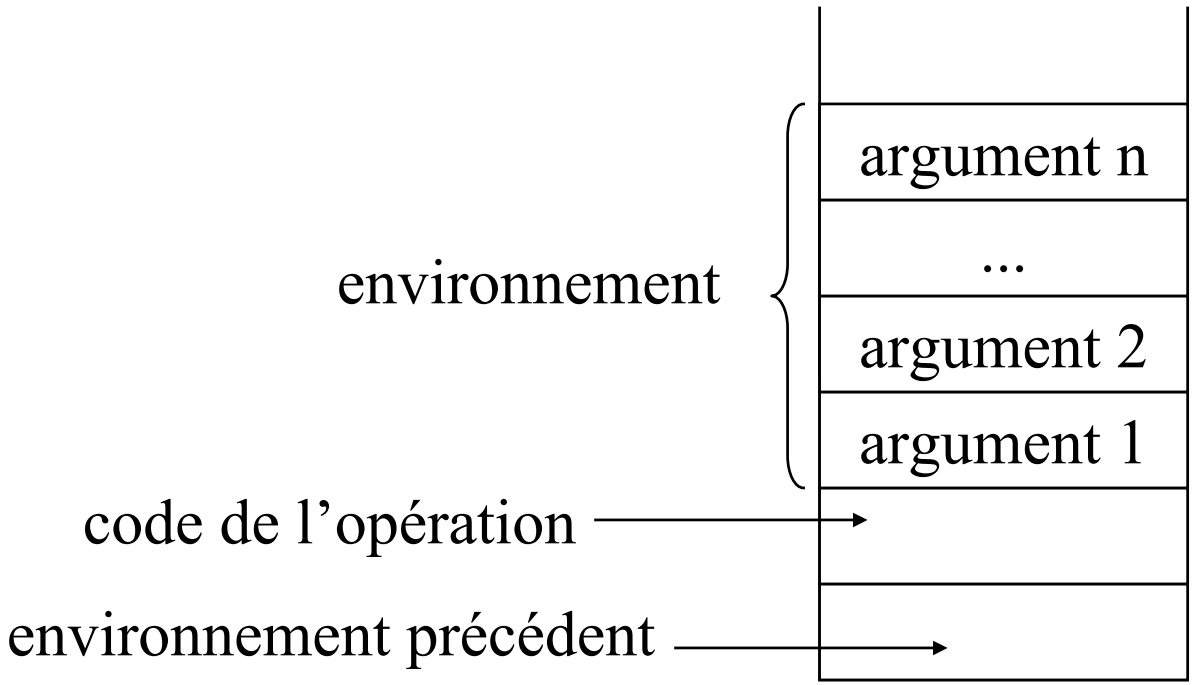
Les insertions et les suppressions sont toujours effectuées en tête

```
fonction pilevide:booléen;
début
        retourner(pile=nil);
fin
```

```
procédure empiler(val:élément in);
début
insertiontete(pile,val)
fin
```

```
procédure dépiler(val:élément out);
 {val est dépilé sauf si la pile est vide}
début
si pilevide
        alors   message d'erreur
        sinon   val:=pile↑.info;
                supprimetete(pile);
finsi;
fin
```

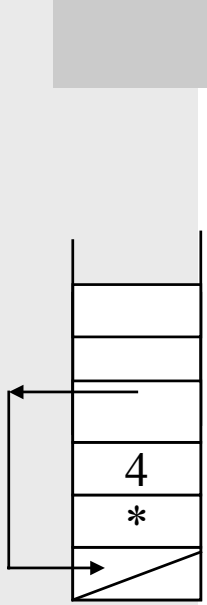
Une application des piles : l'évaluation des fonctions récursives



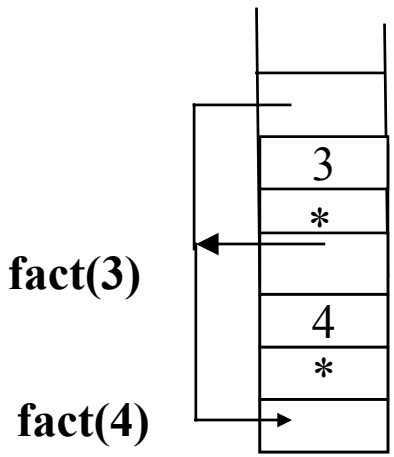
Exemple : le calcul de factorielle

```
fonction fact(n:entier):entier;
début
        si n > 1
                alors
                retourner n * fact(n-1);
                sinon
                retourner(1);
        finsi;
fin
```

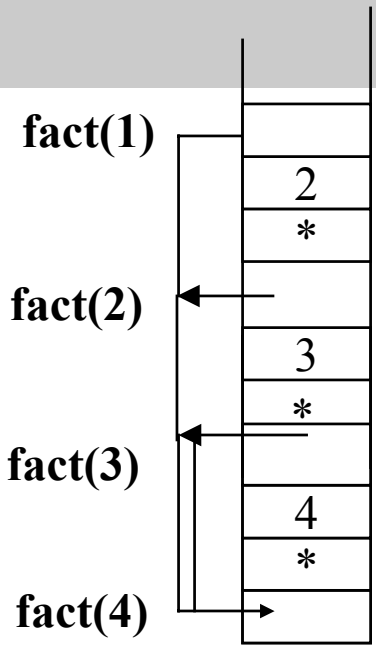
Exemple : le calcul de factorielle



fact(4)=4*fact(3)

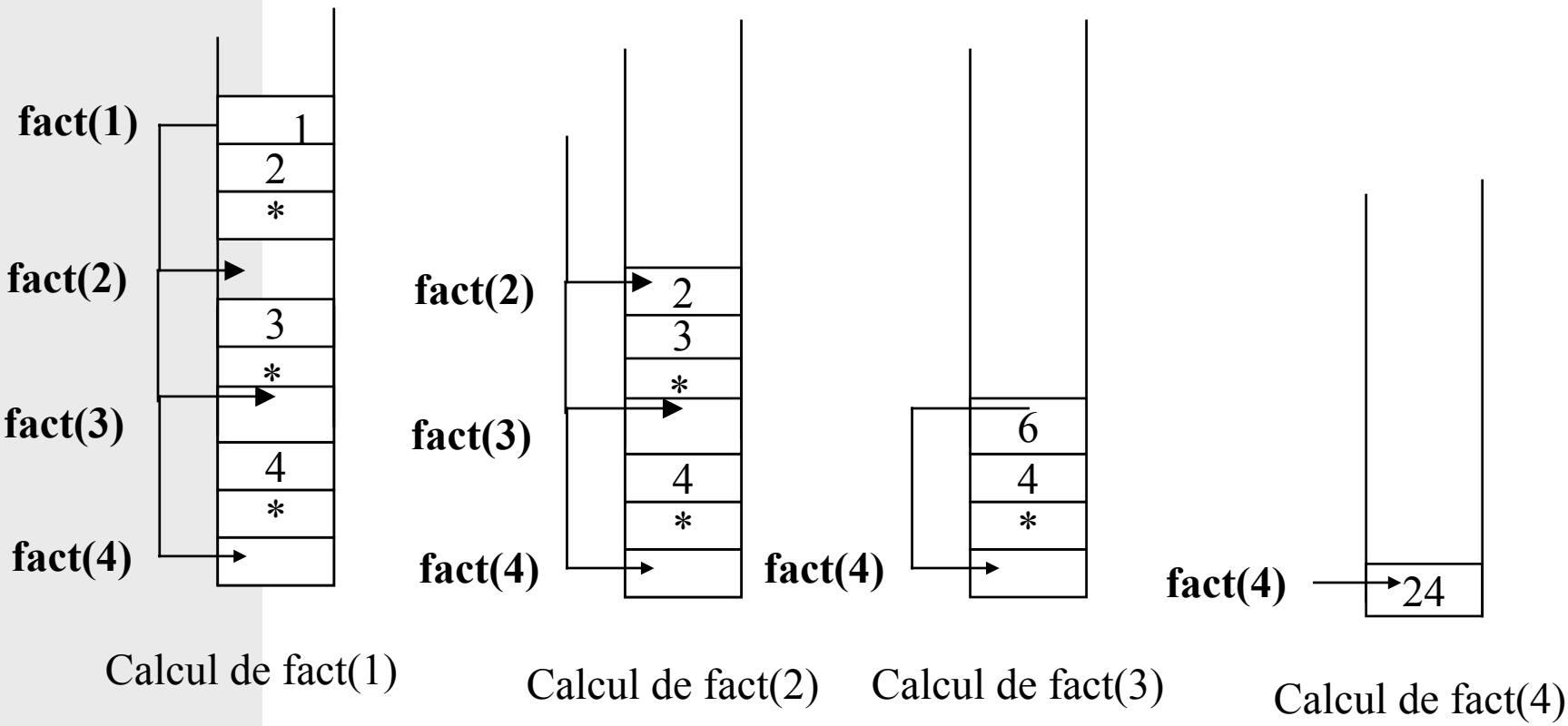


fact(3)=3*fact(2)



fact(2)=2*fact(1)

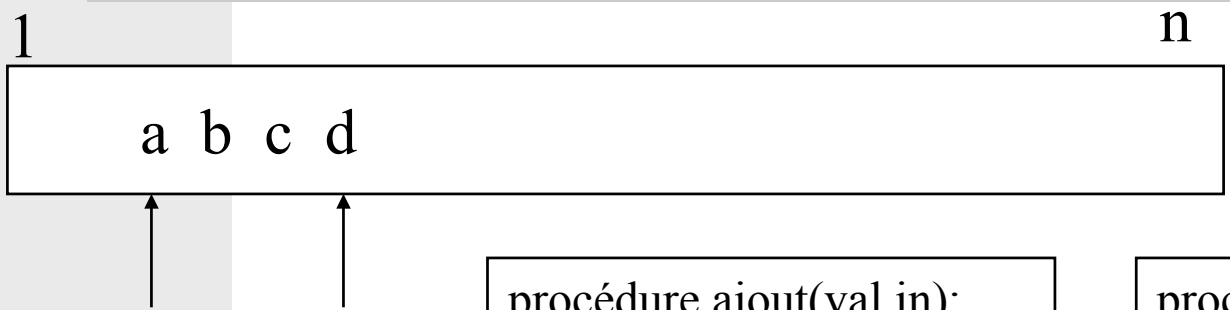
Exemple : le calcul de factorielle



Les files

- A l’image des files d ’attente
- FIFO
- Liste linéaire avec insertion en fin et suppression en tête
- Organisation contiguë ou chaînée

Files en représentation contiguë



file[1..n]
deux entiers tête et queue

```
procédure ajout(val in);
début
si non filepleine
            alors
            queue:=queue+1;
            file[queue]:=val;
finsi
fin
```

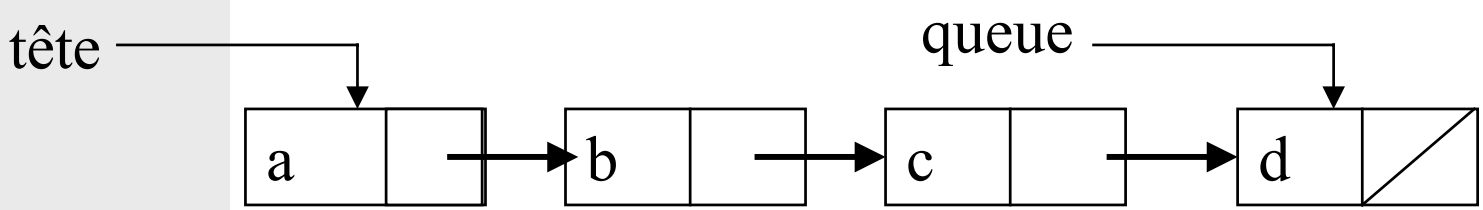
```
procédure supp(val out);
début
si non filevide
            alors
            val:=file[tête];
            tête:=tête+1;
finsi
fin
```

Tête et queue augmentent toujours : pb de taille du tableau

Alternatives de file en représentation contiguë

- On décale les éléments de la file vers la gauche à chaque suppression
  - mais pb de performance
- On gère le vecteur de manière circulaire

Représentation chaînée d'une file



- Liste chaînée classique avec un pointeur « queue »