Structures de Données

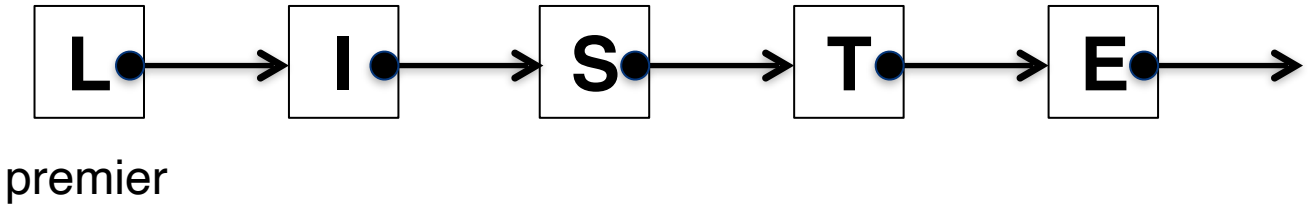
Séance 2 – Liste chaînée

INFRES 12 – janvier 2020

Liste chaînée

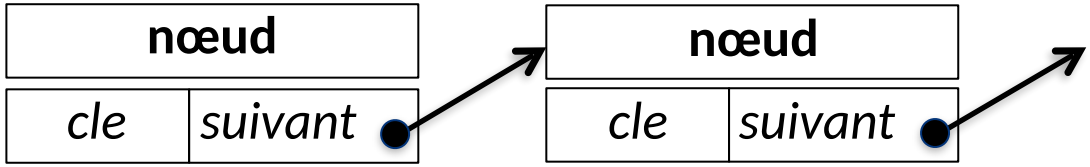
Liste chaînée

- Structure de données permettant de représenter un ensemble d'éléments (appelés **nœuds**) possédant chacun une valeur (appelée **clé**) et un **lien** vers le nœud suivant.

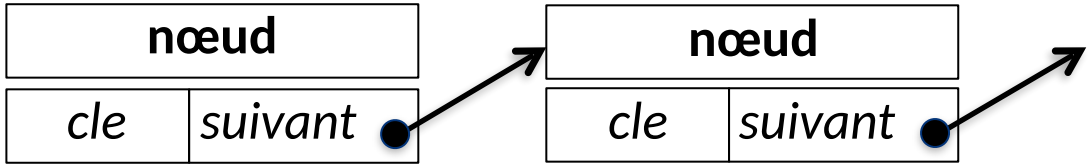


- Accès à la liste par son premier élément

Liste chaînée
Première implantation



Liste chaînée
Première implantation



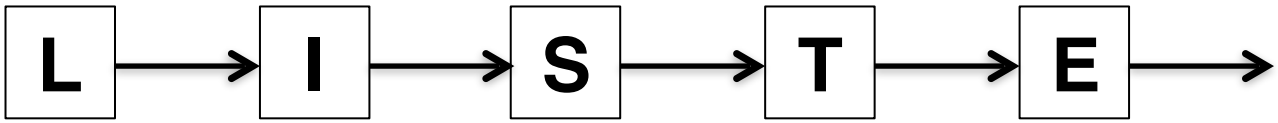

```
struct noeud {
      char cle;    // char ou tout autre type
      struct noeud *suivant; // pointeur
};
```

Liste chaînée vs Tableau

- Avantages :

- Inconvénients :

Indice	Valeur
0	**T**
1	**A**
2	**B**
3	**L**
4	**E**
5	**A**
6	**U**

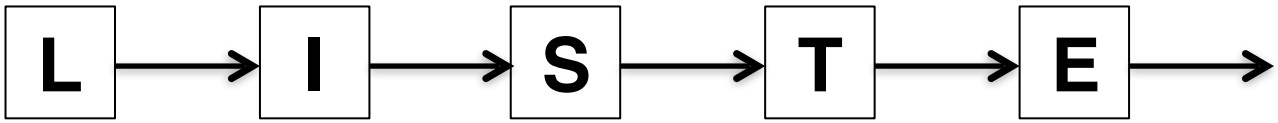


Liste chaînée vs Tableau

- Avantages :
  - Pas de taille fixée à l'avance
  - Facile à réordonner

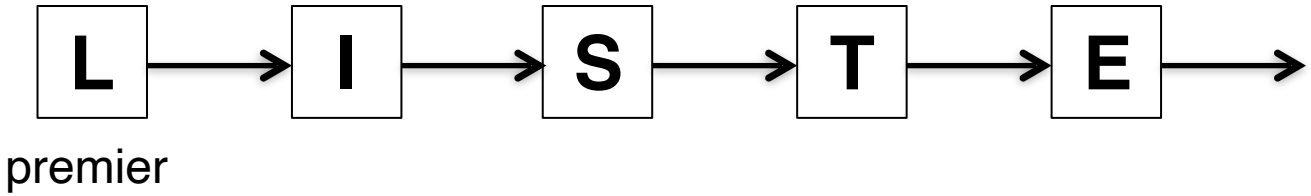
Indice	Valeur
0	**T**
1	**A**
2	**B**
3	**L**
4	**E**
5	**A**
6	**U**

- Inconvénients :
  - pas d'accès direct à un élément
  - plus de mémoire utilisée pour une même taille



Liste chaînée – Insertion

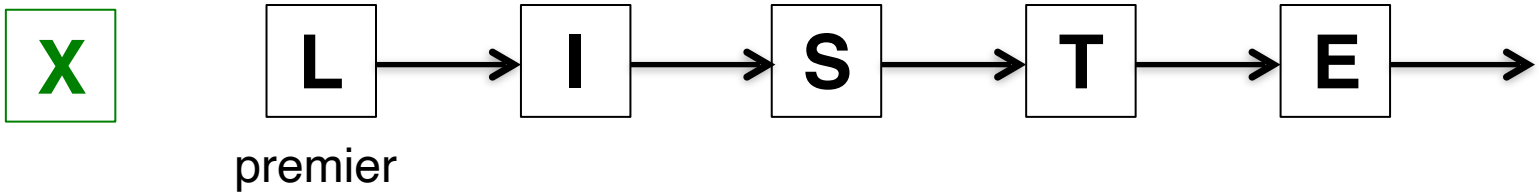
Différents cas à considérer



x ?

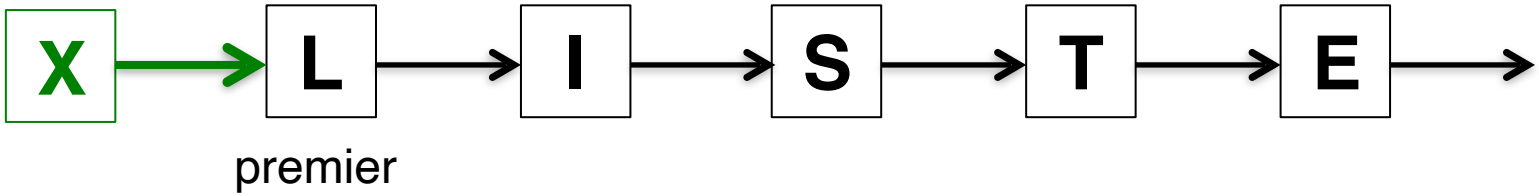
Liste chaînée – Insertion (1)

1. Insertion en tête : création du nœud



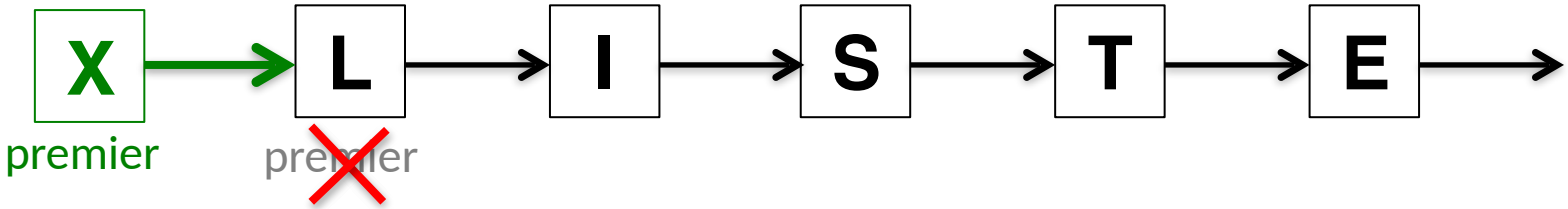
Liste chaînée – Insertion (1)

1. Insertion en tête : lien avec le premier



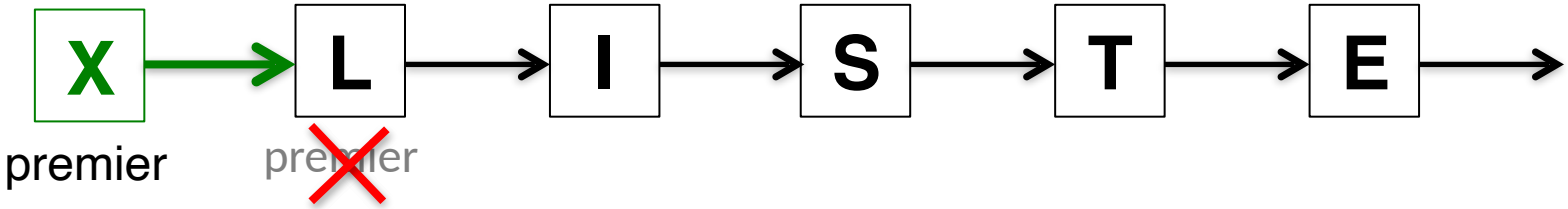
Liste chaînée – Insertion (1)

1. Insertion en tête : nouveau premier



Liste chaînée – Insertion (1)

1. Insertion en tête : nouveau premier



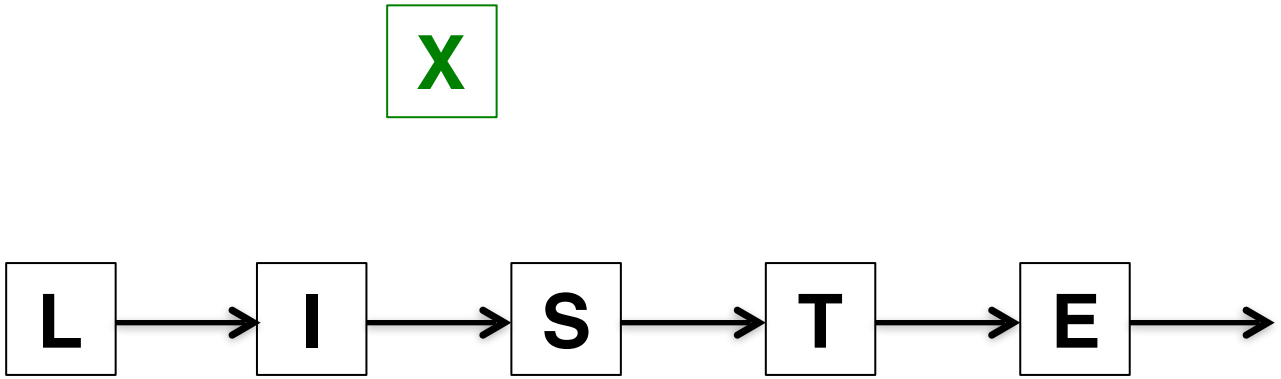
création du nœud x

x->suivant = premier

premier = x

Liste chaînée – Insertion (2)

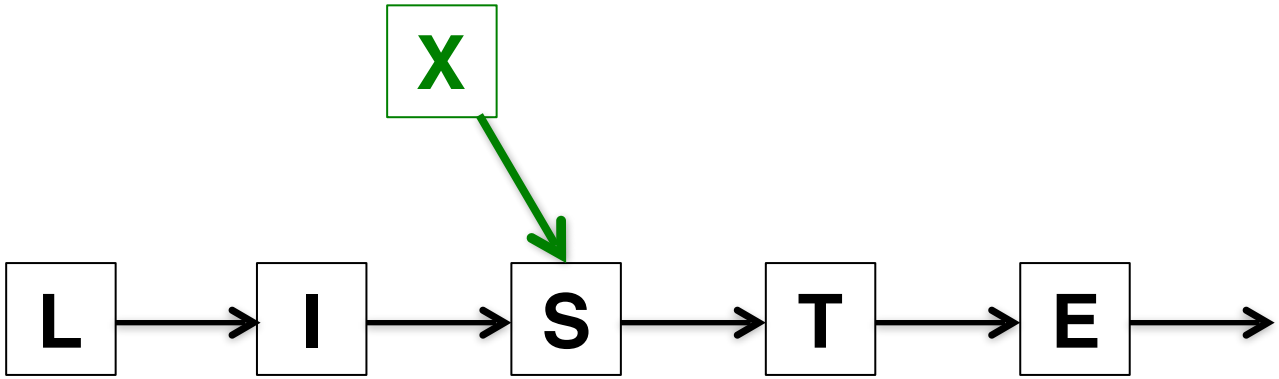
2. Cas général : création du nœud



création du nœud

Liste chaînée – Insertion (2)

2. Cas général : suivant de x

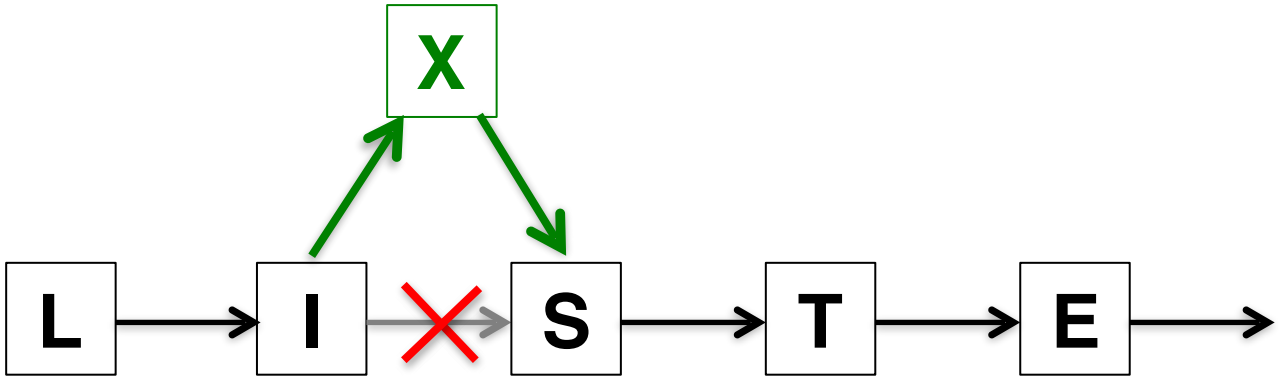


création du nœud

x->suivant = i->suivant

Liste chaînée – Insertion (2)

2. Cas général : précédent de x



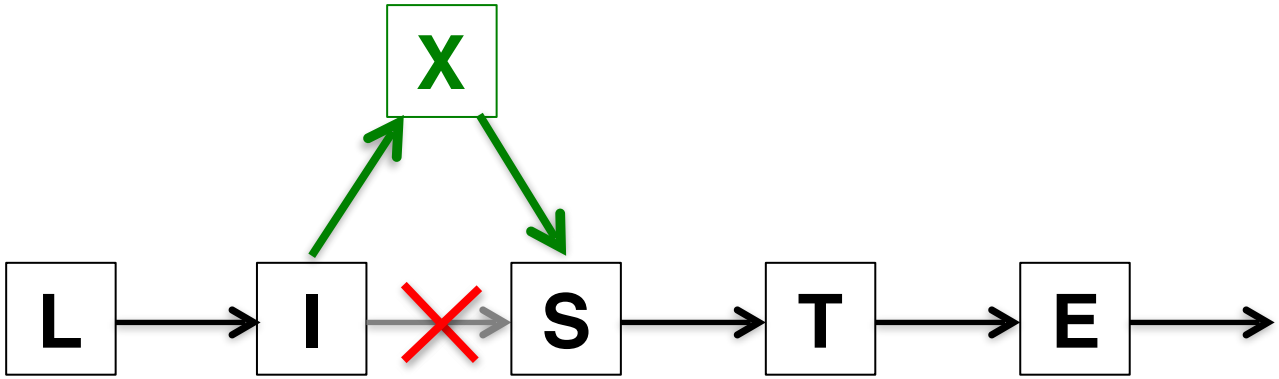
création du nœud

x->suivant = i->suivant

i->suivant = x

Liste chaînée – Insertion (2)

2. Cas général : récapitulatif



création du nœud
x->suivant = i->suivant
i->suivant = x
// Seul le nœud précédent (ici i) doit être connu.

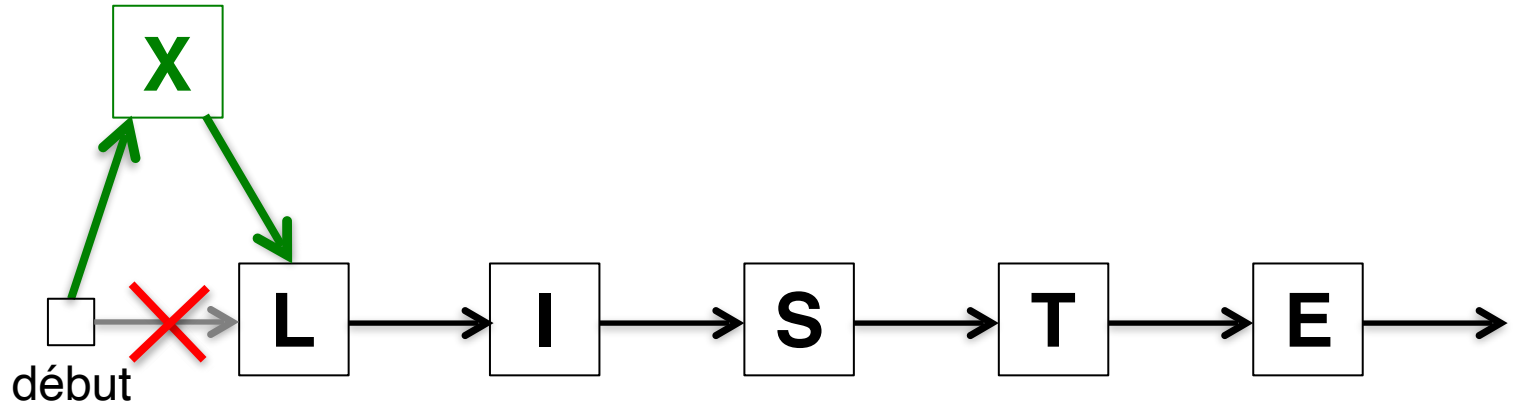
Liste chaînée – Insertion (3)

3. Comment se ramener à un seul cas d'insertion ?

Liste chaînée – Insertion (3)

3. Comment se ramener à un seul cas d'insertion ?

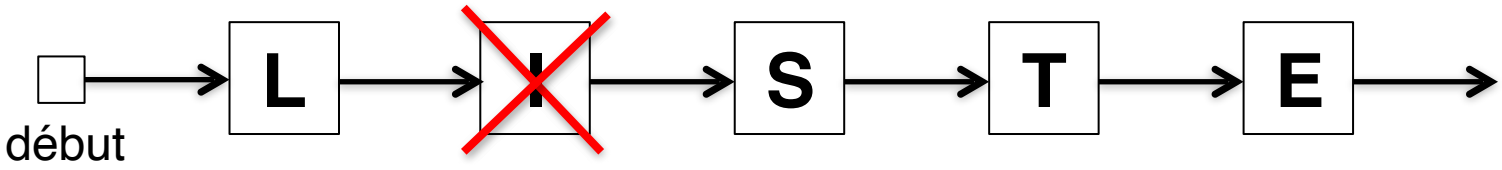
=> Nœud sentinelle de début



- Généralisation : **insererApres**(cle, nœud)

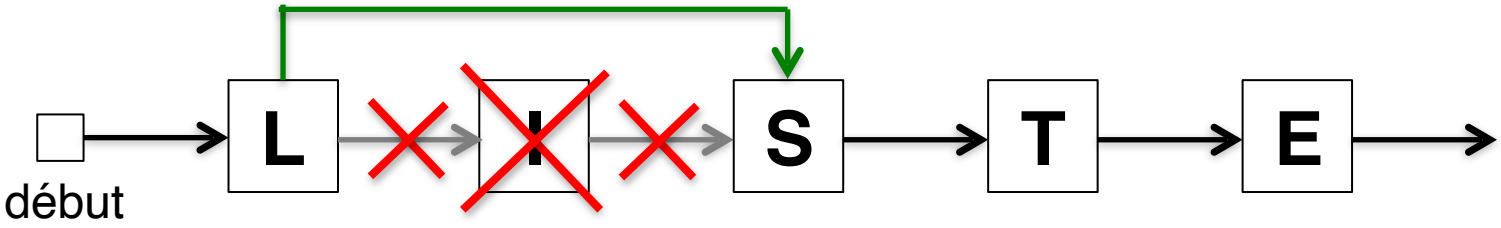
Liste chaînée - Suppression

- Avec nœud sentinelle de début



Liste chaînée - Suppression

- Avec nœud sentinelle de début : un seul cas

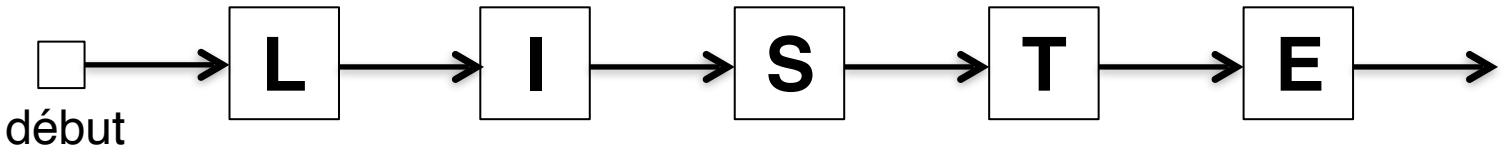


L->suivant = L->suivant->suivant

// Seul le nœud précédent doit être connu (ici L)

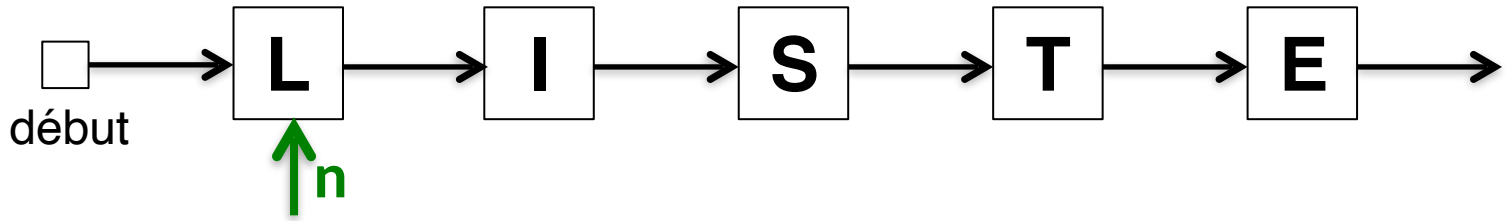
- Généralisation : **supprimerApres**(nœud)

Liste chaînée – Parcours



Boucle pour parcourir tous les nœuds de la liste ?

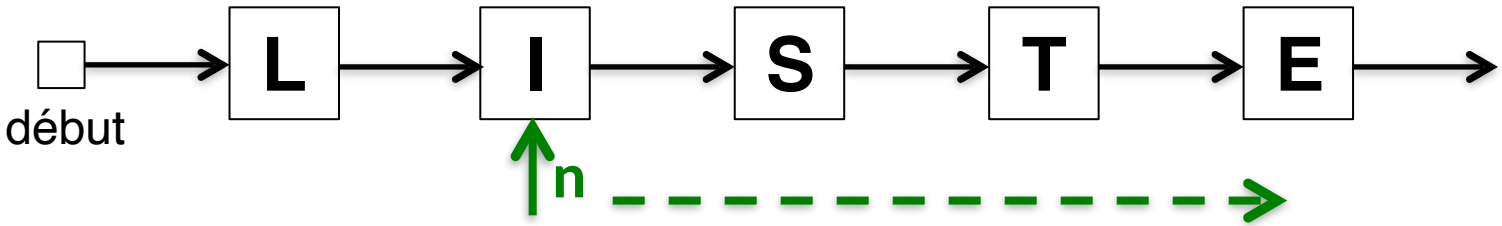
Liste chaînée – Parcours



Boucle pour parcourir tous les nœuds de la liste ?

```
// premier nœud
struct noeud *n = debut->suivant;
```

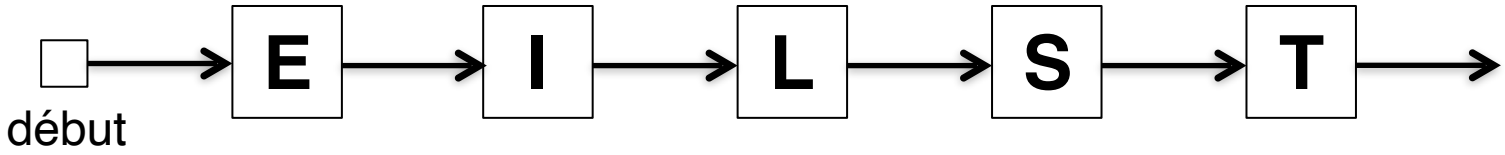
Liste chaînée – Parcours



Boucle pour parcourir tous les nœuds de la liste ?

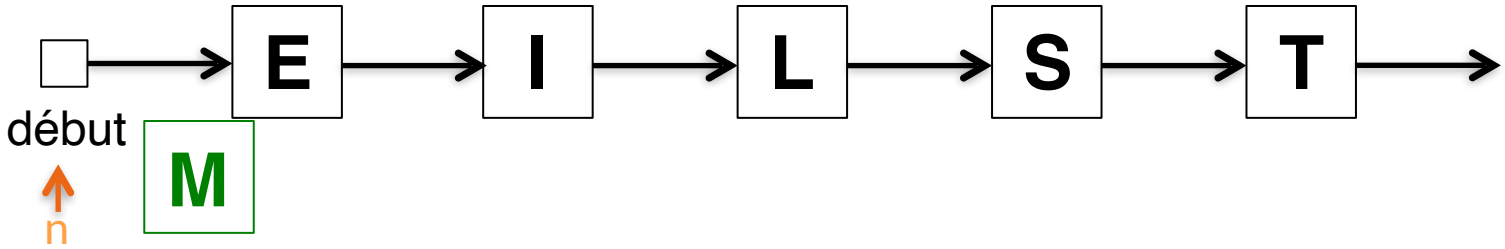
```
// premier nœud
struct noeud *n = debut->suivant;
// itération sur les suivants
while (n != NULL) {
      n = n->suivant;
}
```

Liste chaînée – Insertion ordonnée



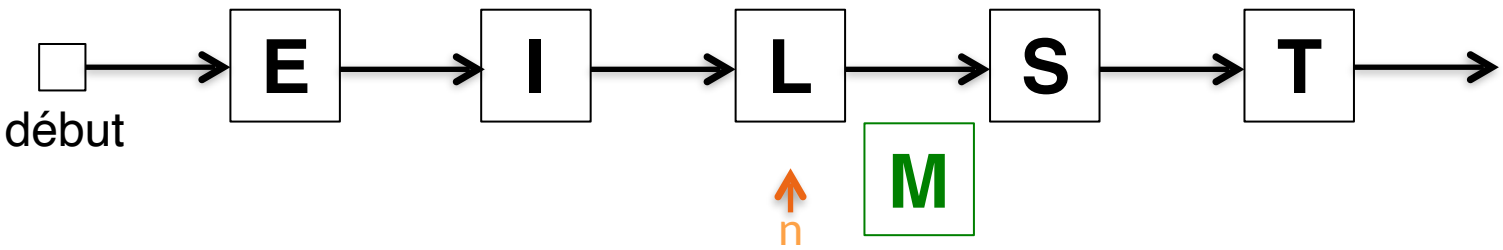
M

Liste chaînée – Insertion ordonnée




```
// Recherche du premier nœud plus grand que M
struct noeud *n = debut;
```

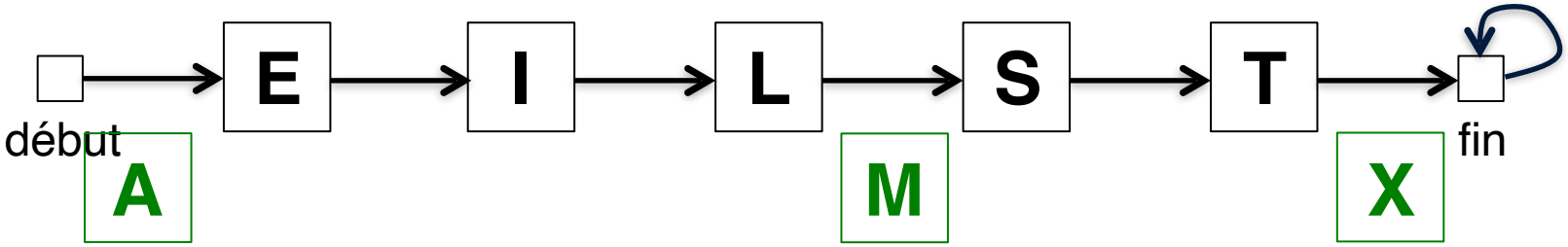
Liste chaînée – Insertion ordonnée




```
// Recherche du premier nœud plus grand que M
struct noeud *n = debut;
while (n->suivant != NULL
            && n->suivant->cle < cleAInserer) {
      n = n->suivant;
}
// insérer avant n->suivant donc après n
```

- Suppression du test `n->suivant != NULL` ?

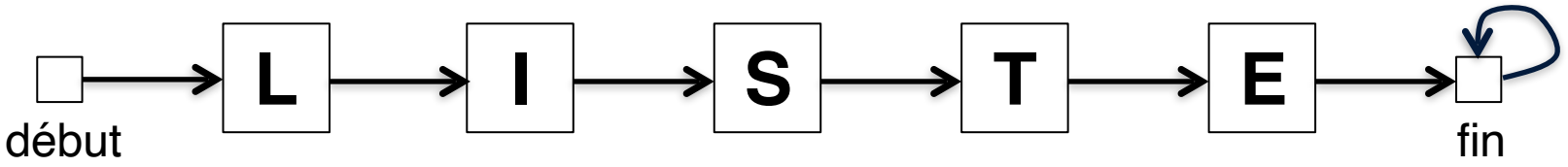
Liste chaînée – Insertion ordonnée



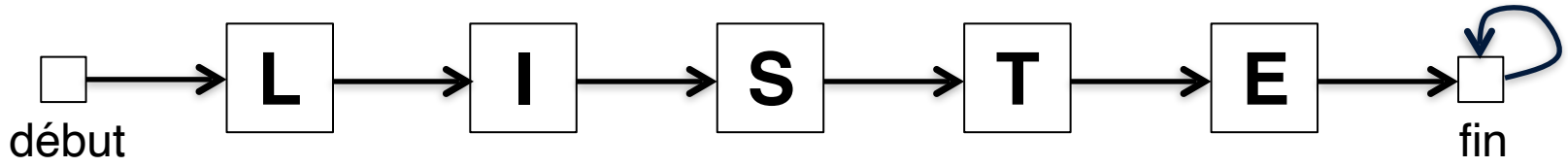
- Ajout d'un nœud sentinelle de fin

```
// Recherche de la position (ordre croissant)
struct noeud *n = debut;
fin->cle = cleAInserer;
while (n->suivant->cle < cleAInserer) {
      n = n->suivant;
}
// insérer avant n->suivant donc après n
```

Liste chaînée – Recherche séquentielle

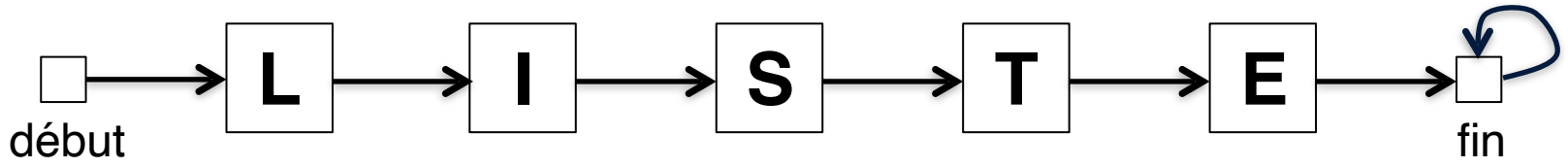


Liste chaînée – Recherche séquentielle



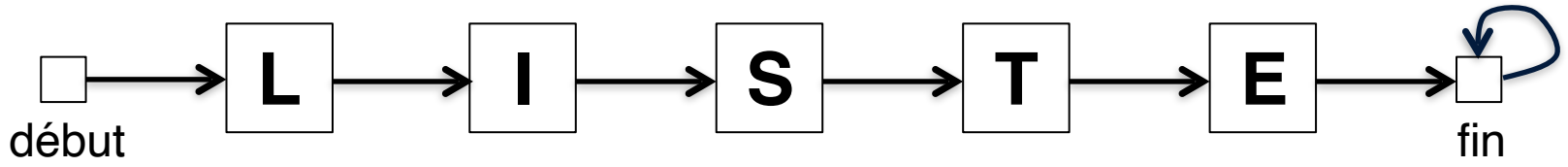

```
// Ordre croissant : recherche de la position
struct noeud *n = debut->suivant;
```

Liste chaînée – Recherche séquentielle



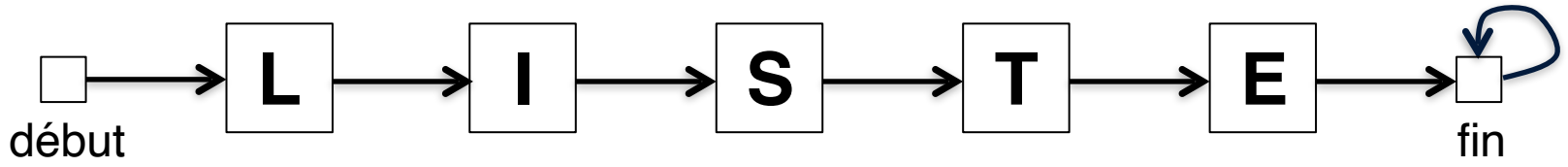

```
// Ordre croissant : recherche de la position
struct noeud *n = debut->suivant;
fin->cle = cleRecherchee;
```

Liste chaînée – Recherche séquentielle



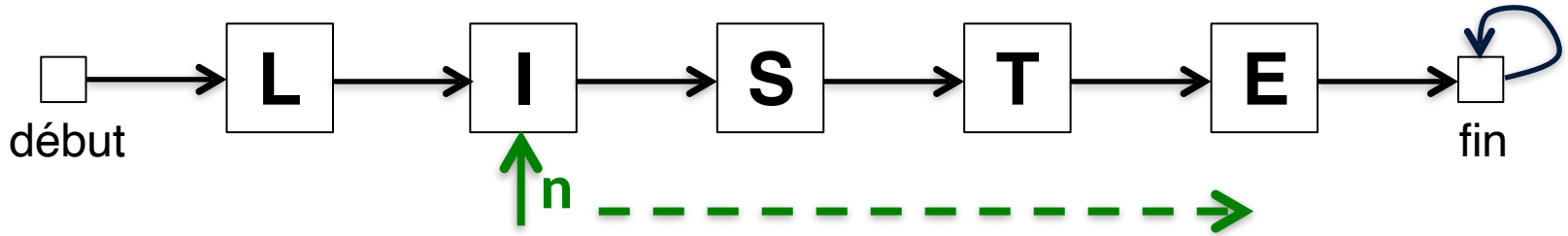

```
// Ordre croissant : recherche de la position
struct noeud *n = debut->suivant;
fin->cle = cleRecherchee;
while (n -> cle != cleRecherchee) {
      n = n->suivant;
}
```

Liste chaînée – Recherche séquentielle




```
// Ordre croissant : recherche de la position
struct noeud *n = debut->suivant;
fin->cle = cleRecherchee;
while (n -> cle != cleRecherchee) {
      n = n->suivant;
}
if (n != fin /* ou n != n->suivant */) {
      // trouvé !
}
```

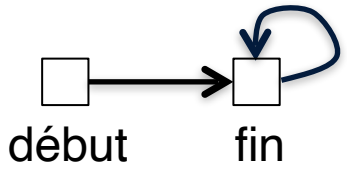
Liste chaînée – Parcours



Avec nœud sentinelle de fin :

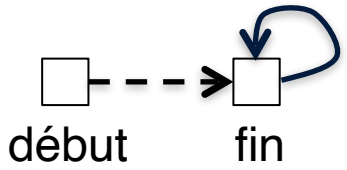
```
// premier nœud
struct noeud *n = debut->suivant;
// itération sur les suivants
while (n != n->suivant) {
      n = n->suivant;
}
```

Liste chaînée
Création avec nœuds sentinelles




```
struct noeud *debut;
struct noeud *fin;
debut = (struct noeud *)malloc(sizeof * debut);
fin = (struct noeud *)malloc(sizeof * fin);
debut->suivant = fin;
fin->suivant = fin;
// l'accès à la liste se fait par le nœud debut
// Le nœud de fin est le seul à être son propre suivant.
```

Liste chaînée
Création avec nœuds sentinelles



Possibilité de créer une structure liste :

La liste se résume au nœud début

```
typedef struct noeud liste;
```

Ou pour pouvoir accéder directement au nœud de fin (que l'on reconnaît aussi car c'est le seul à être son propre suivant) :

```
struct liste {
        struct noeud *debut;
        struct noeud *fin;
};
```

Liste chaînée - Libération mémoire

- Chaque insertion dans la liste donne lieu à une allocation mémoire pour un nouveau nœud.
- Il faut supprimer cet espace lors de la suppression d'un nœud.
- Méthode pour supprimer toute la liste : suppression de l'espace pour tous les nœuds (y compris les nœuds sentinelles).

Liste chaînée
Implantation par tableau simple

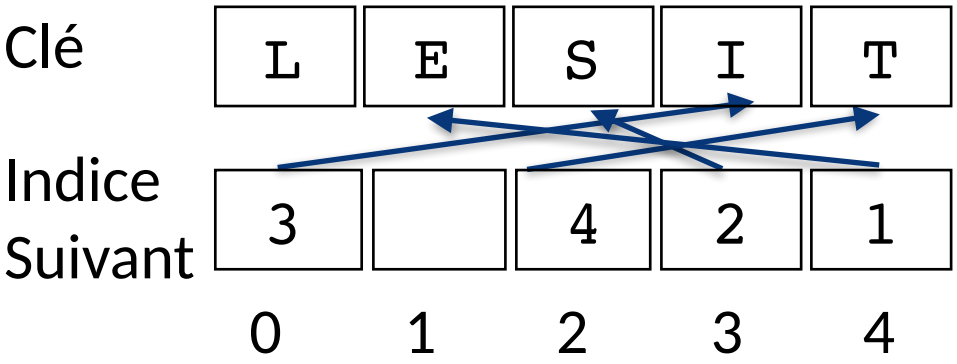
- La liste est représentée sous forme d'un tableau d'éléments contenant une clé et l'indice de l'élément suivant.

```
struct noeud {
     char cle;
     int indiceSuivant;
}
noeud liste[TAILLE_MAX];
// +2 si nœuds sentinelles
```

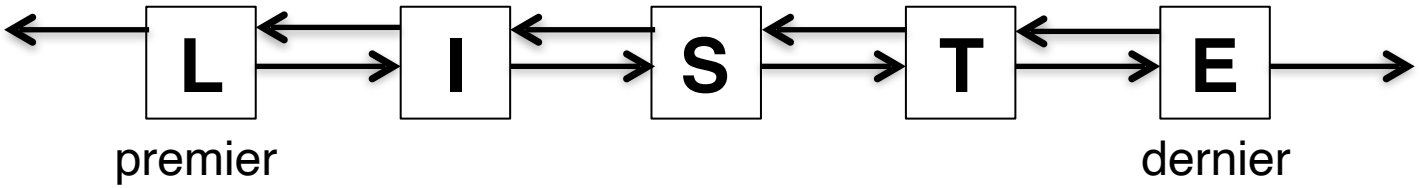
Liste chaînée
Implantation par double tableau

- La liste est représentée sous forme d'un tableau pour les clés et d'un autre tableau pour les indices des suivants.

```
char cles[TAILLE_MAX];
int suivants[TAILLE_MAX];
// +2 si nœuds sentinelles
```



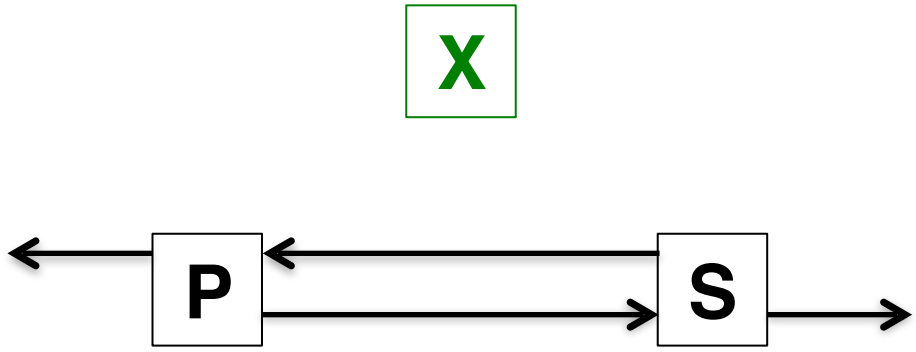
Liste doublement chaînée



- Chaque nœud connaît en plus son nœud précédent. On connaît généralement le premier et le dernier nœud de la liste.
- Avantages
  - parcours dans les deux sens.
  - Possibilité de parcours de toute la liste à partir de n'importe quel nœud.
- Inconvénients :
  - Place mémoire nécessaire plus importante.
  - Primitives plus compliquées (gestion des deux liens)

Liste doublement chaînée – Insertion

1. Création du nœud

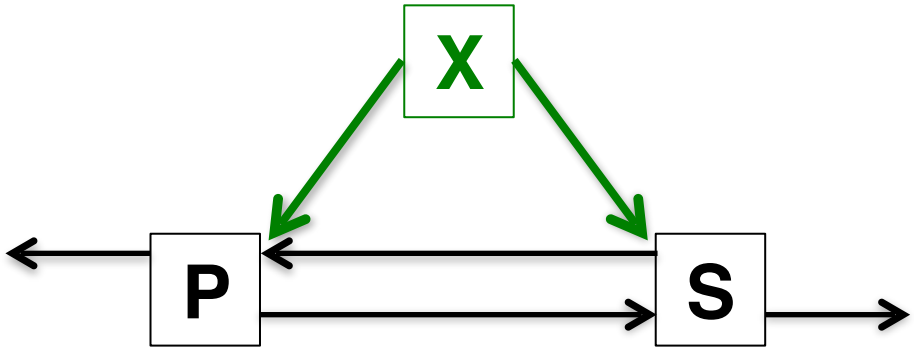


Si l'on connaît p : insertion après **Si l'on connaît s : insertion avant**

création du nœud x

Liste doublement chaînée – Insertion

1. Affectation du suivant et du précédent de R



Si l'on connaît p : insertion après

x->suivant = p->suivant

x->précédent = p

Si l'on connaît s : insertion avant

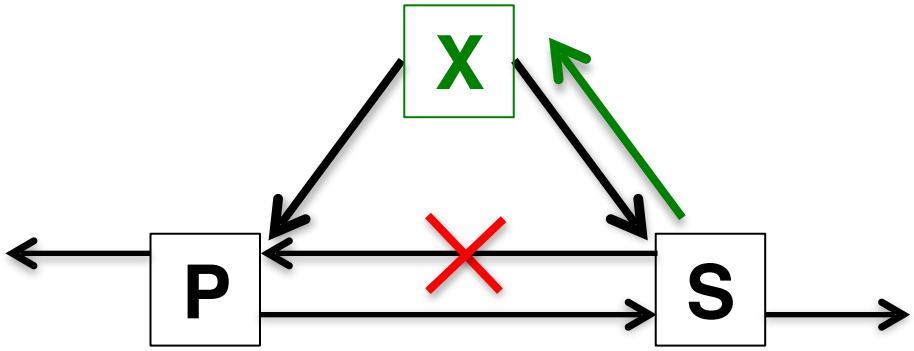
x->suivant = s

x->précédent = s->precedent

création du nœud x

Liste doublement chaînée – Insertion

2. Affectation du précédent de S



Si l'on connaît p : insertion après

création du nœud x

x->suivant = p->suivant

x->précédent = p

Si l'on connaît s : insertion avant

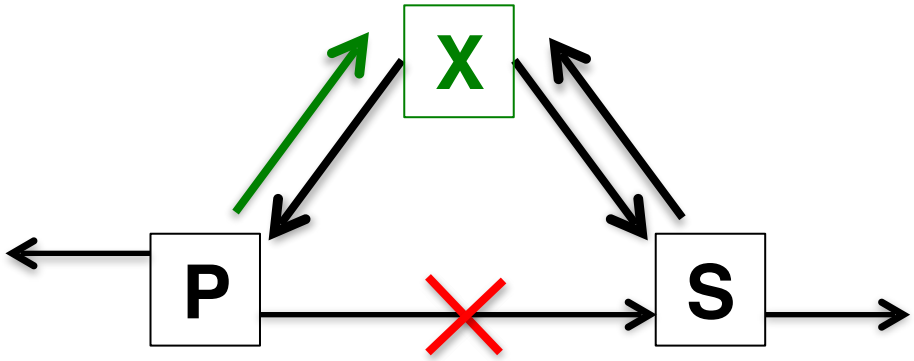
x->suivant = s

x->précédent = s->precedent

x->suivant->precedent = x

Liste doublement chaînée – Insertion

3. Affectation du suivant de P



Si l'on connaît p : insertion après

création du nœud x

x->suivant = p->suivant

x->précédent = p

x->suivant->precedent = x

x->precedent->suivant = x

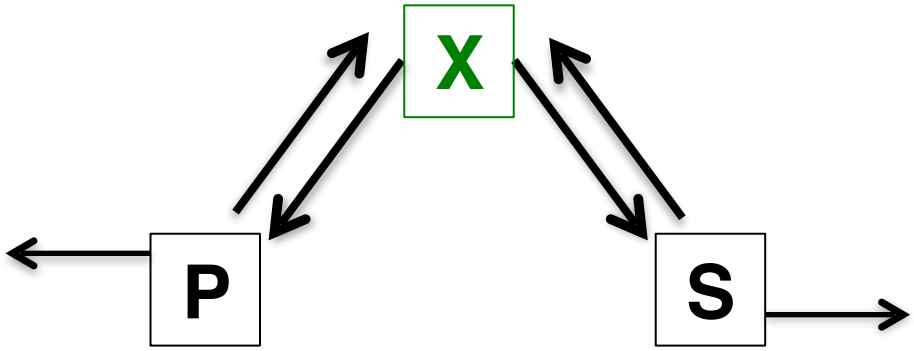
Si l'on connaît s : insertion avant

x->suivant = s

x->précédent = s->precedent

Liste doublement chaînée – Insertion

Récapitulatif



Si l'on connaît p : insertion après

création du nœud x

x->suivant = p->suivant

x->précédent = p

x->suivant->precedent = x

x->precedent->suivant = x

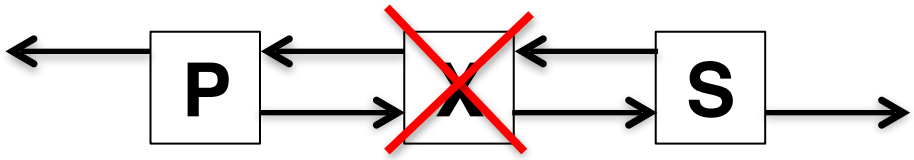
Si l'on connaît s : insertion avant

x->suivant = s

x->précédent = s->precedent

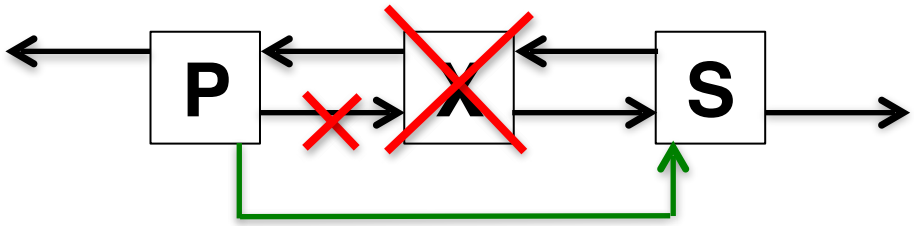
Liste doublement chaînée – Suppression

Suppression du noeud X



Liste doublement chaînée – Suppression

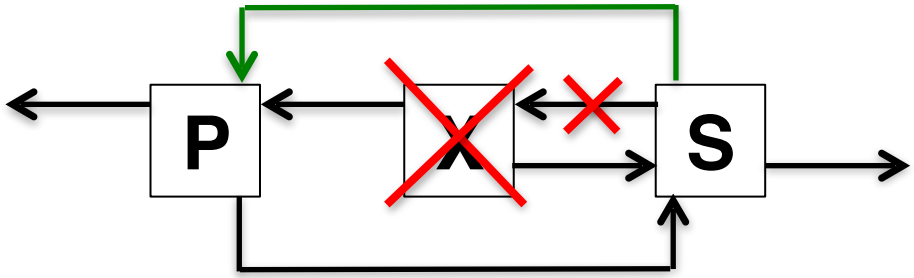
1. Le suivant de X (S) devient le suivant du précédent de X (P)



x->precedent->suivant = x->suivant

Liste doublement chaînée – Suppression

2. Le précédent de X (P) devient le précédent du suivant de X (S)

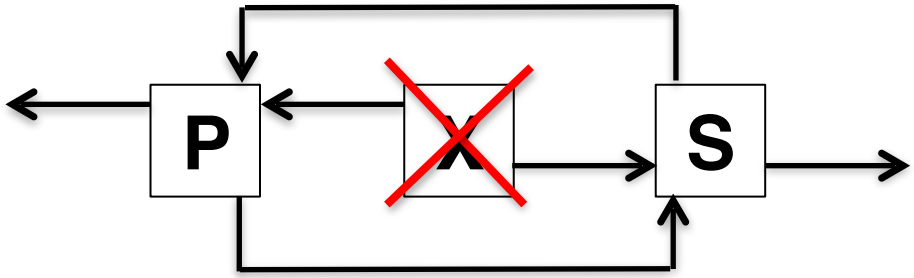


x->precedent->suivant = x->suivant

x->suivant->precedent = x->precedent

Liste doublement chaînée – Suppression

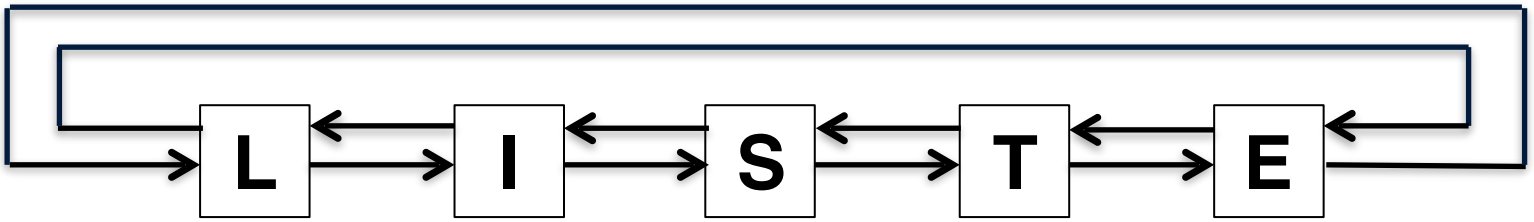
Récapitulatif



x->precedent->suivant = x->suivant

x->suivant->precedent = x->precedent

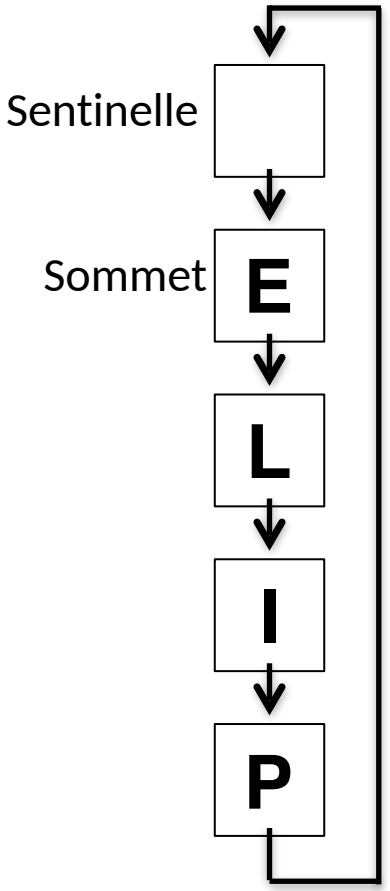
Liste circulaire



- Le dernier et le premier nœud sont reliés entre eux.
- Plus forcément de notion de premier et/ou de dernier.
- Simplement ou doublement chaînée.

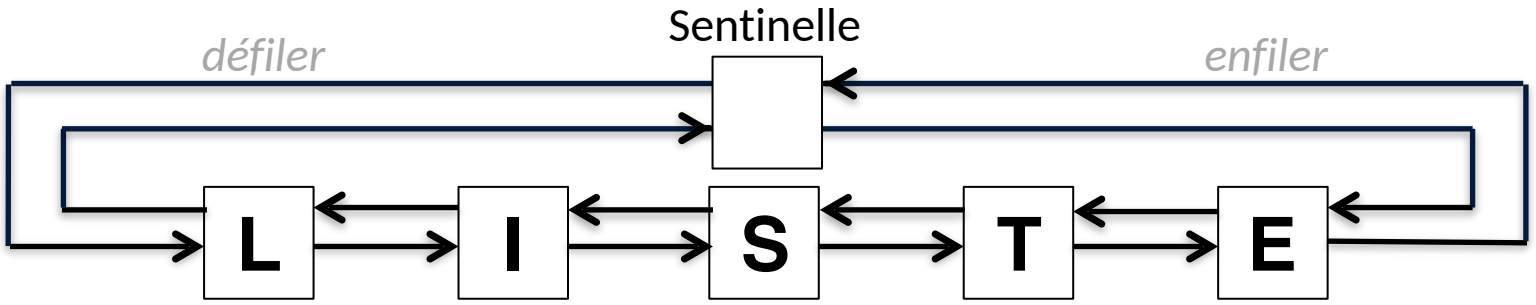
Pile
Implantation par liste chaînée

- Permet une gestion dynamique de la taille de la pile.
- Liste circulaire simplement chaînée avec 1 noeud sentinelle
- créer : noeud sentinelle qui boucle sur lui-même
- empiler : ajout après sentinelle
- dépiler : retrait du suivant de sentinelle
- estVide : sentinelle est son propre suivant



File
Implantation par liste chaînée

- Permet une gestion dynamique de la taille de la file
- Par exemple : liste circulaire doublement chaînée avec 1 noeud sentinelle
- créer : noeud sentinelle qui boucle sur lui-même
- enfiler : ajout après sentinelle.
- défiler : suppression avant sentinelle.
- estVide : sentinelle est son propre suivant (ou précédent)



Exercices

- Implantation par pointeurs d'une **liste simplement chaînée** avec ses deux nœuds sentinelles et des clés de type **unsigned char**.
  - liste* **lcCreerListe**()
  - bool **lcEstVide**(liste * l) (inclure stdbool.h)
  - noeud* **lcInsererApres**(nœud* n, unsigned char cle) // renvoie le noeud créé
  - unsigned char **lcSupprimerSuivant**(nœud* n) // renvoie la clé du nœud supprimé
  - void **lcSupprimerListe**(liste* l)
  - void **lcParcourir**(liste* l) // affichage sous la forme L -> I -> S -> T -> E
  - bool **lcContient**(liste* l, unsigned char cle)
  - noeud * **lcInsererOrdonne**(liste* l, unsigned char cle)
  - bool **lcSupprimerCle**(liste* l, unsigned char cle)
- En option : fusion de deux listes ordonnées : liste * **lcFusionner**(liste* l1, liste* l2)
- Implantation d'une **file de void *** par liste doublement chaînée
- Implantation d'une **pile de void *** par liste simplement chaînée