ECE PARIS • LYON
ÉCOLE D'INGÉNIEURS

Algorithmique et programmation structurée en C

Cours 9

Langage C
Structures de données
Listes Chaînées

Antoine Hintzy

Structures de données

On appelle **structure de données** une façon d'organiser/de structurer les données afin de les traiter plus facilement.

Une structure de données est généralement composée :

- D'un **type de données** :

  Exemples : `struct Vaisseau`, `char chaine[50]`, etc.
- Et éventuellement de **méthodes** permettant d'exploiter les données :

  Exemples : `initVaisseau()`, `strcpy()`, etc.

Structures de données déjà vues

- Les **structures**
- méthodes : fonctions d'initialisation, d'affichage, etc.
- Les **tableaux** *(à une ou deux dimension(s))*
- méthodes : fonctions de tri, de recherche, d'affichage, etc.
- Les **chaînes de caractères**
- méthodes : fonctions de concaténation, de copie, etc. (**`string.h`**)

Nouvelles structures de données

- Les **listes** (chaînées)
- Les **files** (d'attente) : 🇬🇧 FIFO (*First In First Out*)
- Les **piles** (d'assiettes) : 🇬🇧 LIFO (*Last In First Out*)
- Les **arbres**
- Les **graphes**
- etc.

Ces structures de données visent à regrouper un ensemble de données sous une certaine forme qui les rend plus facilement exploitables.

Liste chaînée

- Ensemble de **maillons** reliés entre eux par des pointeurs.

- Chaque maillon est une structure qui contient des données (entiers, chaînes de caractères, tableaux, structures...), et un ou plusieurs pointeurs sur d'autres maillons.

Liste chaînée - Maillon Déclaration

- Définition d'un maillon :

```
typedef struct Maillon {
    // données :
    char firstname[100];
    char lastname[100];
    // pointeur vers le maillon suivant :
    struct Maillon* next;
} Maillon;
```

⚠ Dans la définition d'une structure, nous ne pouvons pas encore utiliser son alias de type créé avec **`typedef`**. Nous pouvons en revanche utiliser le type complet, qui est en cours de création : ici, **`struct Maillon`**.

Liste chaînée - Maillon Déclaration

⊙ Déclaration d'une liste chaînée :

```
int main(void) {
  Maillon* liste = NULL; // contiendra l'adresse du premier maillon
  return 0;
}
```

⚠ Il est important d'initialiser à **NULL** chaque pointeur, pour éviter de récupérer une ancienne valeur en mémoire. En effet, le parcours d'une liste chaînée se fait de maillon en maillon tant que le pointeur du maillon parcouru n'est pas **NULL**.

Liste chaînée - Maillon Déclaration

Création du premier maillon

```
int main(void) {
   Maillon* liste = NULL;

   liste = (Maillon*) malloc(sizeof(Maillon));
   strcpy(liste->firstname, "Toto");
   strcpy(liste->lastname, "Titi");
   liste->next = NULL;

   return 0;
}
```

Liste chaînée - Maillon Déclaration

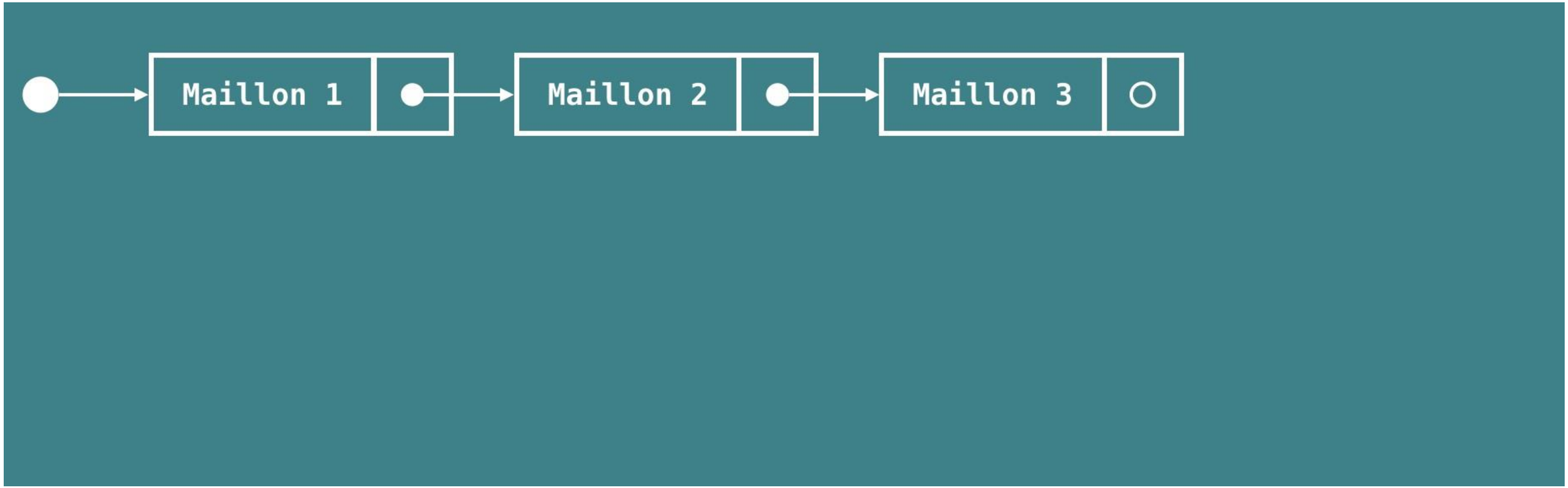
Création du deuxième maillon

```
int main(void) {
Maillon* liste = NULL;

   // Création du premier maillon..., puis :

   liste->next = (Maillon*) malloc(sizeof(Maillon));
   strcpy(liste->next->firstname, "Tata");
   strcpy(liste->next->lastname, "Tutu");
   liste->next->next = NULL;

   return 0;
}
```

Rappel : `liste->next` est équivalent à `(*liste).next`, et correspond au pointeur `next` du premier maillon.

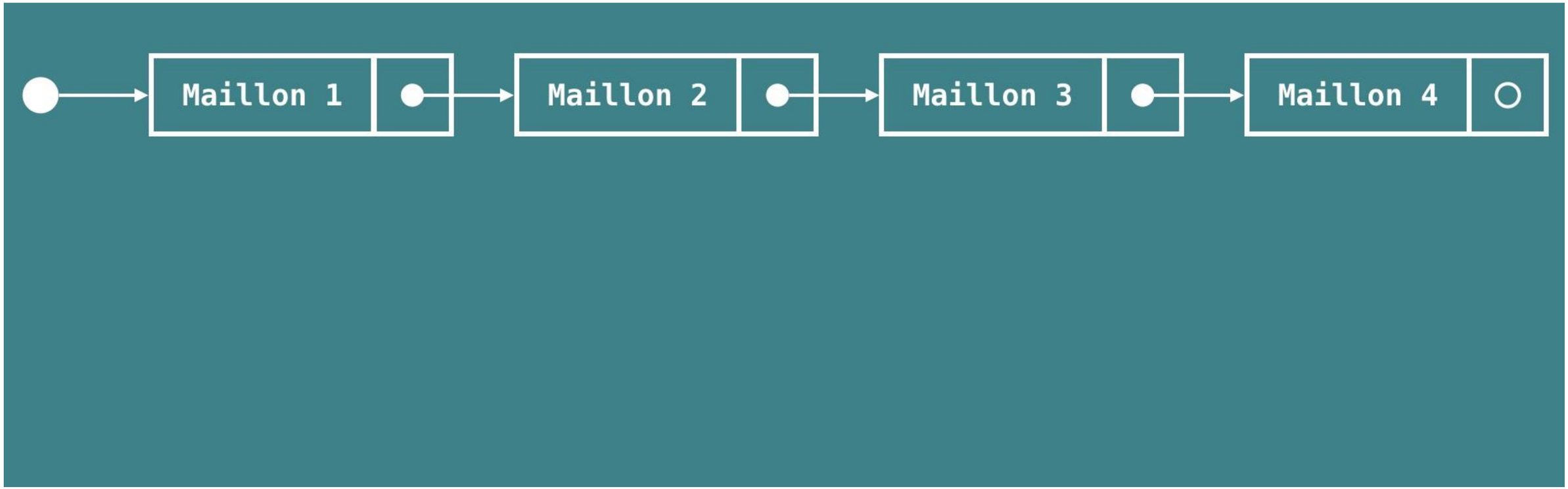
Insertion en tête de liste

Insertion en fin de liste

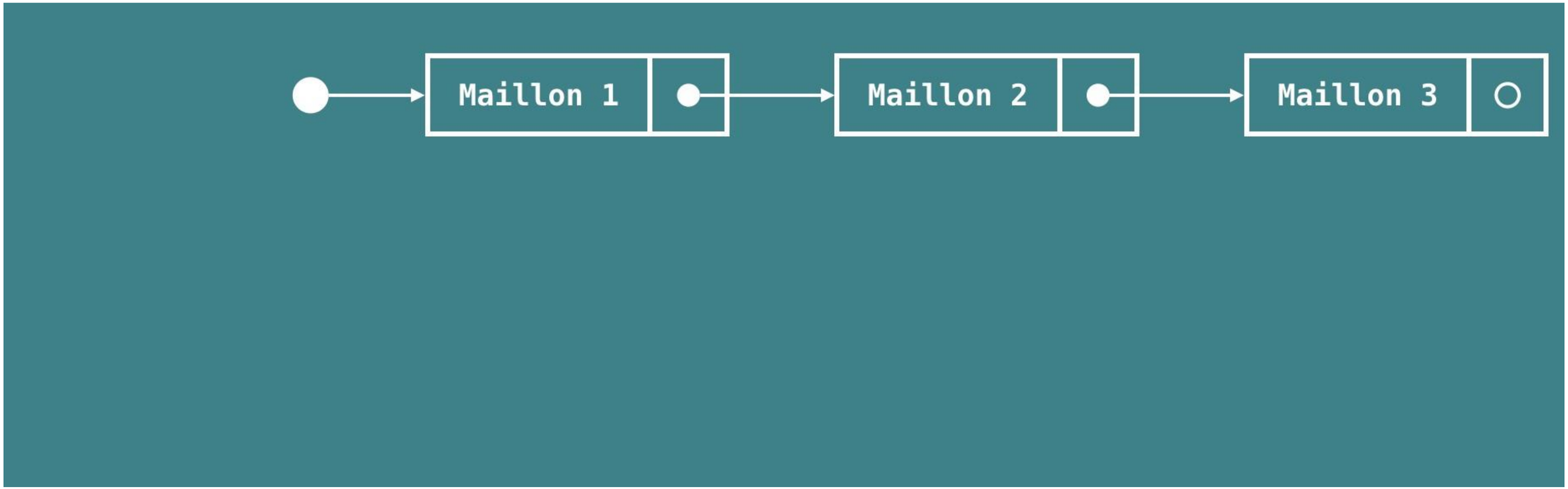


Insertion en fin de liste

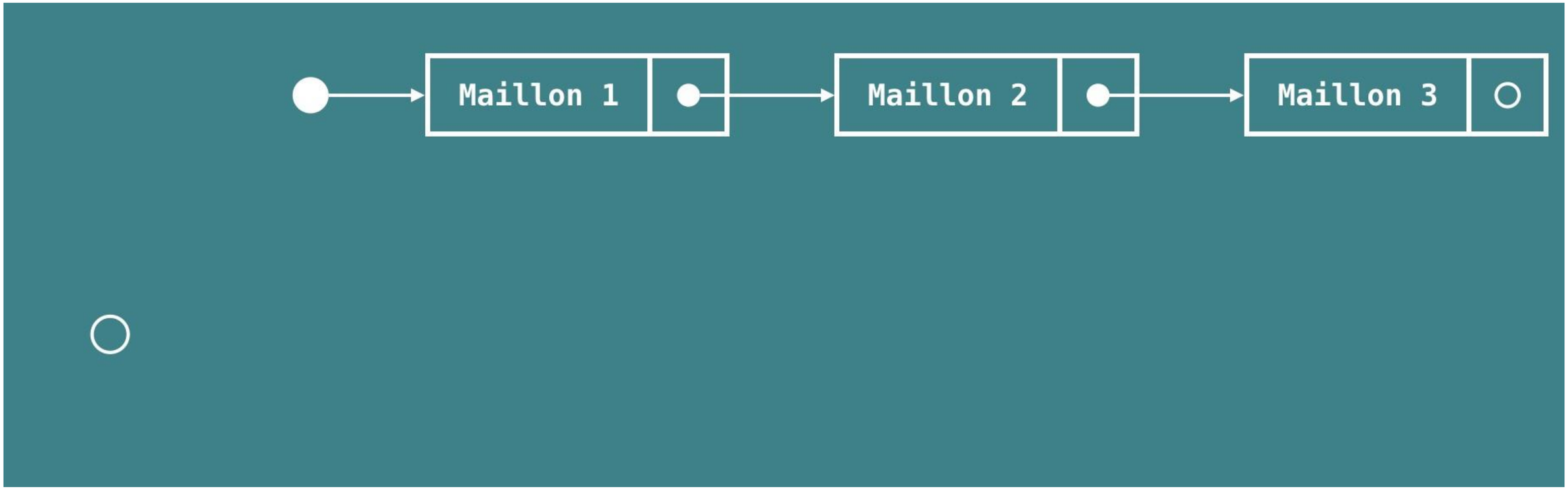
Insertion en tête de liste



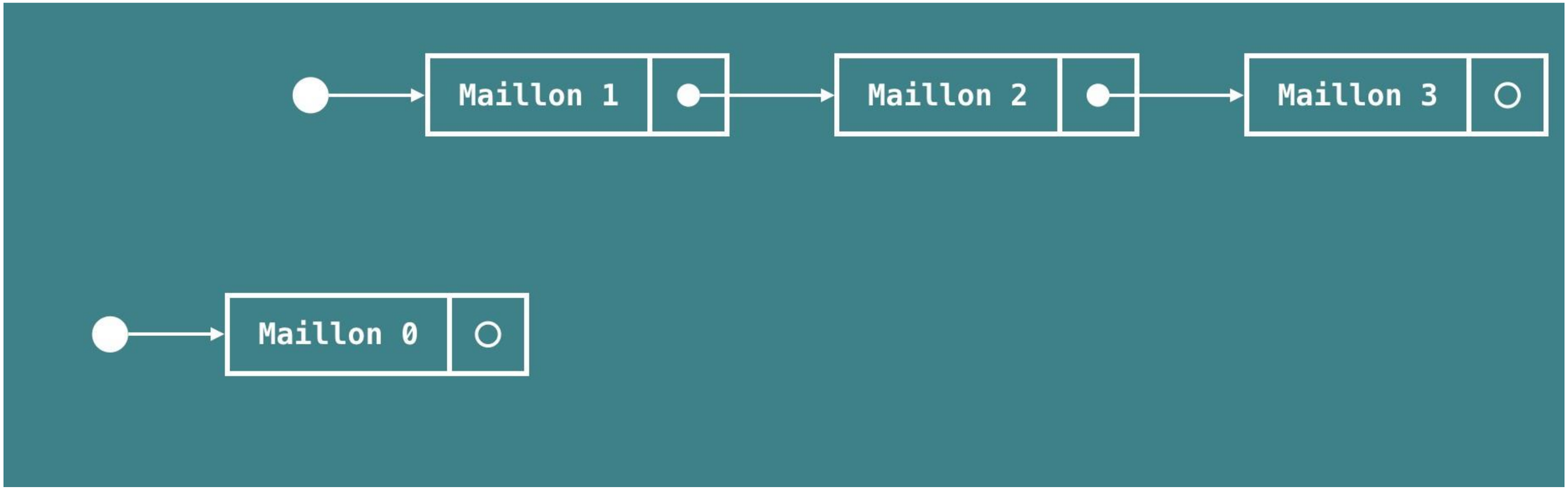
Insertion en tête de liste



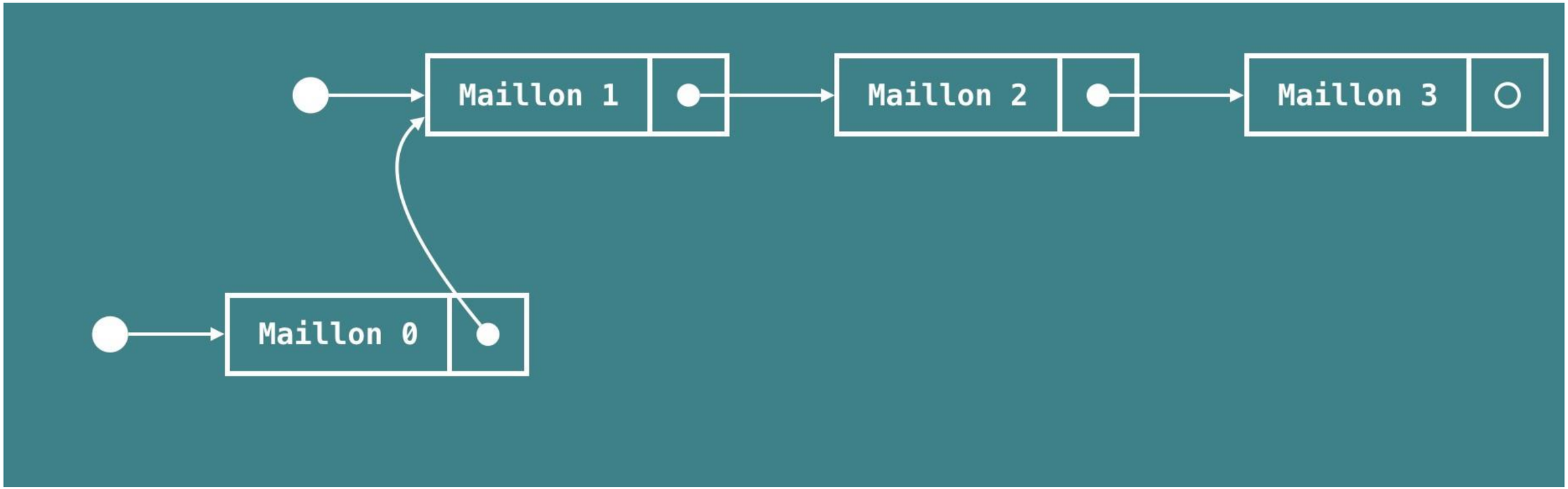
Insertion en tête de liste



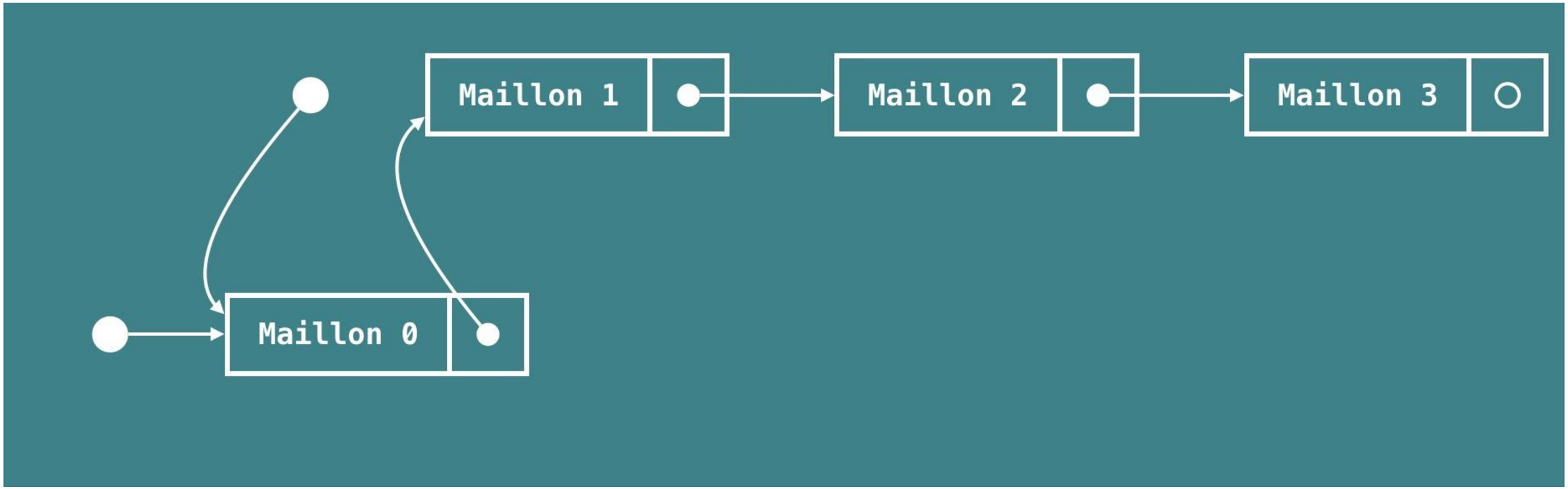
Insertion en tête de liste



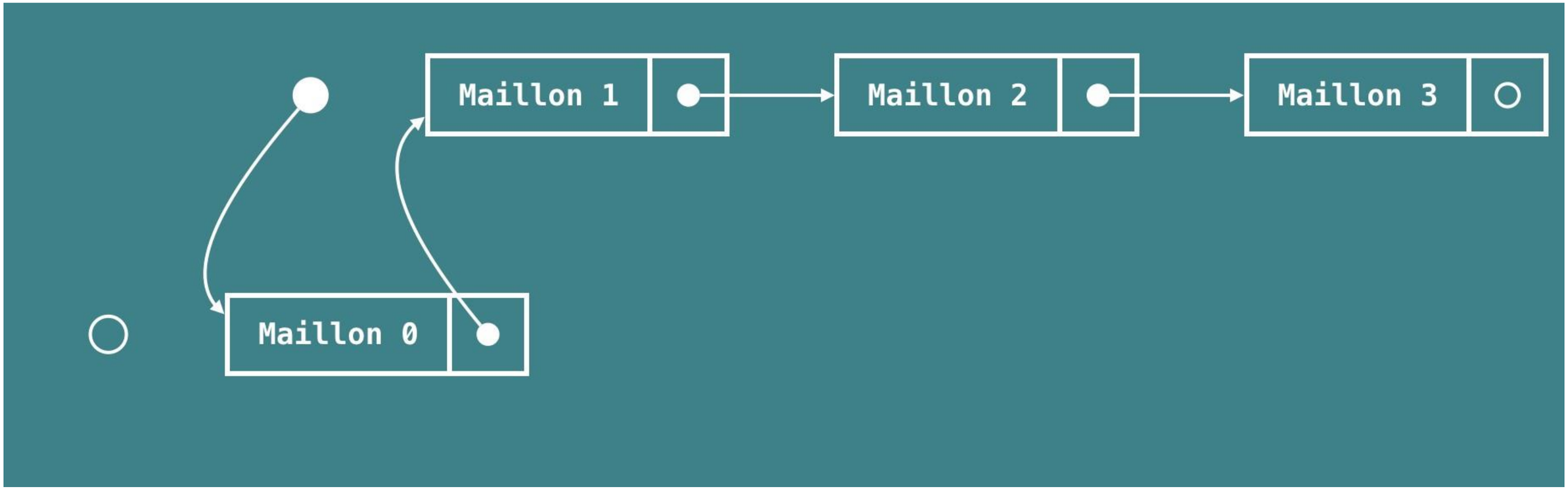
Insertion en tête de liste



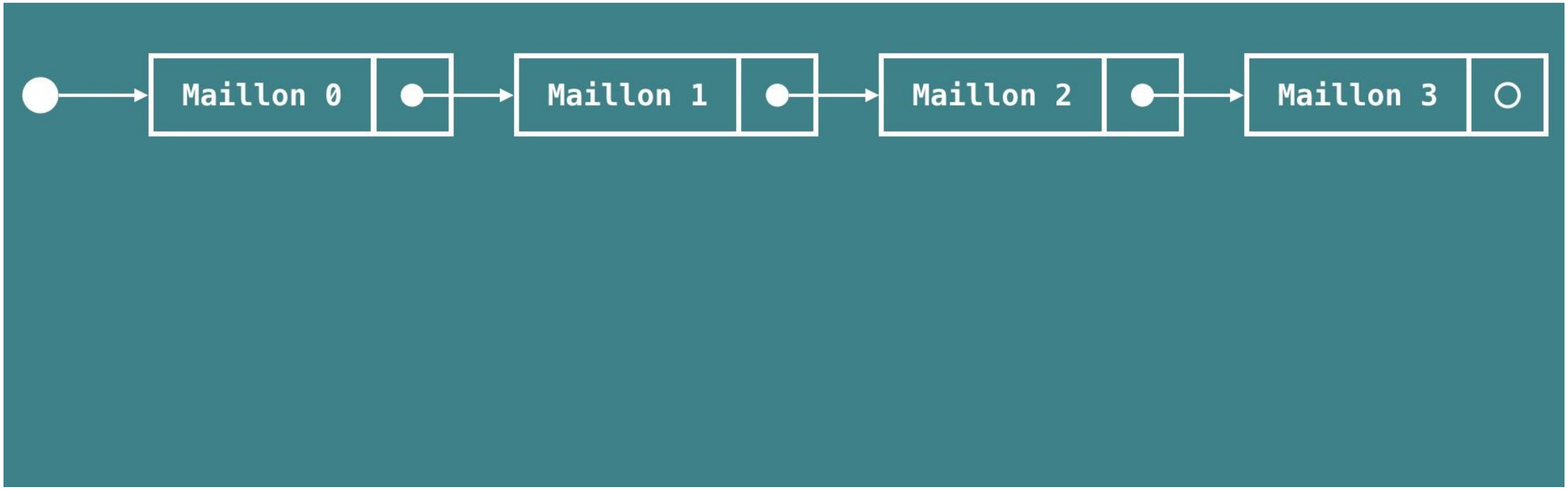
Insertion en tête de liste



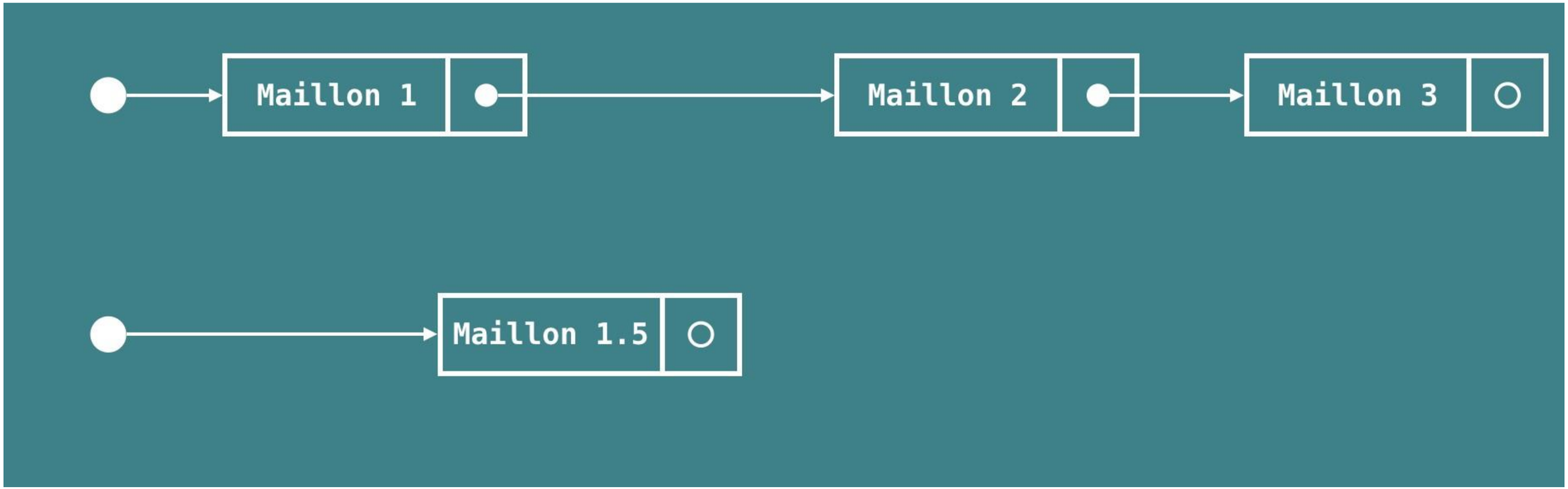
Insertion en tête de liste



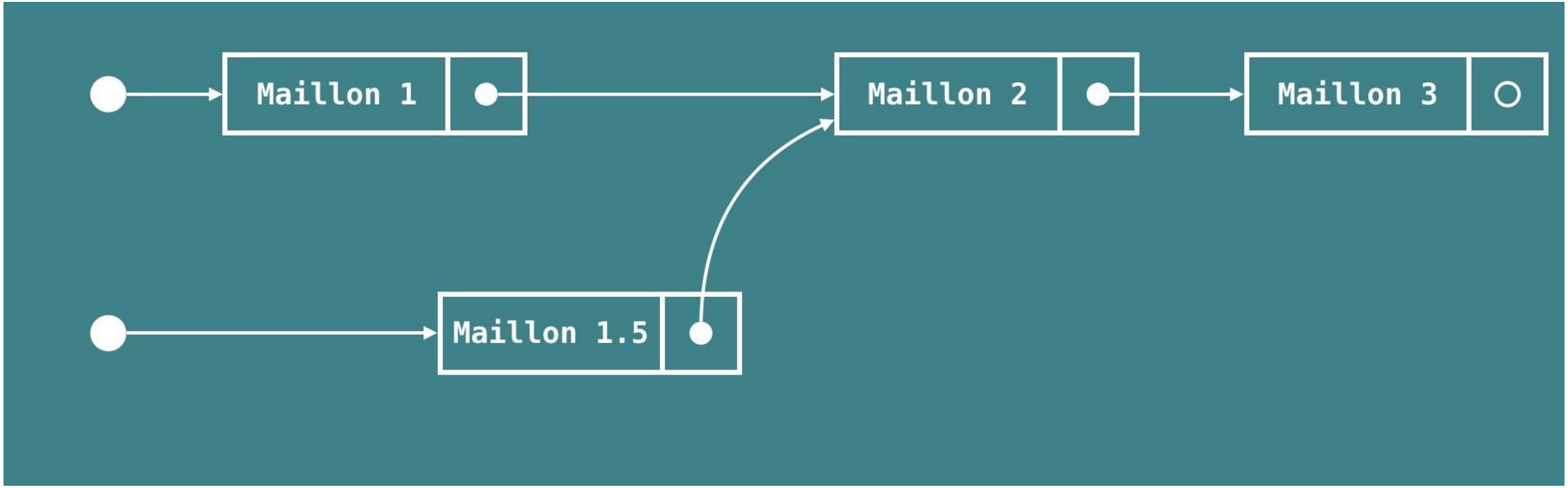
Insertion en tête de liste



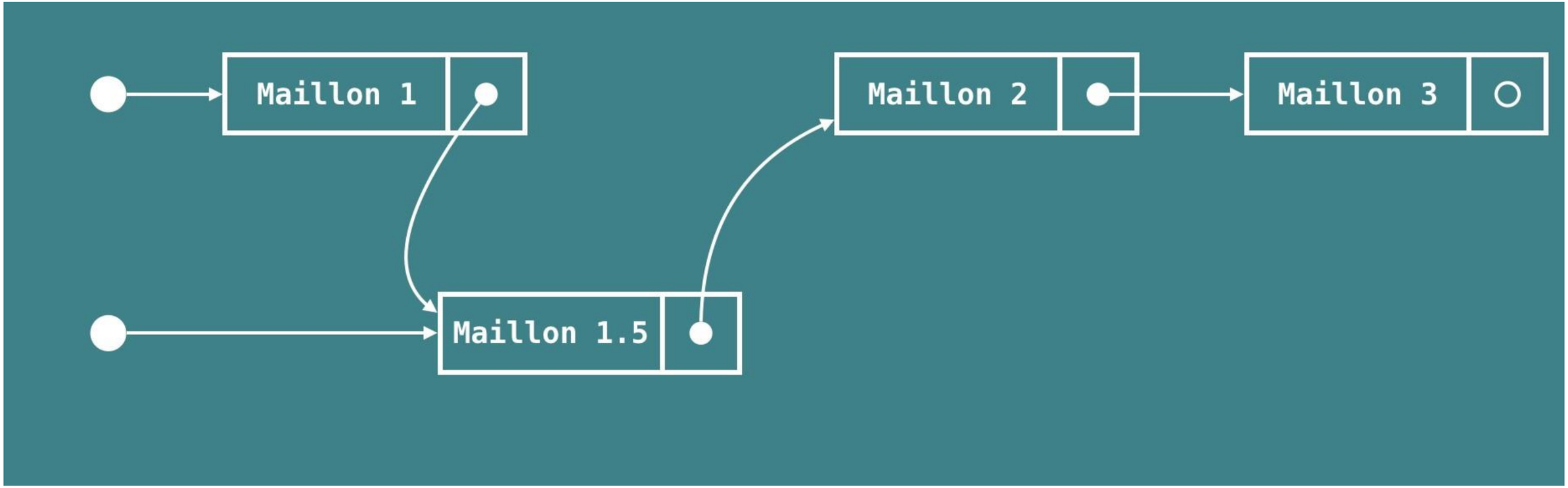
Insertion en milieu de liste



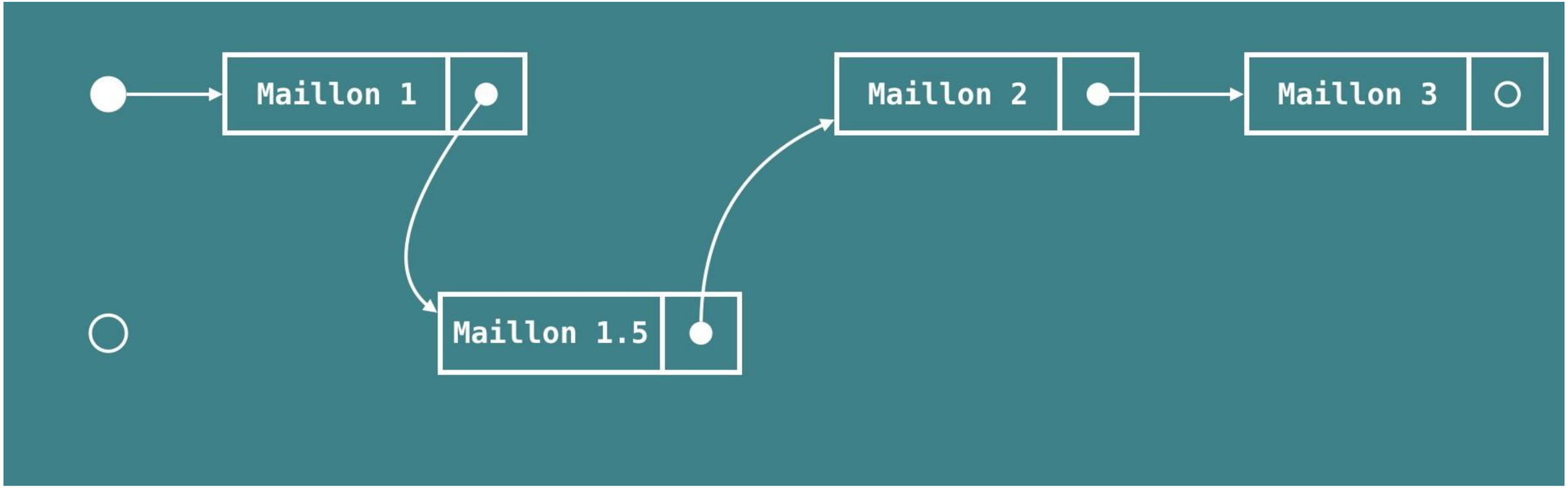
Insertion en milieu de liste



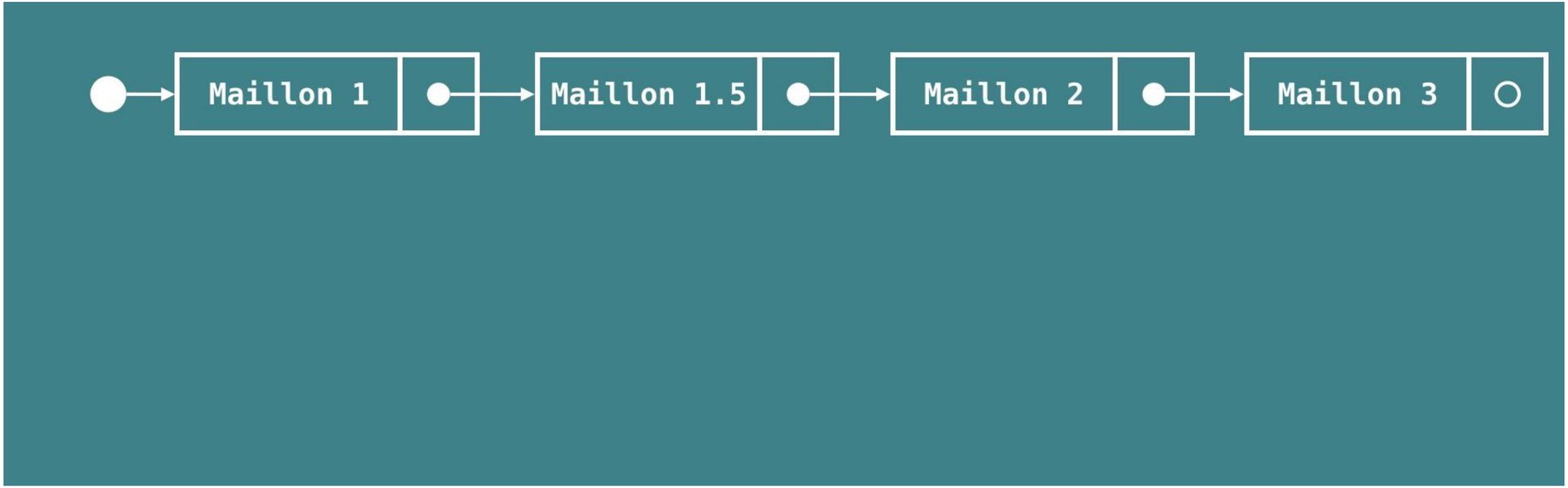
Insertion en milieu de liste



Insertion en milieu de liste



Insertion en milieu de liste



Astuce

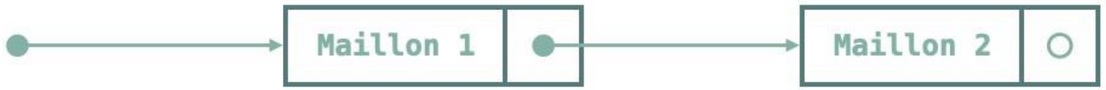
Dans une liste chaînée, pour éviter de parcourir à chaque fois tous les maillons pour accéder au dernier, on peut très bien créer un pointeur qui pointe toujours directement sur celui-ci.

Rien ne nous empêche d'avoir plusieurs pointeurs qui pointent sur différents maillons.

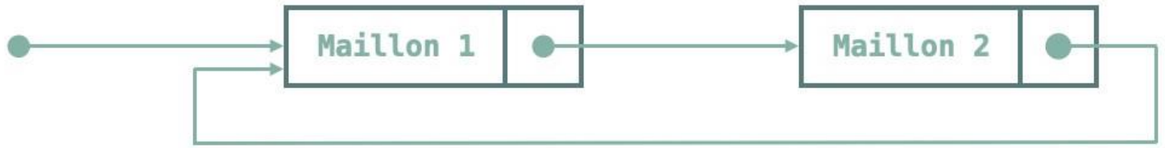
Pour aller plus loin : voir les skip lists.

Différents types de listes chaînées

Listes simplement chaînée



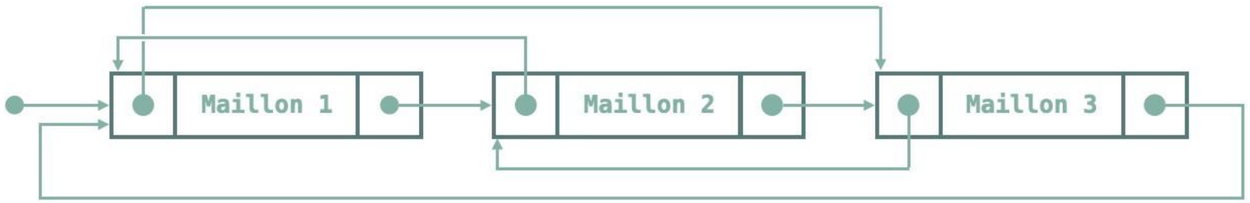
Listes simplement chaînée circulaire



Différents types de listes chaînées

Listes doublement chaînée

Listes doublement chaînée circulaire



Complexité

Suivant l'utilisation que l'on fera des données, il convient de choisir la structure de données la plus adaptée.

Par exemple, les tableaux ne sont pas du tout adaptés si l'on souhaite insérer régulièrement des éléments au début ou au milieu de ceux-ci. En effet, il faudra décaler toutes les cases se trouvant à droite de l'insertion. Il faudra aussi éventuellement agrandir le tableau si le nombre de cases disponibles est insuffisant.

À l'inverse, une liste chaînée est tout à fait adaptée à cet usage puisqu'il suffira de rajouter un maillon au début ou au milieu de la chaîne, sans avoir besoin de déplacer aucun maillon existant. Il suffira de modifier la valeur de quelques pointeurs.

Choix de la structure de données

Selon les tâches à effectuer sur une structure de données, le choix de celle-ci sera différent.

Scénario	Structure de données
Beaucoup d'insertions ou de suppressions, au début ou au milieu	Liste chaînée
Besoin d'accès aux éléments par indice	Tableau
etc.	

Ce qu'il faut retenir

Attention à :

- Faire le bon choix de structure de données (tableau, liste *(simplement/doublement)* chaînée *(circulaire)*...)
- Ne pas perdre de maillon (mauvaise manipulation des pointeurs) 🦮
- Libérer toute la mémoire
- Savoir si on s'arrête quand **`curseur == NULL`** ou quand **`curseur->next == NULL`**

Solution :

-  Faire des schémas !!