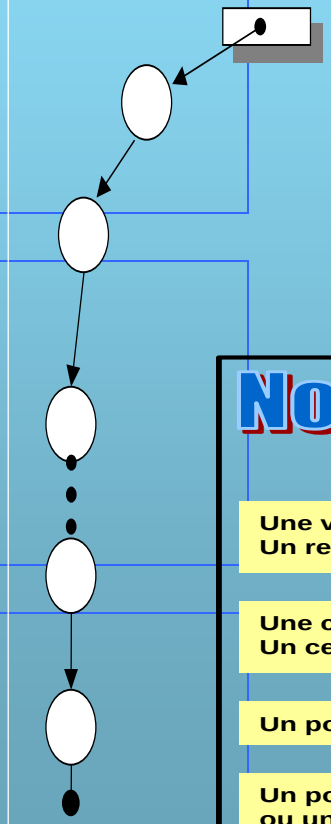
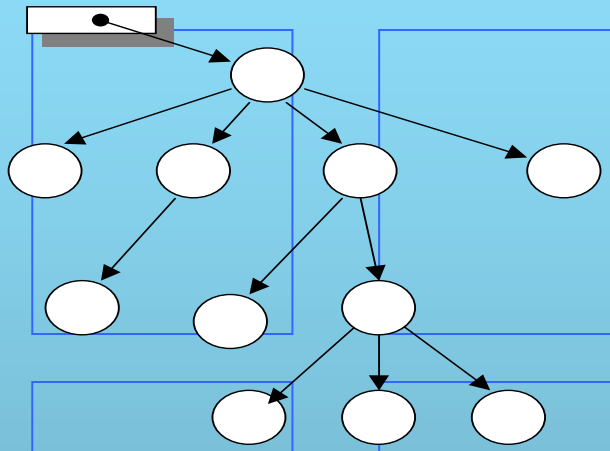
ALGORITHMIQUES

Structures de données élémentaires

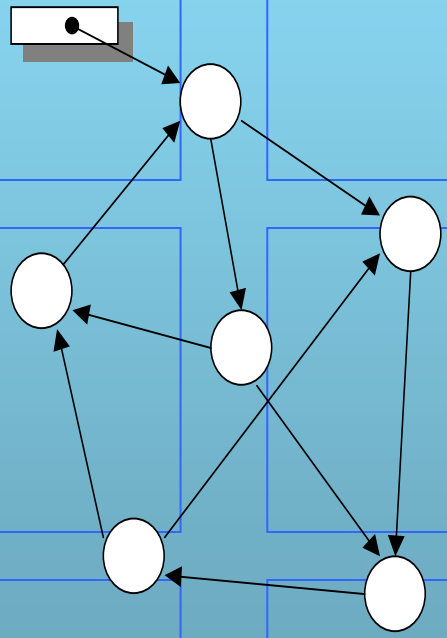
Structures linéaires



Structures arborescentes



Réseaux



Notation

Une variable contenant un pointeur est représenté par
Un rectangle

Une cellule de donnée quelconque est représentée par
Un cercle

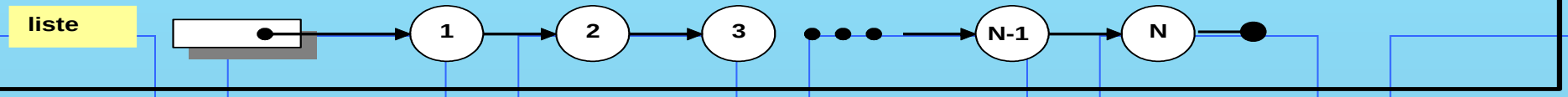
Un pointeur est représenté par une flèche

Un pointeur nul (NIL) est représenté par une flèche NIL
ou une absence de flèche

Différents types de structures linéaires

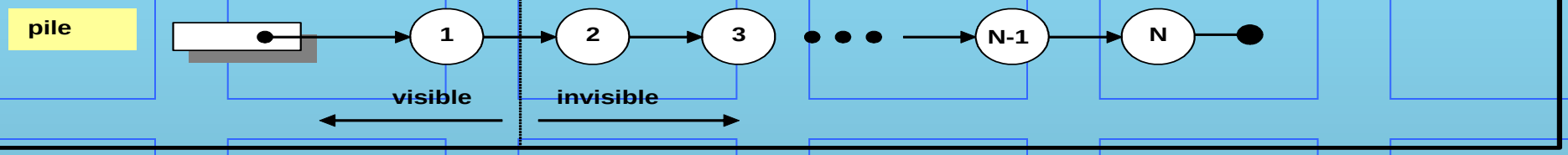
LISTES LINEAIRES

Aucune règle d'accès



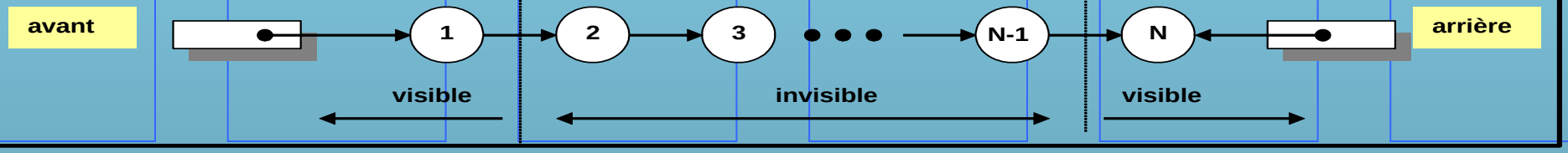
PILES

Seule la 1ère cellule est accessible



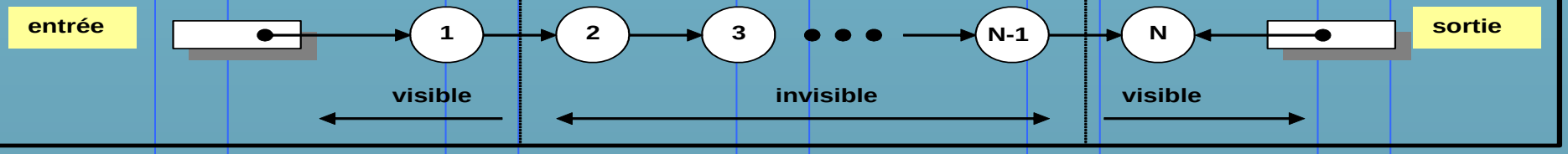
DEQUES

Seules la 1ère et la dernière cellule sont accessibles

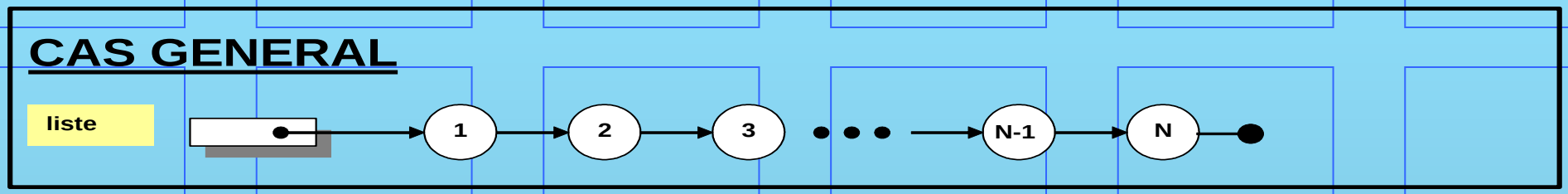


QUEUES

On ajoute les cellules à l'entrée et on ôte les cellules à la sortie

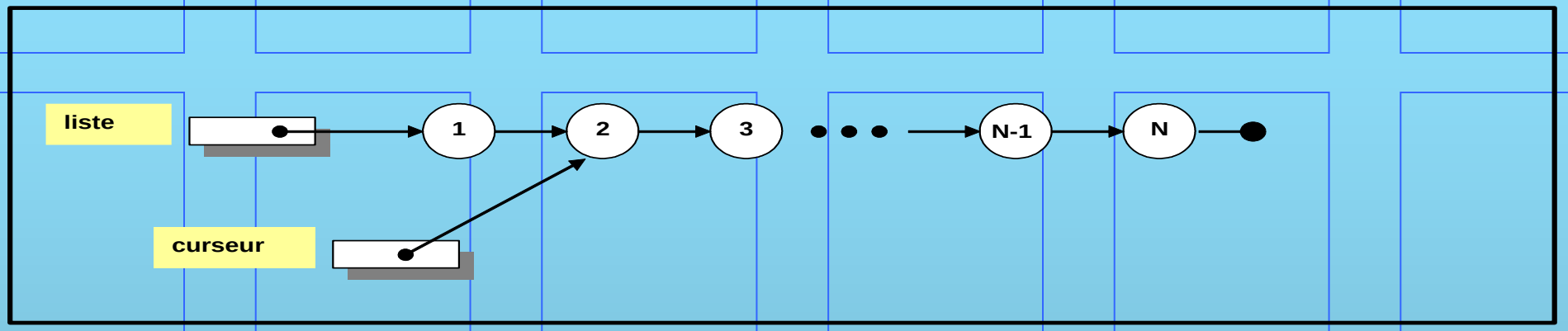


Définition de la liste linéaire



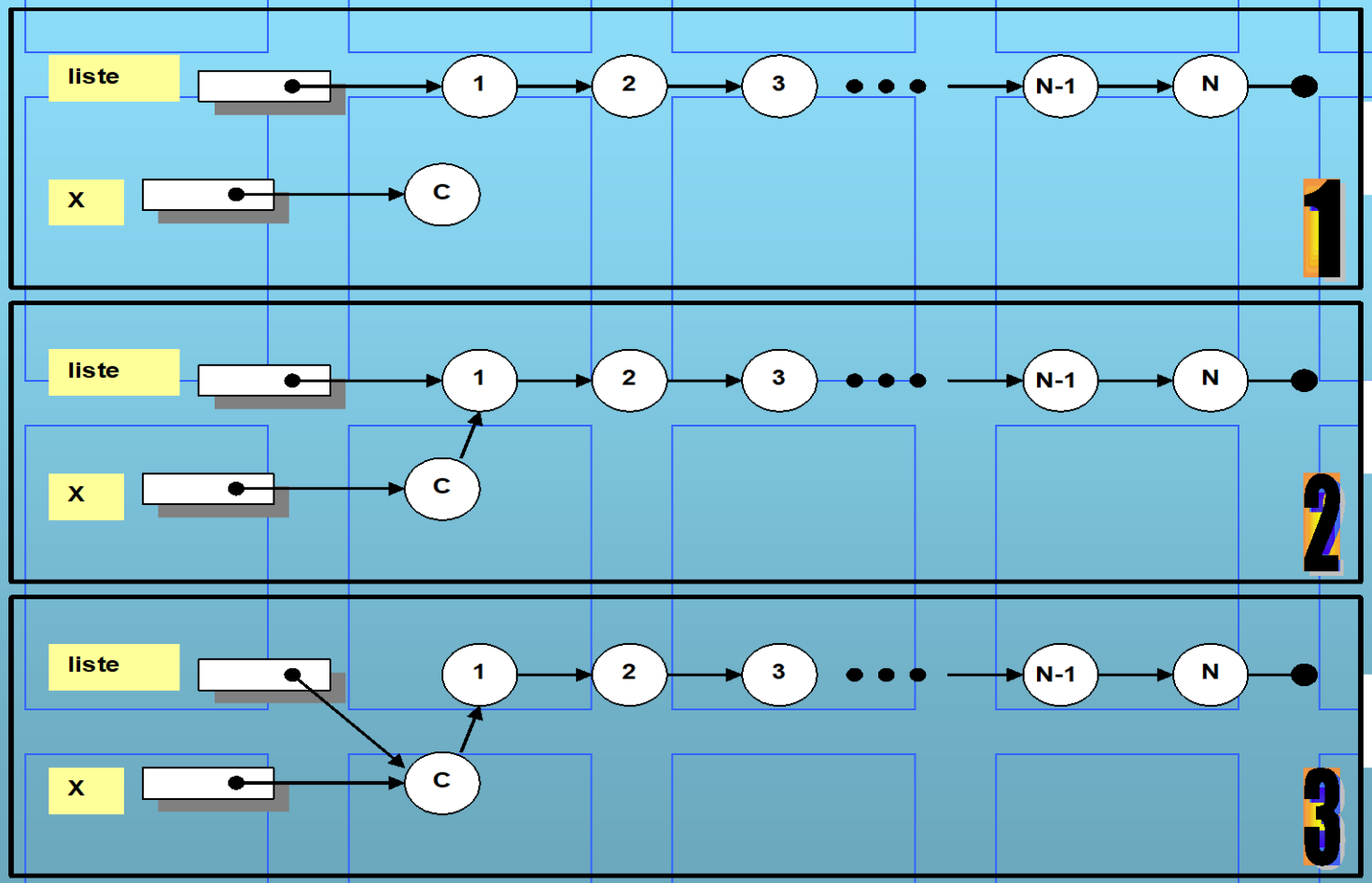

```
type pTCellule = ^Tcellule;

     Tcellule = record
                    clé: {données quelconques};
                    suivant: pTCellule;
                end;
```

Insertion de cellules



- Plusieurs cas peuvent se présenter
  - Insertion en tête de la liste
  - Insertion en fin de la liste
  - Insertion après le curseur
  - Insertion sous le curseur

Insertion en tête de liste

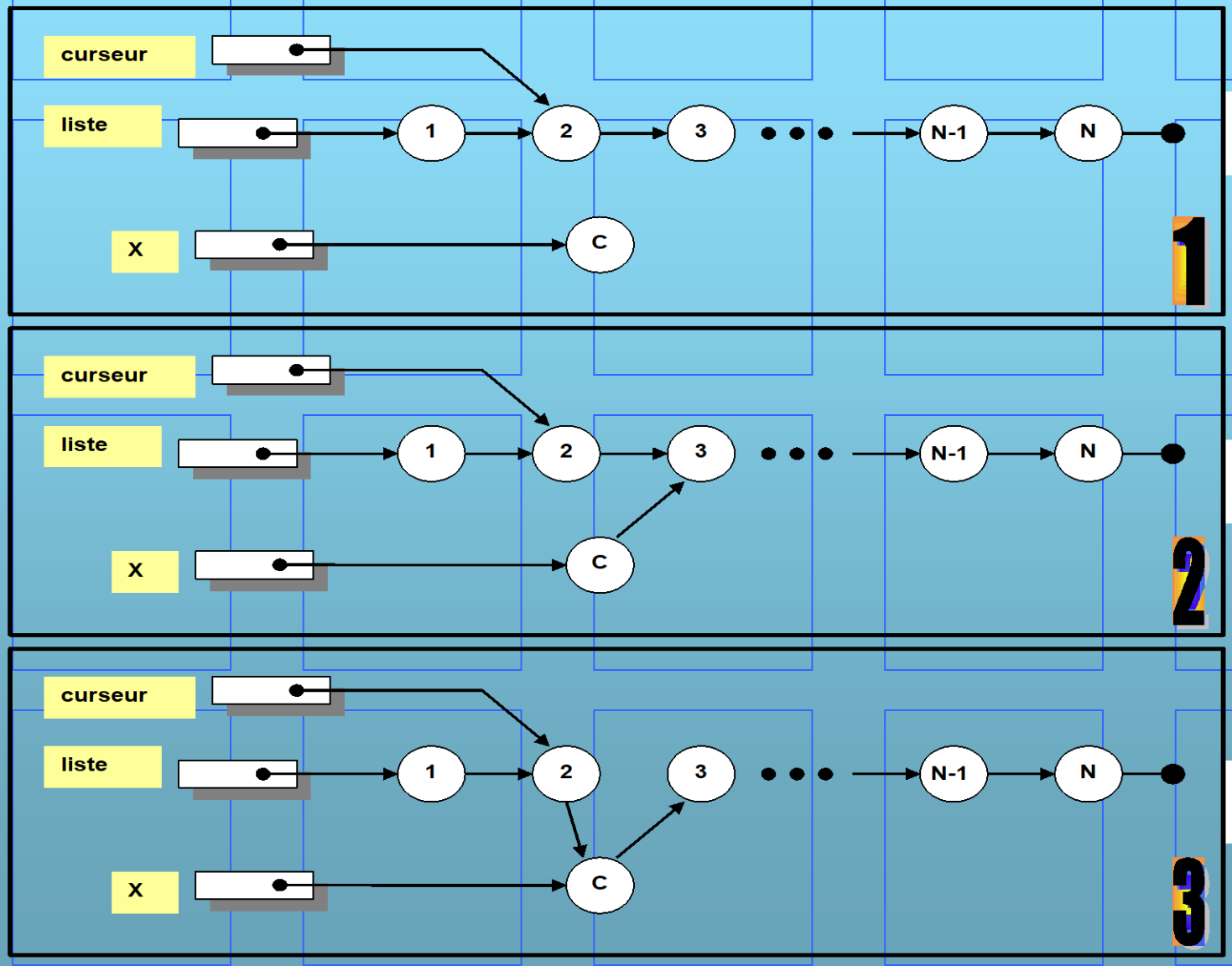


New (X)
X^.CLE := ... {les éléments à insérer}

X^.suivant := liste;

Liste := X;

Insertion après le curseur

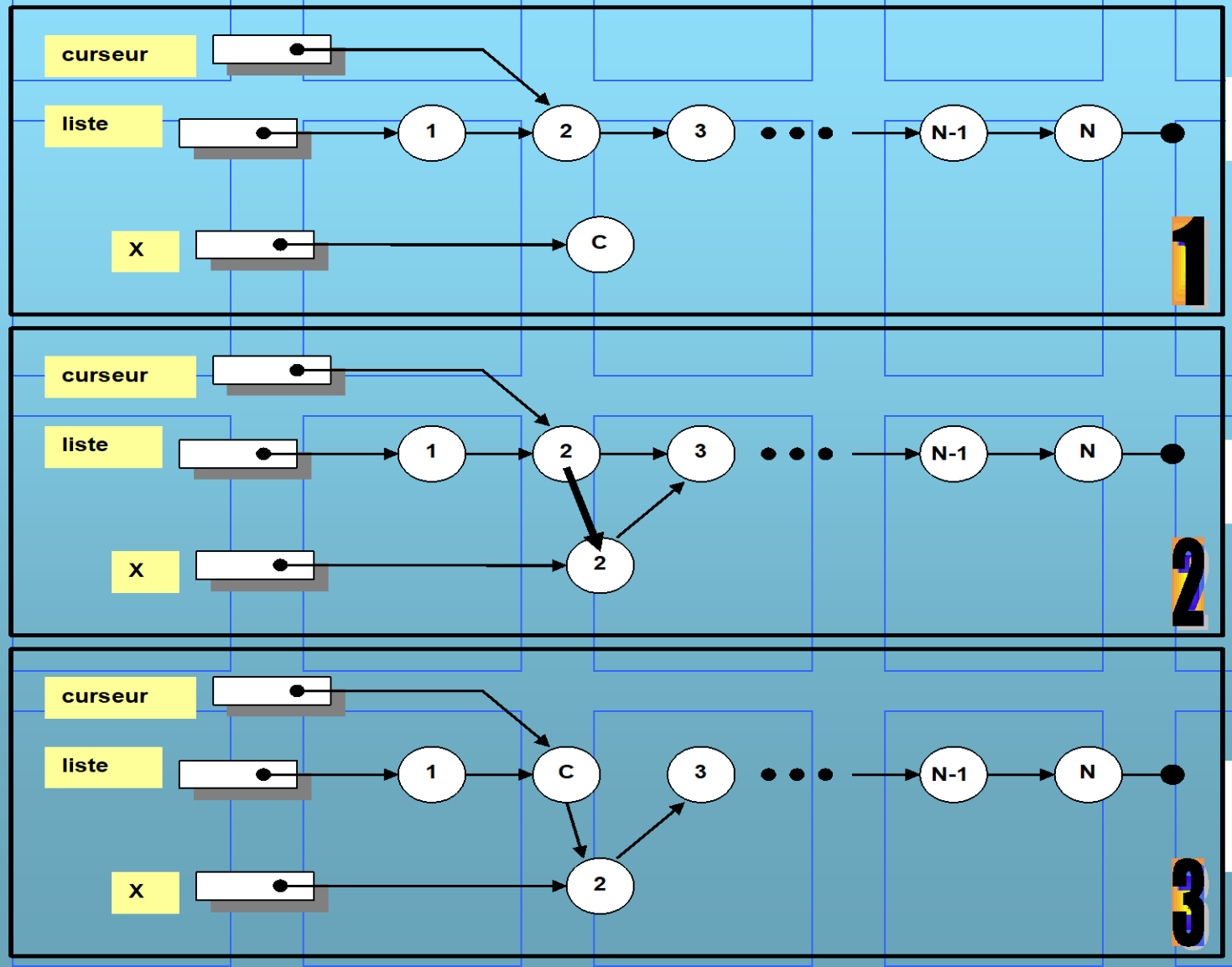


New (X)
X^.CLE := ... {les éléments à insérer}

X^.suivant := curseur^.suivant;

Curseur^.suivant := X;

Insertion sous le curseur

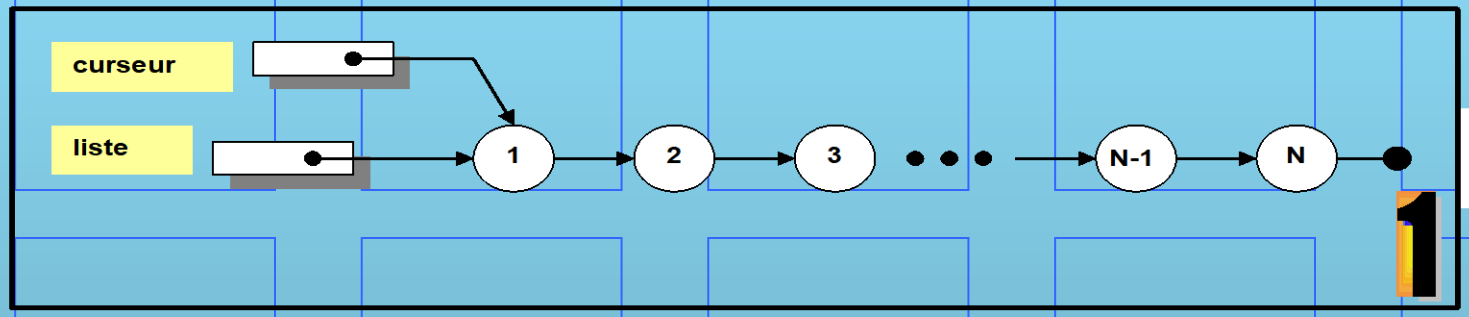


New (X)
X^.CLE := ... {les éléments à insérer}

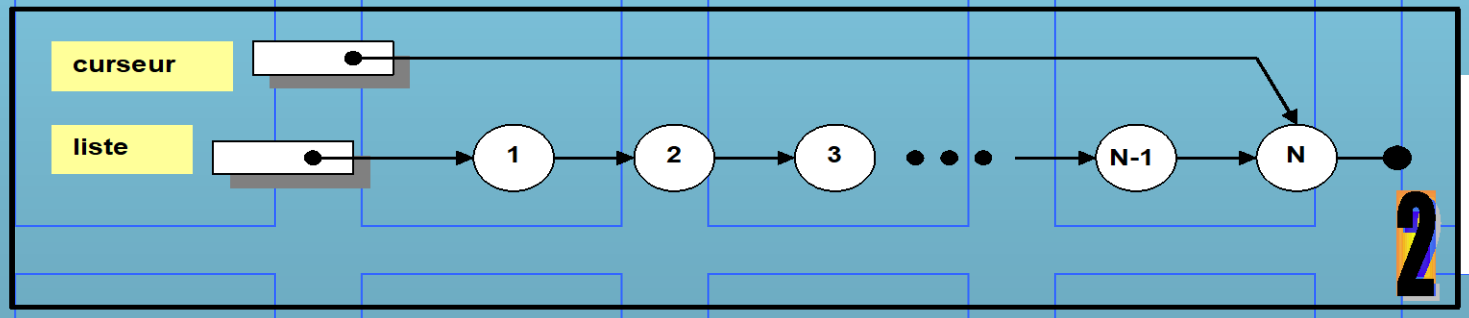
X^ := curseur ^

Curseur^.cle := ...{ éléments à insérer}
curseur^.suivant := X;

Insertion en fin de liste




```
Curseur := liste;
While curseur^.suivant<>nil then
  Curseur := curseur^.suivant;
```

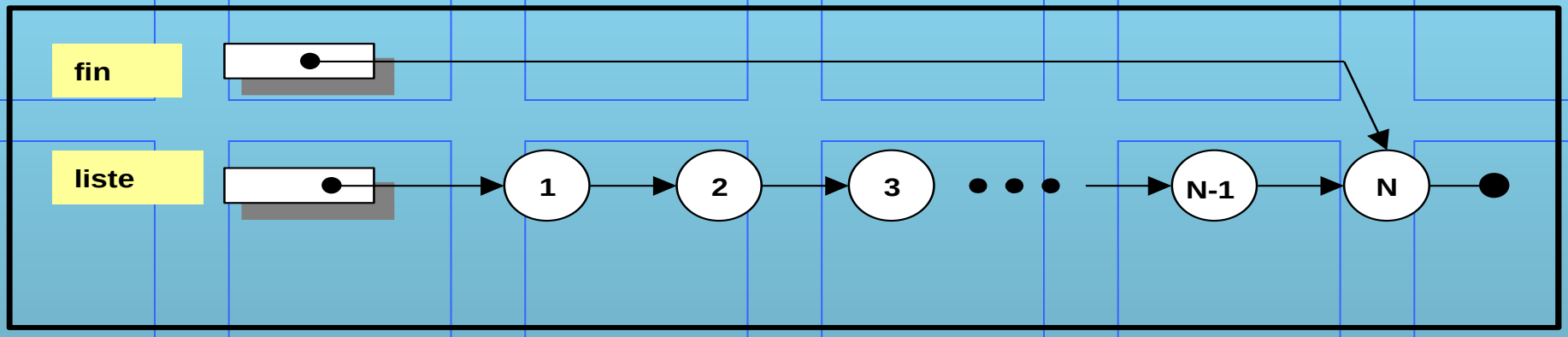


Quand on ressort de la boucle
le curseur pointe sur N

On se retrouve dans le cas où on
cherche à insérer après le curseur

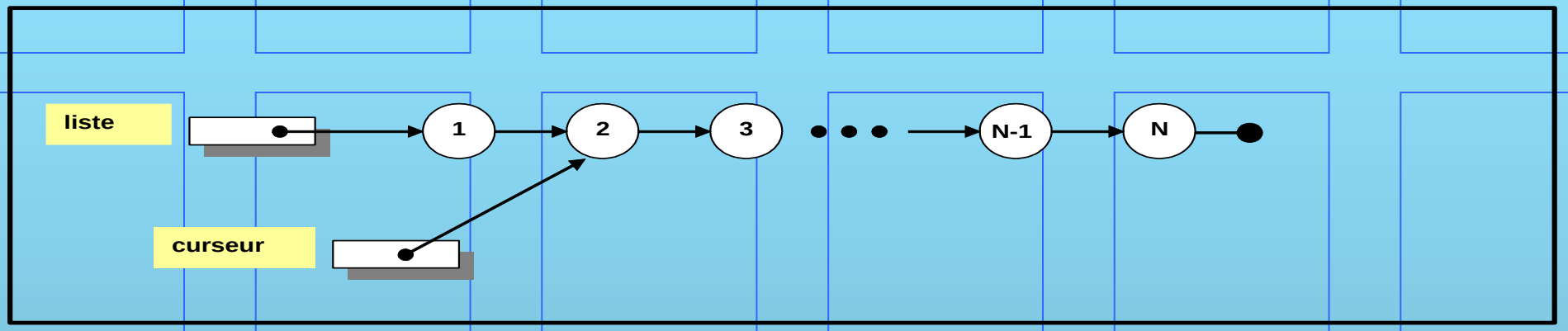
Insertion en fin de liste

Pour chaque insertion de cellule, il faut parcourir toute la liste. Cette méthode est très gourmande en temps.
Dans ce cas, il est intéressant de modifier un peu la structure de donnée en rajoutant un pointeur sur la dernière cellule



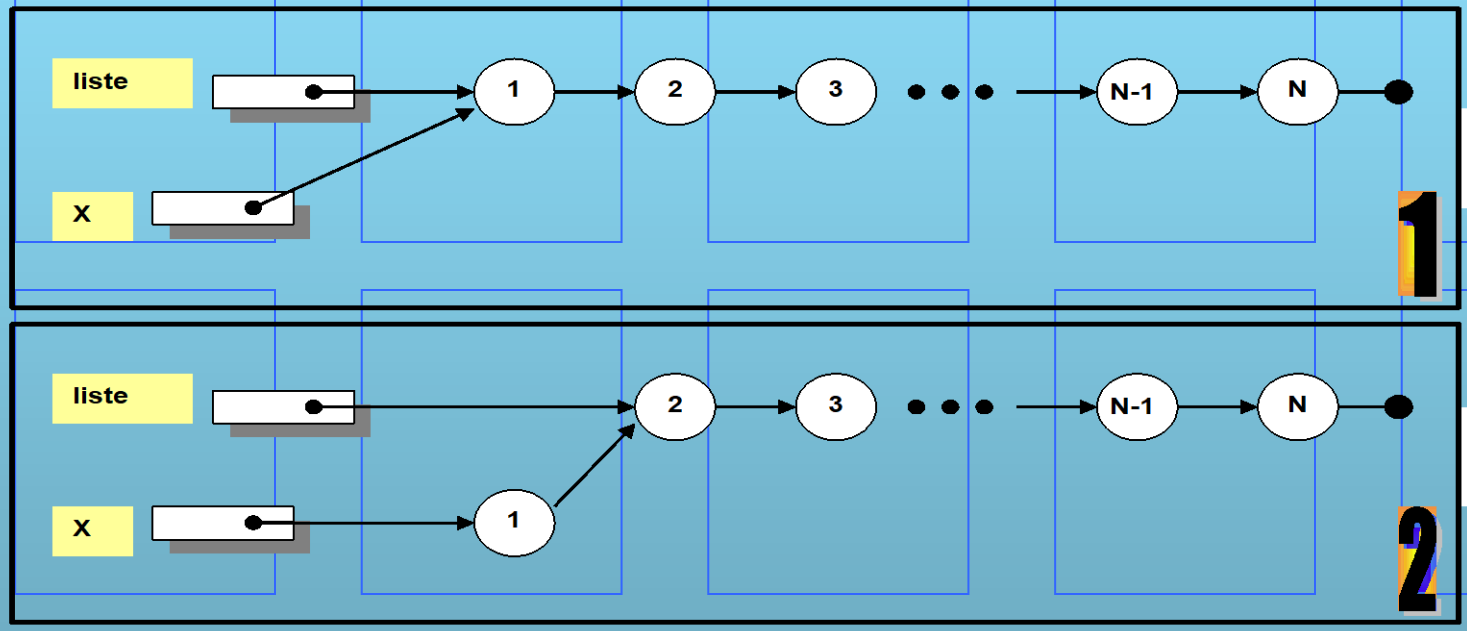
Avec cette astuce, et une perte mémoire minime, on se retrouve dans le cas de l'insertion d'une cellule après un curseur.

Destruction de cellules



- Plusieurs cas peuvent se présenter
  - Destruction en tête de la liste
  - Destruction après le curseur
  - Destruction sous le curseur
  - Destruction en fin de la liste

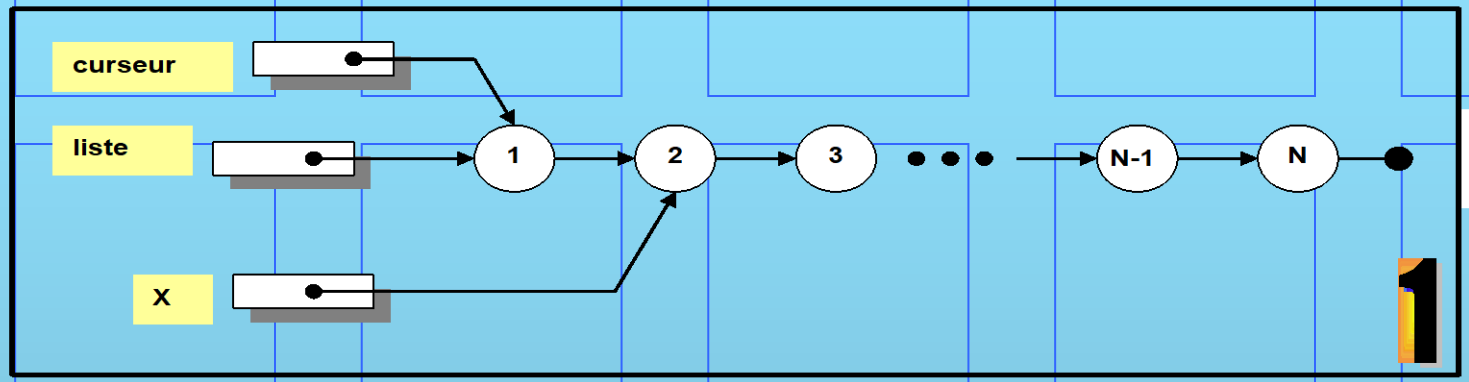
Destruction en tête de liste



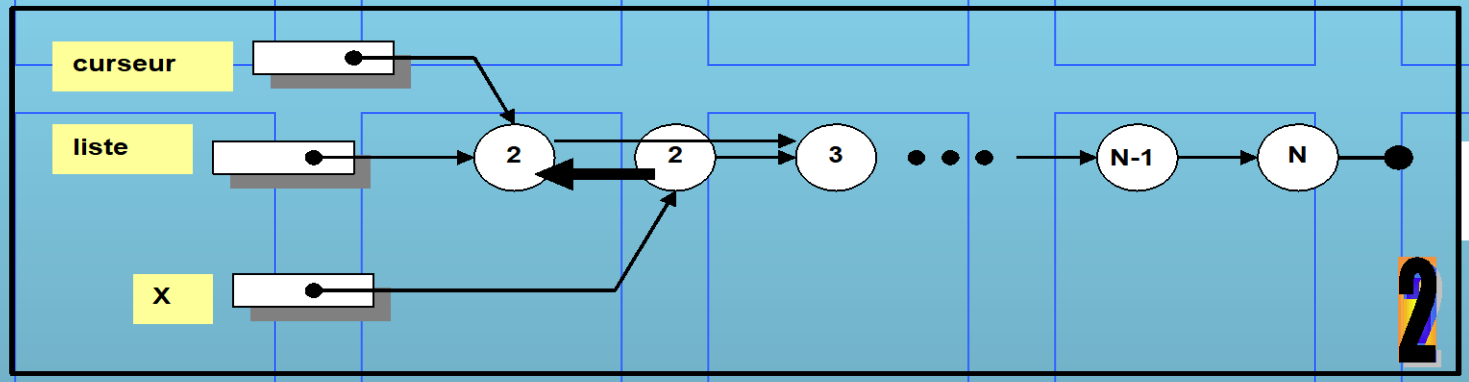

```
X := liste;
```

```
Liste := X^.suivant;
Dispose (X);
```

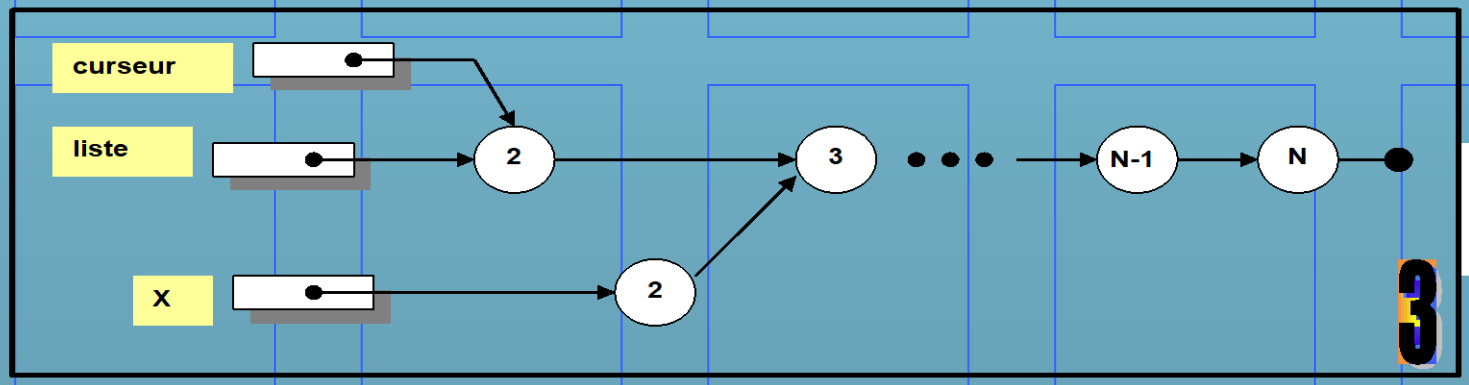
Destruction sous le curseur



X := curseur^.suivant;

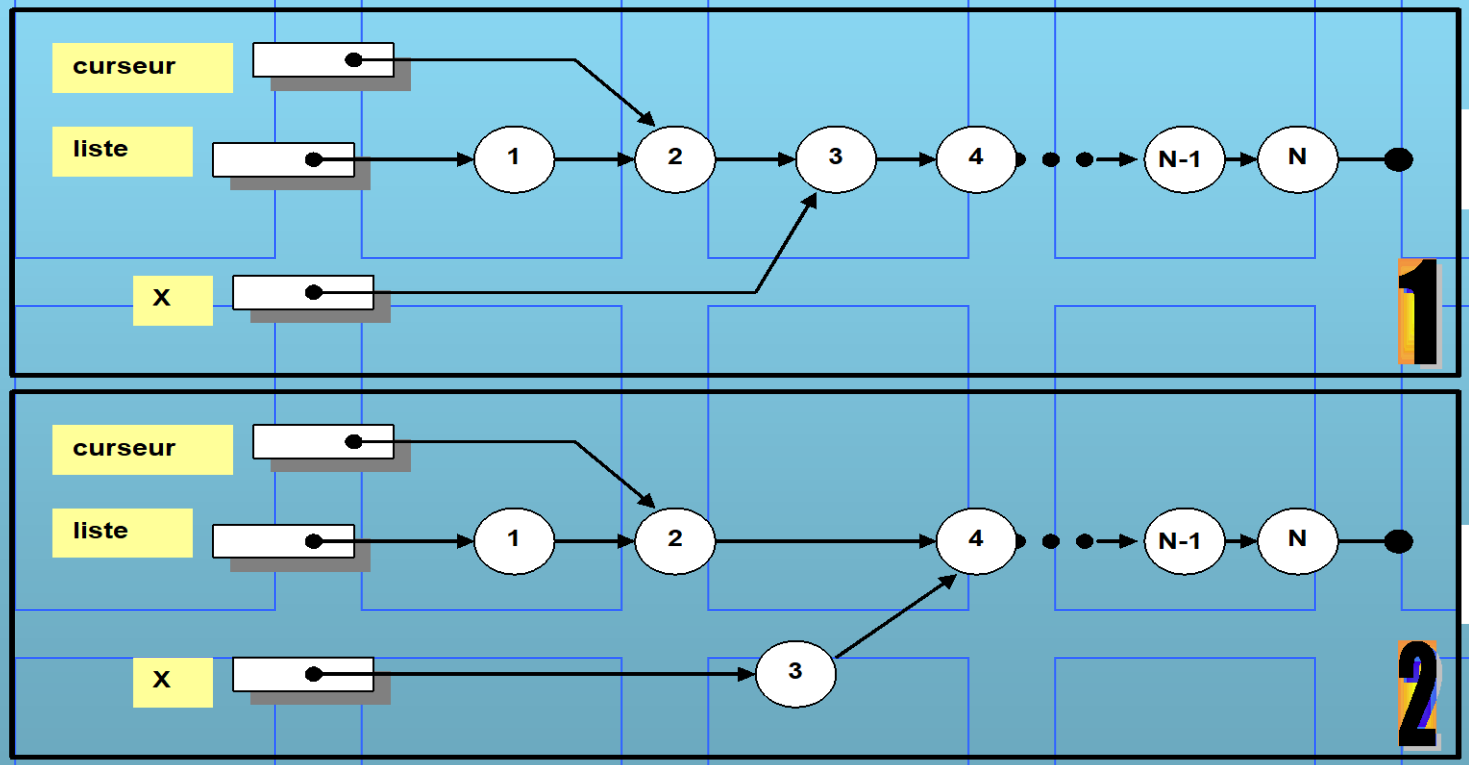


{Copie de la cellule}
curseur^ := X^



{Récupère la mémoire}
Dispose (X);

Destruction après le curseur

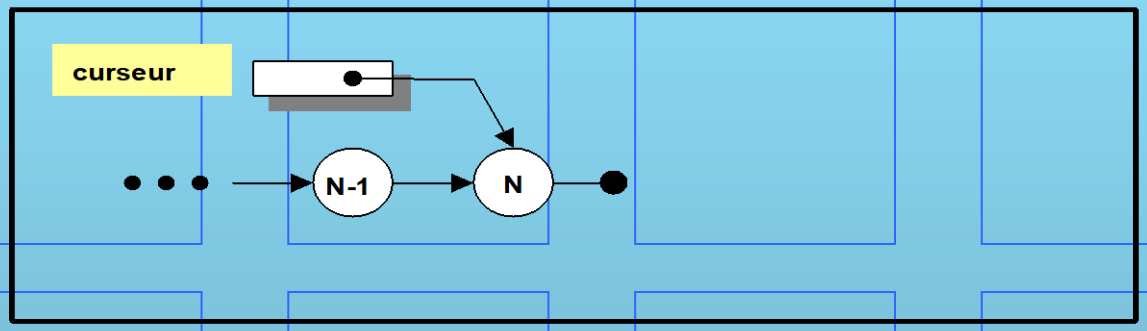



```
X:= curseur^.suivant
```

```
curseur^.suivant := X^.suivant;
dispose (X);
```

Destruction en fin de liste

Les deux derniers algorithmes se plantent lamentablement dans le cas suivant:

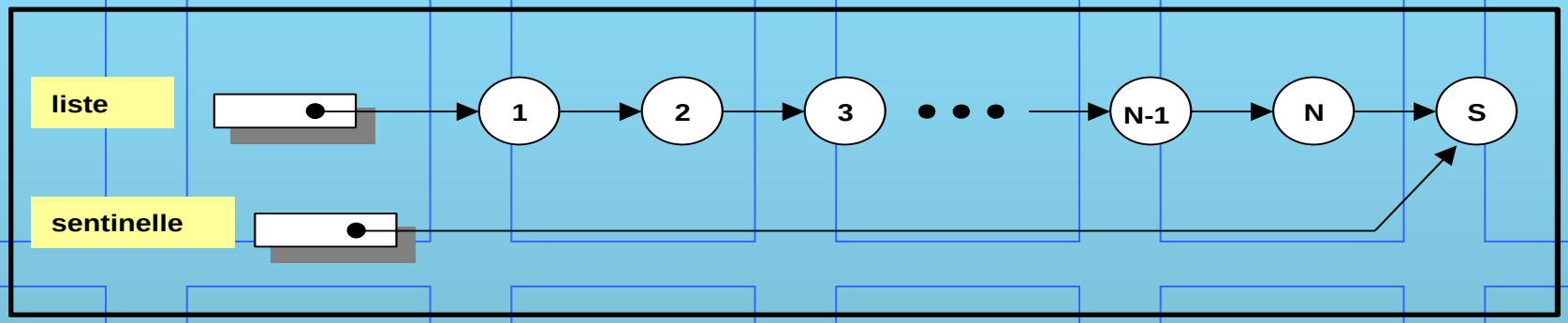


Il n'y a pas de manière simple de détruire la cellule (N) sans devoir parcourir toute la liste et rechercher la cellule (N-1).

VRAIMENT ?

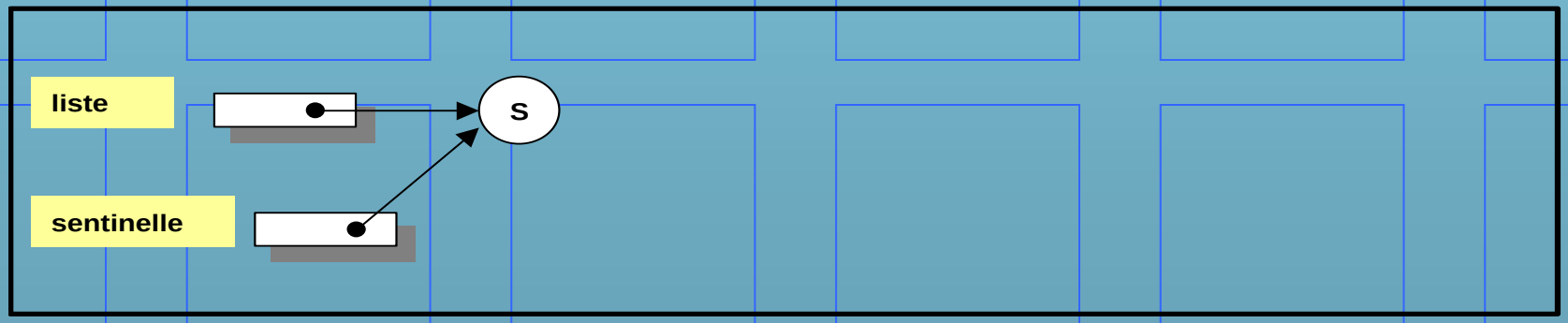
Technique de la sentinelle

Redéfinissons la liste ainsi:



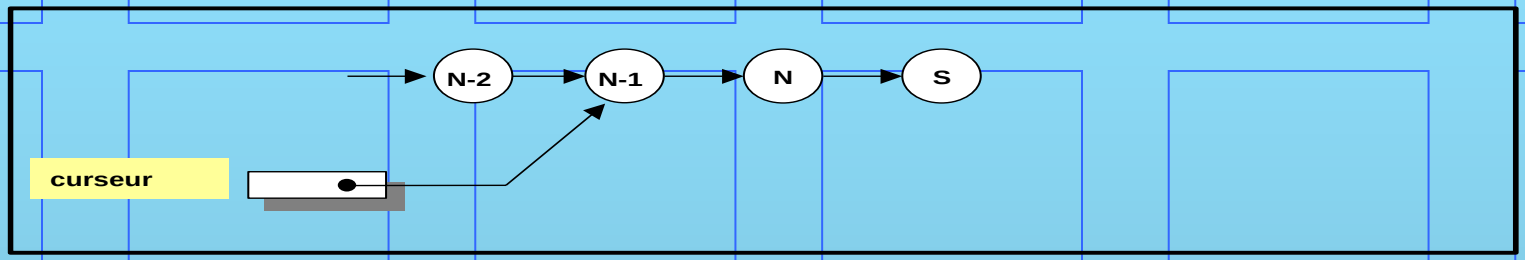
Le dernier élément de la liste est une cellule bidon appelée cellule sentinelle (S), qui sert uniquement comme terminateur de liste.

La liste vide devient:



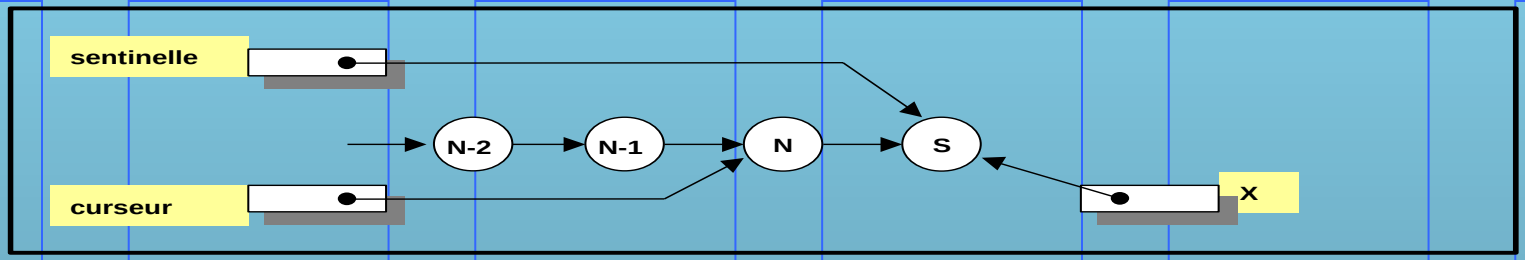
Destruction d'une cellule

Détruire après le curseur



Si (N) est la sentinelle, on cherche à détruire une cellule qui n'existe pas. On a probablement une erreur.

Détruire sous le curseur



On peut utiliser ici sans problème l'algorithme vu précédemment.

Attention: dans notre exemple, ne pas oublier de corriger la sentinelle si la cellule à détruire est la dernière:

```
  If curseur=sentinelle then begin
    {probablement une erreur}
  end {if};
{recopier la cellule}
  X := curseur^.suivant;
  curseur^ := X^;
{corriger la sentinelle}
  if X=sentinelle then begin
    sentinelle := curseur;
  end {if};
{récupérer la mémoire}
  Dispose (X);
```

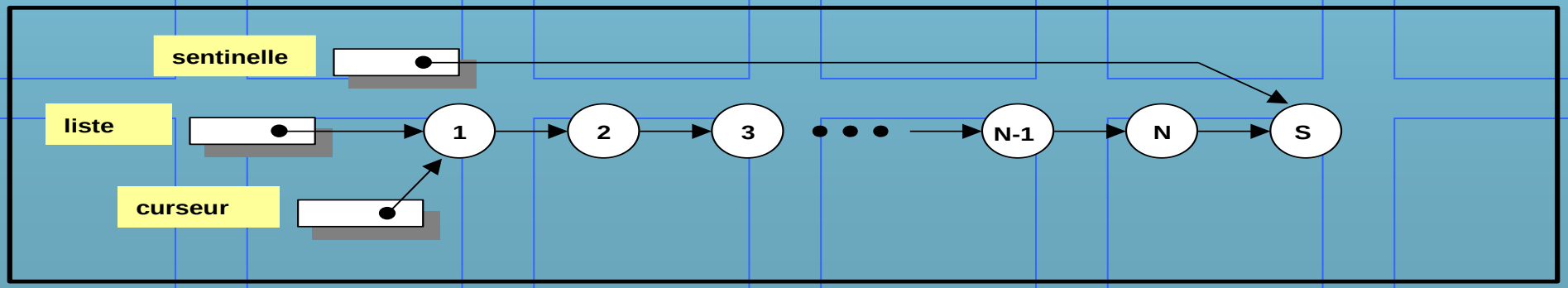
Parcours d'une liste

sans sentinelle

```
Trouve := false;
Curseur := liste;

While curseur^.suivant<>nil then begin
  If curseur^.cle=recherche then begin
    Trouve := true;
    Break;
  end {if};
  curseur := curseur^.suivant;
end {while};

if trouve then begin
  {effectuer les opérations voulue}
end {if};
```

avec sentinelle

```
Curseur := liste;

While curseur <>sentinelle then begin
  If curseur^.cle=recherche then begin

    Break;
  end {if};
  curseur := curseur^.suivant;
end {while};

if curseur<>sentinelle then begin
  {effectuer les opérations voulue}
end {if};
```



Warning

Vous pensez que ces méthodes fonctionnent en tout temps ?

Vous pensez que ces algorithmes sont corrects ?

Warning

Vous pensez que ces méthodes fonctionnent en tout temps ?

Vous pensez que ces algorithmes sont corrects ?

Si vous répondez oui, vous risquez des surprises...

Hypothèses

Les méthodes décrites ci-dessus sont génériques et sont correctes.

Maintenent, explicitons certaines hypothèses «cachées»

1 Taille d'une liste : 10 millards d'élèments

2 Clé : taille variable («variant»)

3 Nombre d'insertion/destruction : 1 millard

Ce ne sont pas des hypothèses farfelues... Dans le pire des cas, on peut avoir 1, 2 et 3 simultanément.

Hypothèses

On voit qu'il faut impérativement tenir compte d'autres paramètres !

Avec «l'énorme» taille de la liste, on se rend compte qu'elle ne tiendra pas dans la mémoire vive de notre ordinateur. On suppose que l'OS sur lequel notre programme fonctionnera gère correctement le «swapping» et qu'il gére aussi un adressage >32 bits. Si ce n'est pas le cas, c'est à nous de le faire, ce qui compliquera la programmation.

Hypothèses

Avec une taille variable des cellules, l'OS et surtout le gestionnaire de mémoire, doit être suffisemment «intelligent» pour éviter la fragmentation de la mémoire.

Attention : les procédure NEW et DISPOSE sont liés à l'OS et peuvent cacher de grosses surprises, comme la fragmentation de la mémoire, des «fuites» (leak) de mémoire.

Le plus simple est de gérer nous même la mémoire, avec une liste de cellules libres...

sans oublier le problème de la taille de notre liste !

Hypothèses

Si le nombre d'opérations à faire (insertion/destruction/lecture) est très grand, le temps utilisé peut vite exploser. Si la liste est grande et qu'elle ne tient plus en mémoire vive, cela peut très vite devenir prohibitif. Le temps passé proviendra de la qualité de la gestion mémoire de l'OS, des librairies new/dispose et de la procédure de déplacement du curseur.

Hypothèses

Heureusement, dans la plupart des cas, les méthodes décrites ci-dessus suffisent.

Il faut toujours se poser la question s'il n'est pas plus judicieux de gérer nous même les procédures NEW/DISPOSE avec la technique des listes libres.

Pour traiter les cas «extrêmes», il faudrait un autre exposé.