Algorithmique et structures de données en C

Listes

Enseignant: P. Bertin-Johannet

Une liste

- Une liste est une collection d'éléments.
- Quand sa taille n'est pas modifiable dynamiquement on parle de tableau.
- En C, par défaut il n'existe pas de liste de taille modifiable dynamiquement.

```
int main(){
  int tab[3]; // tab ne contiendra jamais plus que 3
éléments
}
```

Vecteur/ArrayList

Vecteur

- Une manière de créer une liste dont la taille changera dynamiquement est de recopier les élements dans un nouvel emplacement mémoire plus grand lors d'un ajout.
- Dans ce cours, nous appellerons ce type de liste "Vecteur" (le nom choisit dans les langages C++ et Rust).
- Ce type de liste est plus souvent appelé "ArrayList", "List" ou "Array" (ArrayList en Java, List en C#, slice en go, toutes les listes python, etc...).

Vecteur - problème

- Si l'on recopie tous les éléments du tableau à chaque ajout, le nombre de copie augmente rapidement.
- Le nombre de copies effectuées pour remplir une liste de 5 éléments est donc (1 + 2 + 3 + 4 =) **10**.

```
struct ListeInt {
  int* elements; // les éléments contenus
  int nombre_elements; // le nombre d'éléments présents
}
```

Vecteur - solution

- Afin d'éviter de recopier l'integralité des éléments à chaque ajout, on peut réserver plus d'espace que nécéssaire.
- Pour implémenter cette version, on aura besoin d'enregistrer le nombre d'éléments courant ainsi que l'espace disponible.

```
struct ListeInt {
  int* elements; // les éléments contenus
  int taille_reservee; // la taille reservée
  int nombre_elements; // le nombre d'éléments présents
}
```

Recherche dans un vecteur

- Lorsqu'on cherche un élément dans le tableau, on doit toujours vérifier que l'on ne dépasse pas la taille maximale.
- Dans l'exemple ci-dessous, on effectue deux conditions à chaque passage de boucle.

```
int cherche_lettre(ListeChar liste, char a){
  int i = 0;
  while (a != liste.elements[i] && i < liste.nombre_elements){
    i++;
  }
  return i; // si pas trouvé, i == nombre_elements - 1
}
```

Element Sentinelle

- Pour réduire à une seule condition, on peut rajouter un élément appelé “sentinelle” à la fin.

```
int cherche_lettre(ListeChar liste, char a){
  liste.elements[liste.nombre_elements] = a;
  int i = 0;
  while (a != liste.elements[i]){
    i++;
  }
  return i;
}
```

Vecteur - Coût des opérations

- Pour estimer le coût des opérations sur un vecteur, nous regarderons la complexité asymptotique.
- Cela signifie grossièrement que nous considèrerons le nombre d'opération selon n et garderons le facteur qui "grandit" le plus.
- Par exemple pour accéder à l'élément n d'un tableau, il suffit de faire une addition
- Le nombre d'opérations ne changeant pas quelque soit n, on dit que la complexité est **constante** (ou $O(1)$).

```
int get_value_at(ListeInt liste, int i){
  return *(liste.elements + i); // ou liste.elements[i]
}
```

Vecteur - Coût de l'insertion

- Pour calculer la complexité, on considère toujours le cas avec le plus d'opérations.
- L'insertion d'un element dans un vecteur demande, dans le pire des cas, de recopier l'intégralité du vecteur dans un nouvel emplacement.
- Il y a alors n opérations pour une liste de taille n.
- On parle alors de compléxité **linéaire** (ou $O(n)$).

```
int insert_valeur_fin(ListeInt liste, int i){
  int* ancienne_addresse = liste.elements;
  liste.elements = malloc(sizeof(int) * (liste.taille + 1));
  memcpy(liste.elements, ancienne_addresse, sizeof(int) * liste.taille)
  liste.elements[liste.taille++] = i;
  free(ancienne_addresse);
}
```

Vecteur - Coût de la suppression

- Lorsqu'on supprime un élément dans un vecteur, on doit déplacer tous les éléments qui le suivent d'une case en arrière.
- La suppression d'un élément peut donc nécéssiter n opération.
- C'est encore une complexité **linéaire** (ou $O(n)$)

```
void supprime_element(ListeInt liste, int n){
  liste.nombre_elements--;
  for (int i = n; i < liste.nombre_element; i++){
    liste.elements[i] = liste.elements[i + 1];
  }
}
```

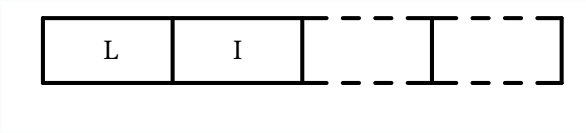
Vecteur - Coût de la recherche

- Pour chercher un élément dans un vecteur, on le parcours au maximum une fois.
- La complexité est donc encore **linéaire** (ou $O(n)$)

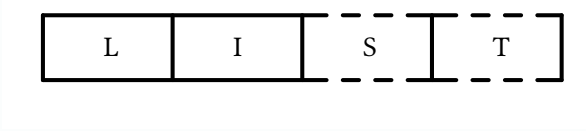
```
int cherche_lettre(ListeChar liste, char a){
  liste.elements[liste.nombre_elements] = a;
  int i = 0;
  while (a != liste.elements[i]){
    i++;
  }
  return i;
}
```

Complexité dans le pire des cas - limites

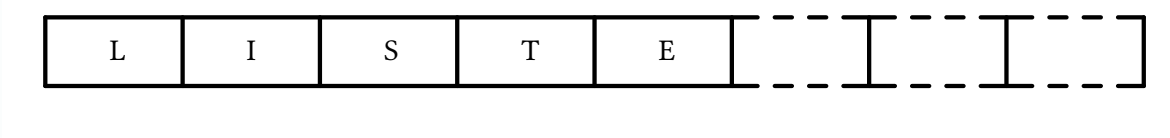
Soit le tableau suivant de taille 2 auquel on a alloué 4 octets et qui contient les lettres L et I.



- Ajouter S et T ne demande pas de recopier les éléments.



- Ajouter E demande de recopier L I S et T dans un nouveau tableau.



Coût amorti

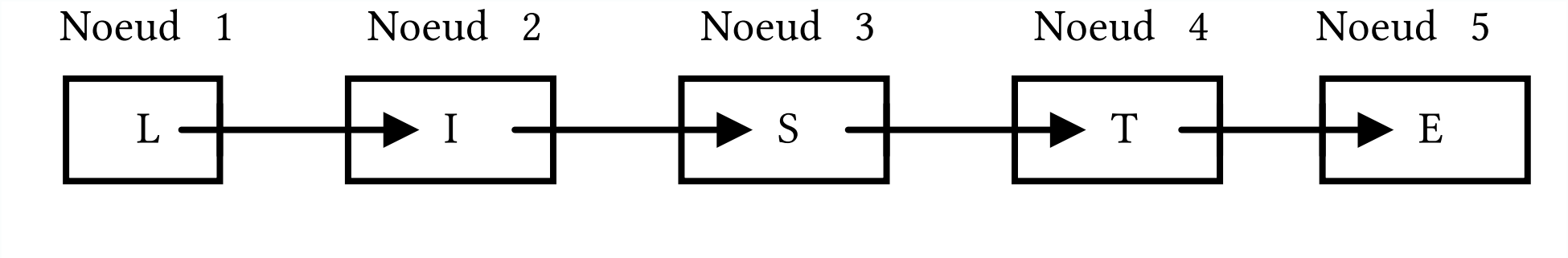
- Dans le cas précédent, l'ajout des deux premiers éléments a demandé **0 copies**.
- Au total, l'ajout de **3 éléments** a demandé **4 copies** seulement.
- Le coût moyen de l'ajout dans ce tableau n'est donc pas le même que le coût dans le pire des cas.
- On parle dans ce cas de **coût amorti**.
- Bien que la complexité soit **linéaire** (ou $O(n)$), la complexité amortie est **constante**.

Nombre d'éléments:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Copies pour ajouter:	0	1	0	0	4	0	0	0	8	0	0	0	0	0	0	0	16	0
Copies totales:	0	1	1	1	5	5	5	5	13	13	13	13	13	13	13	13	29	29
Copies moyennes:	0	0.5	0.3	0.25	1	0.8	0.7	0.6	1.4	1.3	1.2	1.1	1	0.9	0.9	0.8	1.7	1.6

Listes chainées

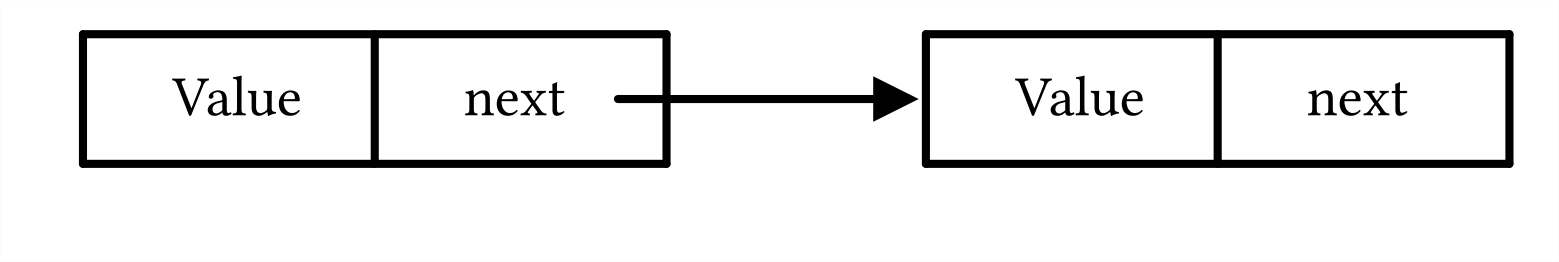
Liste chainée

- Une liste chainée est une liste d'éléments contenant chacun une valeur ainsi que l'addresse du noeud suivant.
- Il sera ainsi possible de parcourir la liste en suivant les liens d'un noeud vers l'autre.



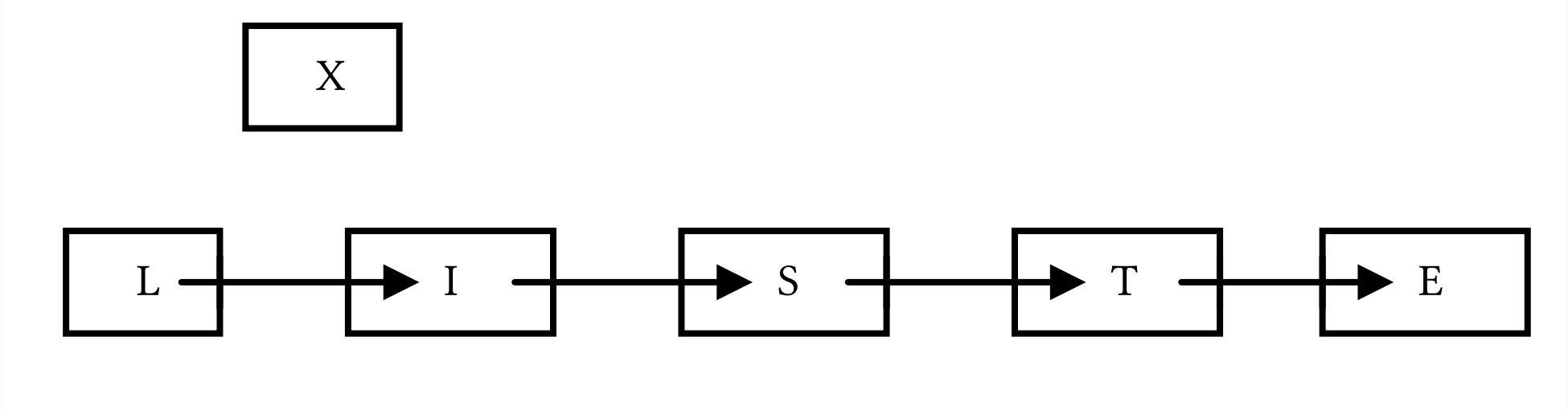
Liste chainée - Première implémentation

```
struct ListElem {
  int value;    // la valeur de l'élément
  ListElem* next; // un pointeur vers le prochain élément
}
```



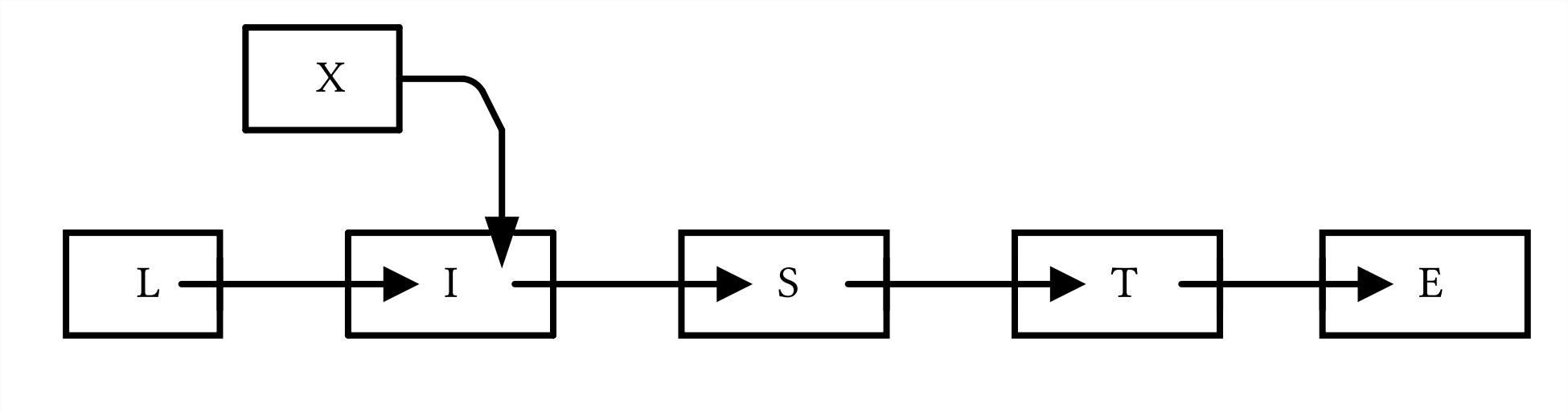
Liste chainée - Insertion

- On veut ajouter X entre L et I



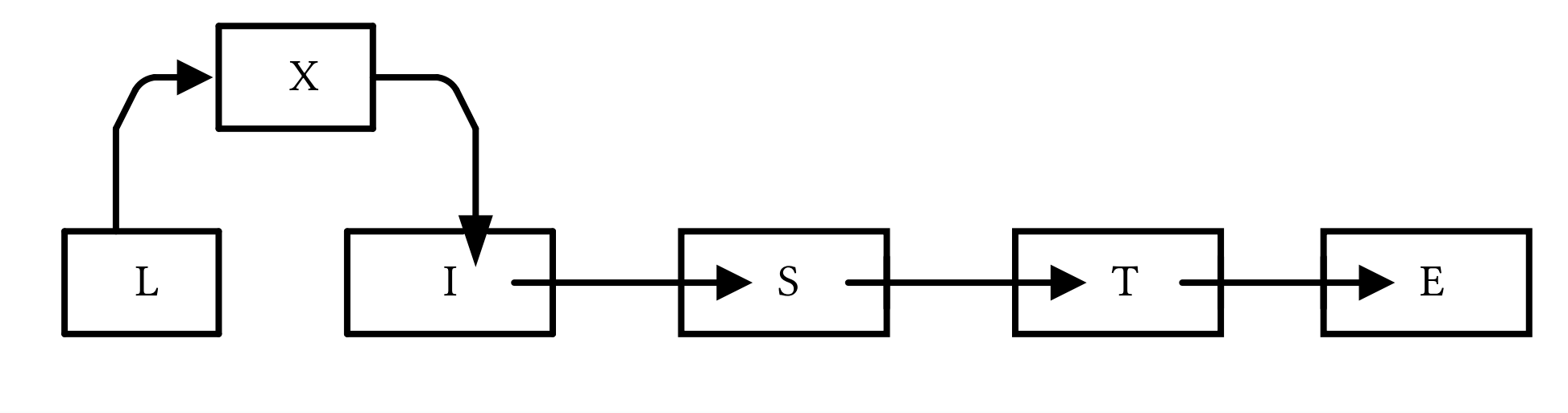
Liste chainée - Insertion

1. On change le suivant de X pour qu'il pointe vers I



Liste chainée - Insertion

1. On change le suivant de X pour qu'il pointe vers I
2. On change le suivant de L pour qu'il pointe vers X

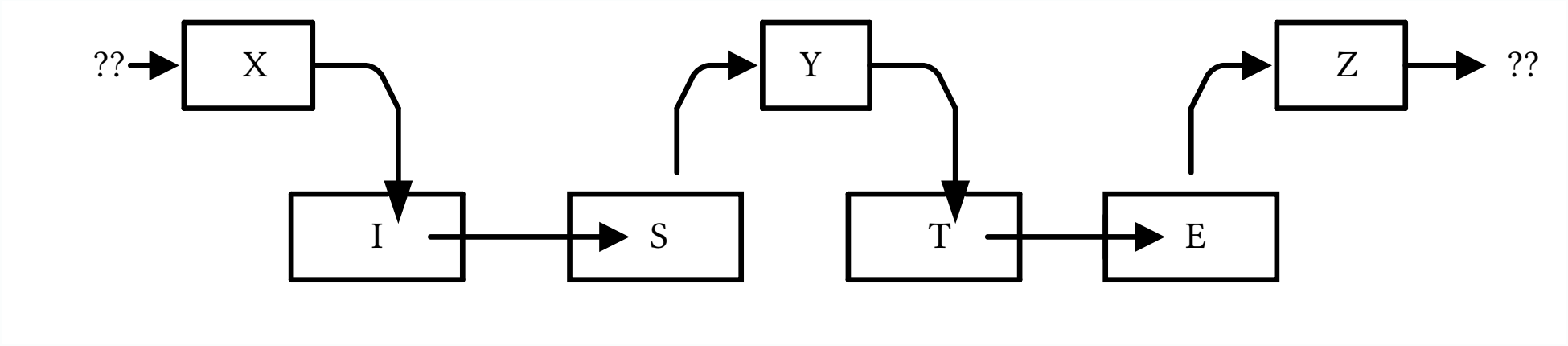


Liste chainée - Coût de l'insertion

- Pour calculer la complexité, on considère toujours le cas avec le plus d'opérations.
- L'insertion d'un element dans une liste chainée demande le même nombre d'opérations quelque soit sa taille
- On parle alors de compléxité constante (ou $O(1)$).

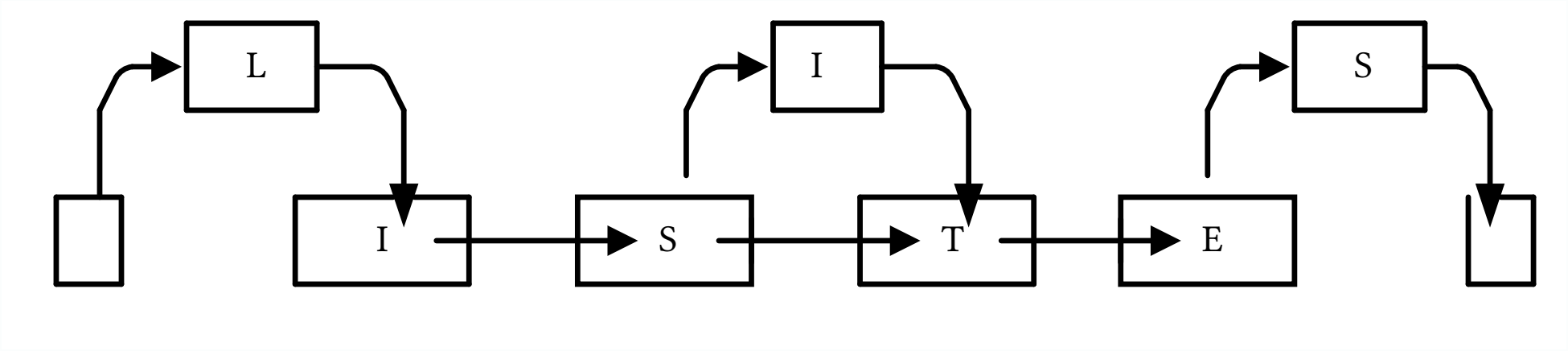
Liste chainée - Opérations en début et fin

- Certaines opérations sur un élément fonctionnent différement si il est positionné en début, en milieu ou en fin de liste
- Par exemple pour l'insertion et la suppression on doit gérer le cas ou le noeud précédent ou suivant n'existe pas



Liste chainée - Elements sentinelle

- Pour simplifier l'usage de la liste, on peut ajouter des elements sentinelles en début et fin de liste.



Liste chainée - Seconde implémentation

- La liste chainée doit donc contenir un pointeur vers les deux éléments sentinelles

```
struct ListElem {
  int value;    // la valeur de l'élément
  ListElem* next; // un pointeur vers le prochain élément
};
typedef struct ListElem ListElem;
struct LinkedList {
  ListElem sentinelStart; // la sentinelle de début
  ListElem sentinelEnd; // la sentinelle de fin
}
```

Liste chainée - Recherche d'un élément

- L'élément sentinelle de fin peut être utilisé pour accélérer la recherche d'une valeur dans la liste.

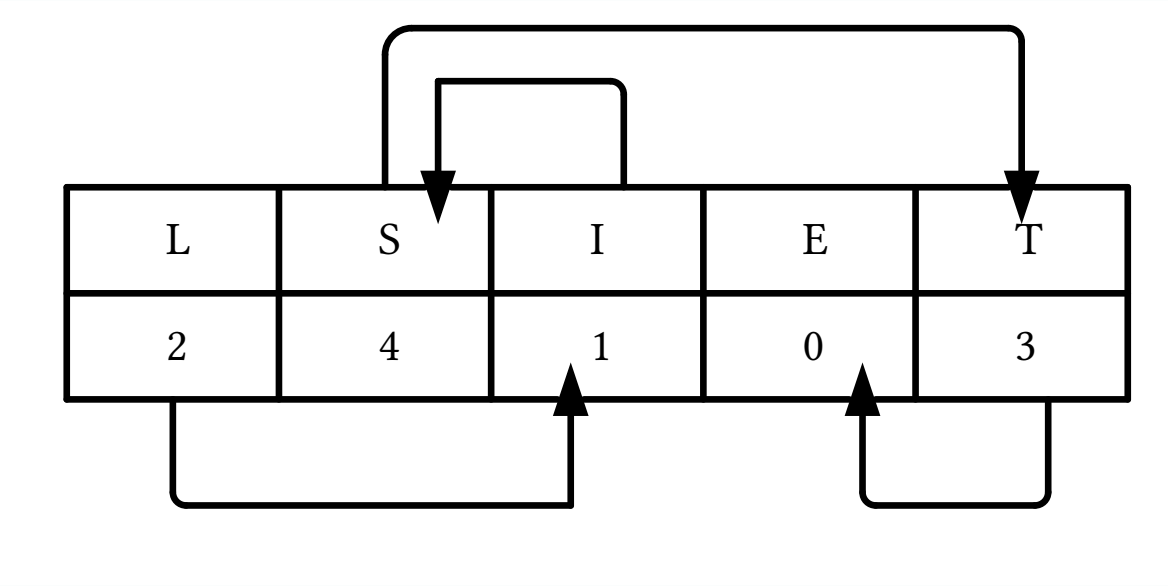
```
void trouve_elem(LinkedList l, int toFind){
  l.sentinelEnd.value = toFind; // on met la valeur cherchée dans l'élément de fin
  ListElem* elem = l.sentinelStart->next; // on commence par le premier noeud
  while (elem->value != toFind){ // tant qu'on a pas trouvé un noeud avec la valeur
    elem = elem->next; // on avance dans la liste
  }
  if (elem == &(l.sentinelEnd)){
    printf("l'element n'est pas dans la liste");
  }
  printf("l'élément est dans la liste");
}
```

Inconvénients de la liste chainée

- Malgré les avantages précédement abordés, une liste chainée rencontre des problèmes importants de performances sur les machines réelles.
    1. Manque de localité mémoire : les noeuds sont dispersés dans la mémoire et il est difficile de tous les enregistrer dans le cache.
    2. Consommation mémoire accrue : chaque élément requiert un espace supplémentaire pour le pointeur suivant.
    3. Indirection pointeur : chaque accès implique une étape de déréférencement du pointeur suivant.
    4. Prédiction d'accès difficile : le processeur peut difficilement prédire le prochain élément utilisé.
    5. Vectorisation des opérations difficile : Executer une opération sur plusieurs noeuds en même temps est difficile.
- Ces problèmes peuvent ralentir le code d'un facteur allant jusqu'a **plusieurs milliers**.

Liste chainée - Implémentation par tableau

- On peut aussi implémenter une liste chainée en utilisant deux tableaux.
- Un tableau contiendra les indices des elements suivants et un autre tableau contiendra les valeurs.

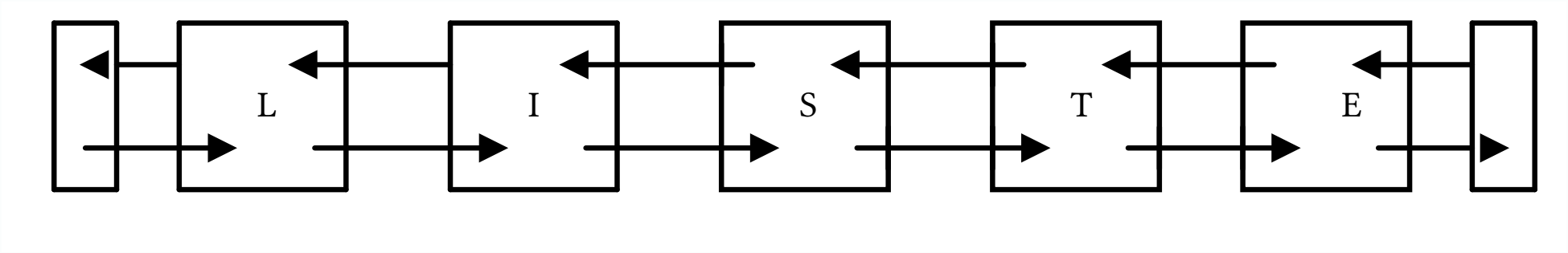


Avantages et désavantages de l'implémentation par tableau

1. Avantages:
   - Meilleure localité des valeurs que l'implémentation précédente.
   - Les indices occupent moins d'espace qu'un pointeur.
   - On peut réutiliser un vecteur pour allouer de l'espace pour plusieurs noeuds en avance.
   - La suppression reste en temps constant.
2. Désavantages
   - L'insertion devient une opération en temps linéaire car on peut avoir à recopier le contenu quand on réalloue.

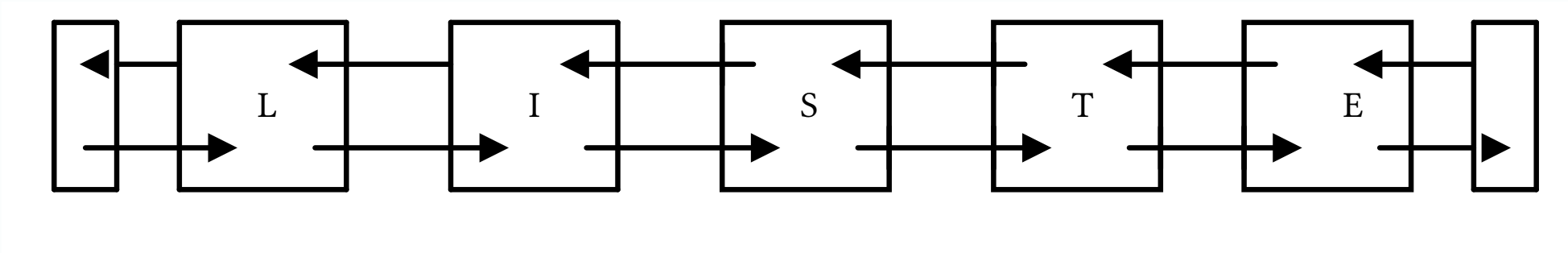
Liste doublement chainée

- On parle de liste doublement chainée lorsque chaque élément contient aussi un pointeur vers le précédent.
- Les opérations de suppression et d'insertion nécéssitent plus d'opérations.
- On garde toujours deux éléments sentinelles.



Avantages et désavantages de la liste doublement chainée

1. Avantages:
   - Cela permet permet d'effectuer des parcours dans les deux sens.
   - On peut insérer et supprimer à la fin de la liste en temps constant.
2. Désavantages
   - L'usage de deux fois plus de pointeurs implique un usage mémoire accru.
   - Le nombre d'instructions augmente.



Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début			
Accès à la fin			
Accès arbitraire			
Insertion à la fin			
Insertion au début			
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin			
Accès arbitraire			
Insertion à la fin			
Insertion au début			
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire			
Insertion à la fin			
Insertion au début			
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin			
Insertion au début			
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin	Linéaire	Linéaire	Constante
Insertion au début			
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin	Linéaire	Linéaire	Constante
Insertion au début	Linéaire	Constante	Constante
Insertion arbitraire			
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin	Linéaire	Linéaire	Constante
Insertion au début	Linéaire	Constante	Constante
Insertion arbitraire	Linéaire	Linéaire	Linéaire
Suppression au début			
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

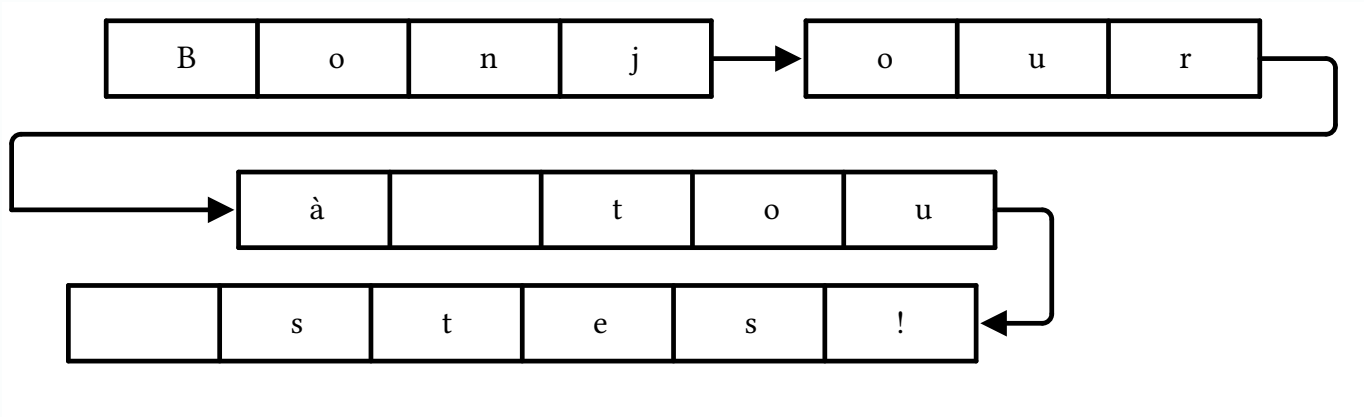
Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin	Linéaire	Linéaire	Constante
Insertion au début	Linéaire	Constante	Constante
Insertion arbitraire	Linéaire	Linéaire	Linéaire
Suppression au début	Linéaire	Constante	Constante
Suppression arbitraire			

Comparaison Vecteur - Liste Chainées

Complexités pour:	Vecteur	Liste chainée	Liste doublement chainée
Accès au début	Constante	Constante	Constante
Accès à la fin	Constante	Linéaire	Constante
Accès arbitraire	Constante	Linéaire	Linéaire
Insertion à la fin	Linéaire	Linéaire	Constante
Insertion au début	Linéaire	Constante	Constante
Insertion arbitraire	Linéaire	Linéaire	Linéaire
Suppression au début	Linéaire	Constante	Constante
Suppression arbitraire	Linéaire	Linéaire	Linéaire

Liste chainée “déroulée”

- Afin d’obtenir un compromis entre les deux structures précédentes on peut créer une liste chainée dite “déroulée” (unrolled ou blocked en anglais).
- On découpe ainsi la liste en noeuds contenant chacun un petit vecteur.
- Lorsqu’un vecteur devient trop grand, on le sépare en deux vecteurs de plus petite taille.

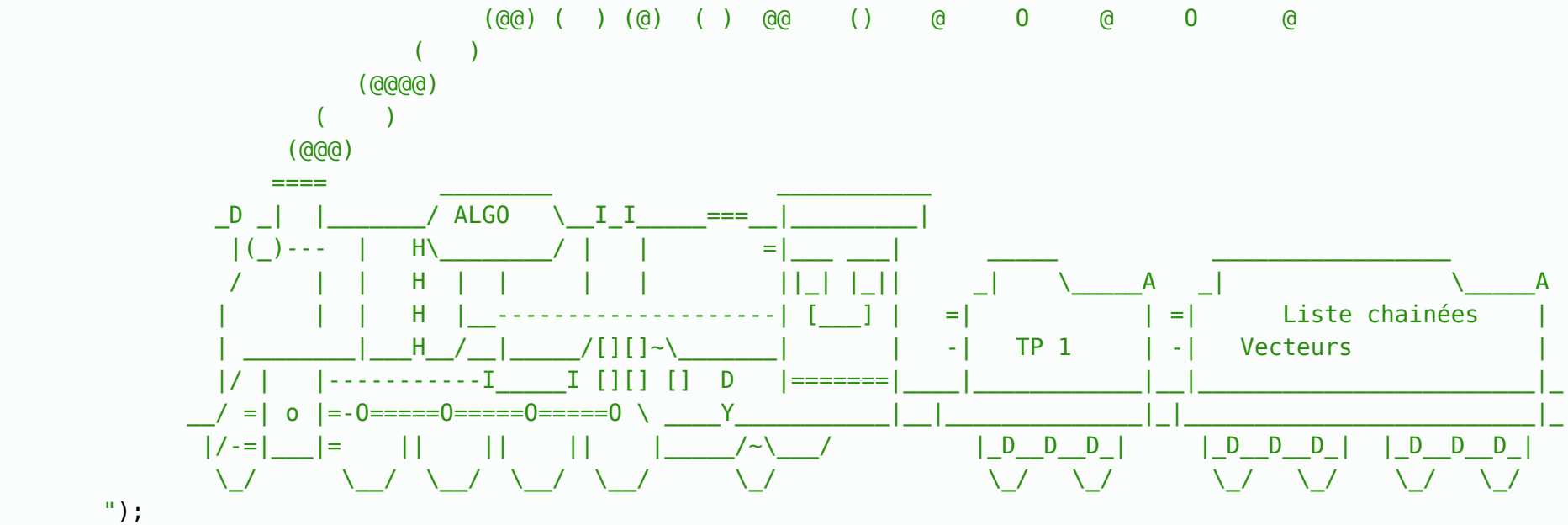


Avantages et désavantages de la liste déroulée

1. Avantages:
   - Meilleure localité des données qu'une liste chainée.
   - Moins de mémoire utilisée qu'une liste chainée (un seul pointeur par vecteur).
   - Facilité de prédiction accrue pour le procésseur.
   - L'insertion/la suppression d'un élément ne demande que la copie des éléments d'un petit vecteur.
2. Désavantages
   - L'Accès n'est toujours pas en temps constant.
   - L'usage mémoire est superieur à celui d'un vecteur.
   - L'ajout d'un élément peut déclencher la séparation d'un vecteur en deux, une opération plus coûteuse que pour une liste chainée.

Mise en pratique

```
printf("
```




```
");
```