

Structures Linéaires





6. Listes simplement chaînées

Motivations

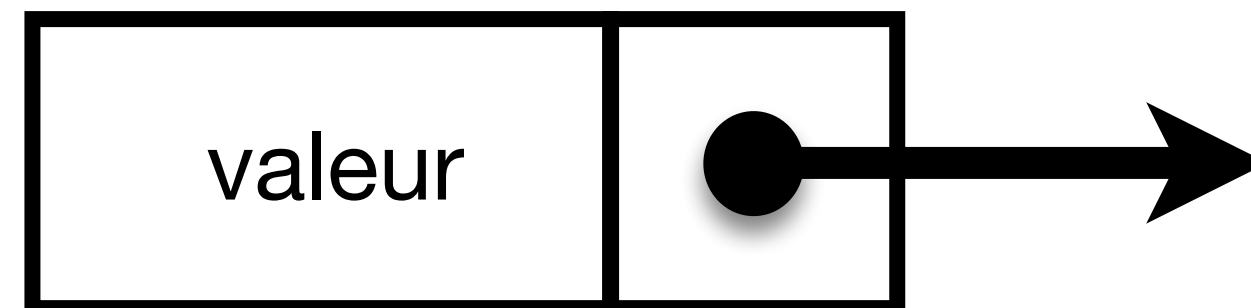


- **Tableaux** : insertion/suppression inefficace en position arbitraire.
 - *Obligé de déplacer tous les éléments pour faire de la place.*
- **Proposition** : ne plus stocker dans des emplacements mémoire consécutifs
- **Problème** : on perd la relation implicite suivant/précédent
- **Solution** : nécessité de représenter cette relation explicitement
 - Chaque élément est stocké dans une structure appelée *maillon* ...
 - ... qui inclut un pointeur vers le maillon suivant (et précédent)
 - L'ensemble de ces maillons constitue une liste chaînée

Un maillon simple



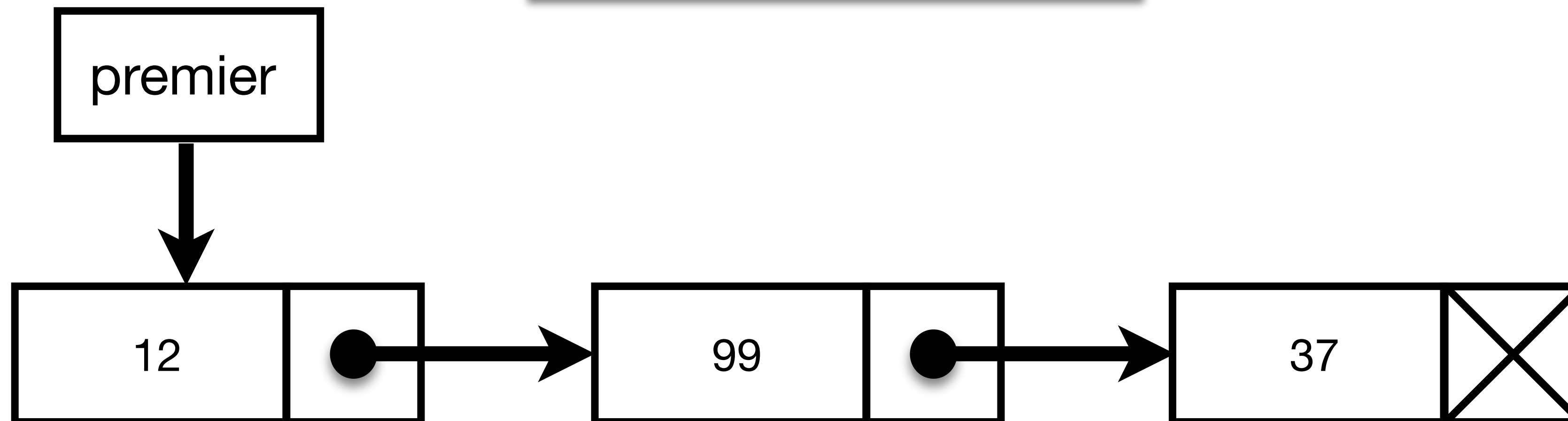
```
structure Maillon
  Element valeur
  Maillon* suivant
```



Liste simplement chaînée



```
structure Maillon
  Element valeur
  Maillon* suivant
```



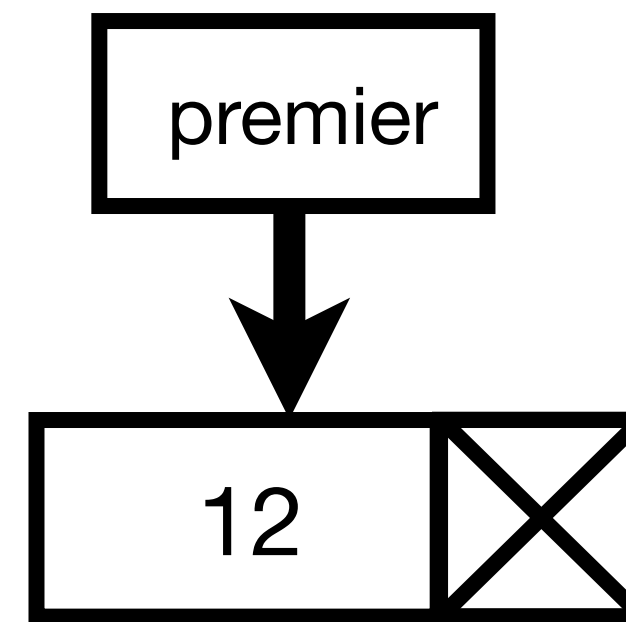
Construction



Par la fin

*(nouveaux maillons
ajoutés à la fin)*

```
premier ← nouveau Maillon(12, ∅)
```



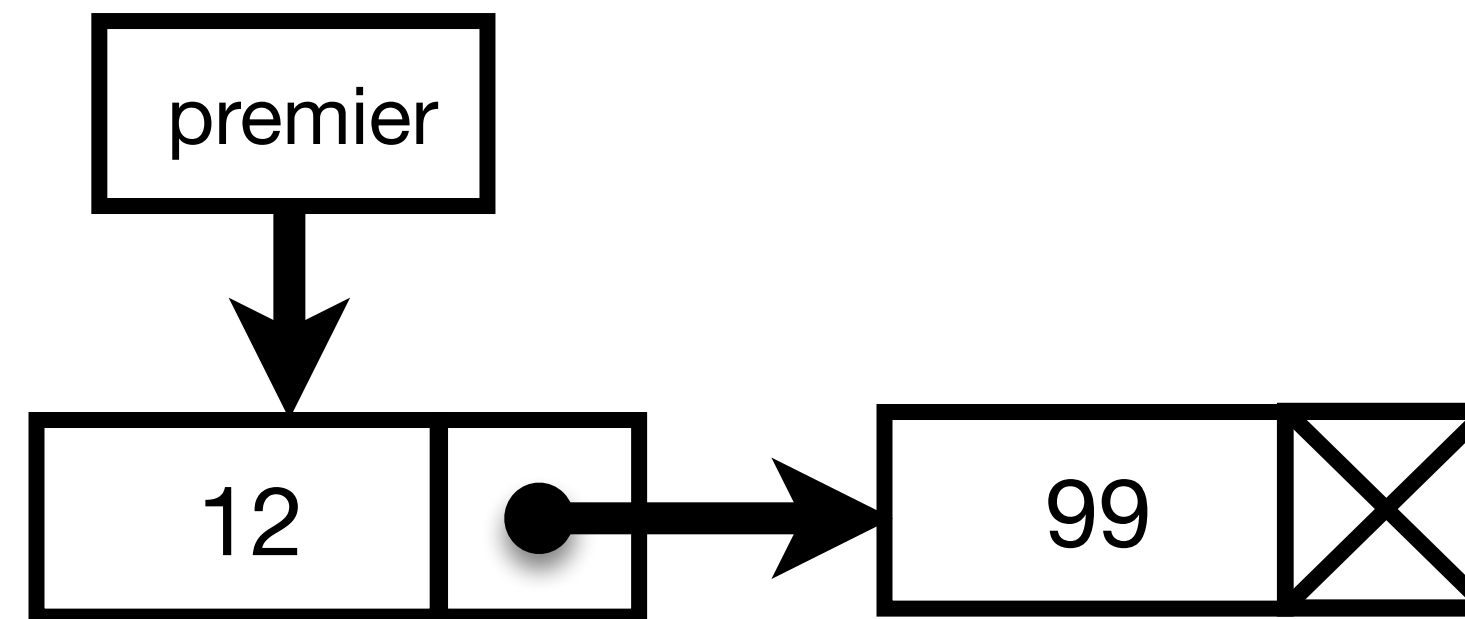
Construction



Par la fin

*(nouveaux maillons
ajoutés à la fin)*

```
premier ← nouveau Maillon(12, ∅)
premier.suivant ← nouveau Maillon(99, ∅)
```



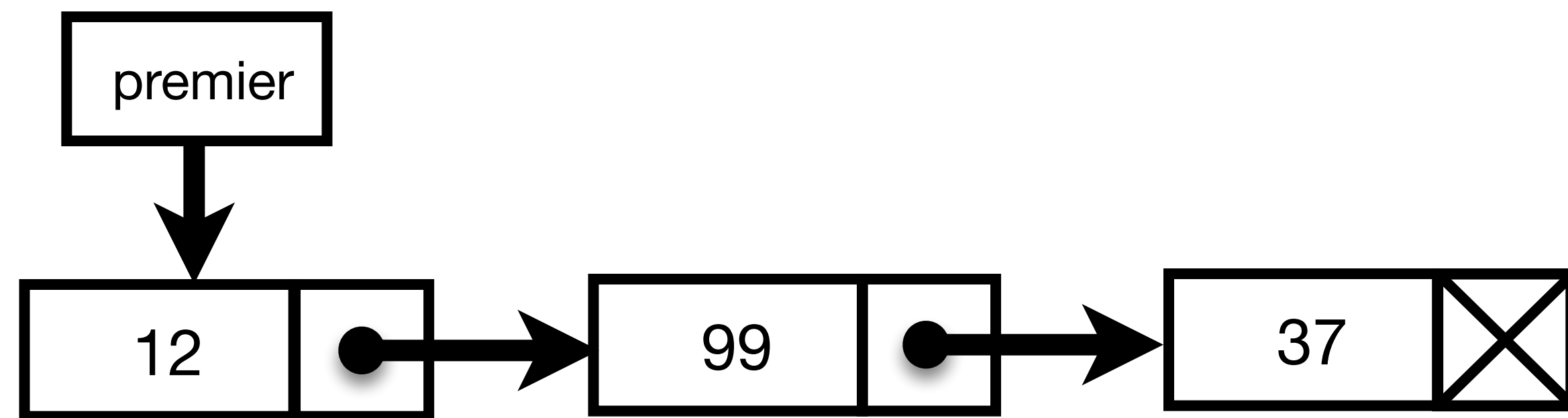
Construction



Par la fin

*(nouveaux maillons
ajoutés à la fin)*

```
premier ← nouveau Maillon(12, ∅)  
premier.suivant ← nouveau Maillon(99, ∅)  
premier.suivant.suivant ← nouveau Maillon(37, ∅)
```



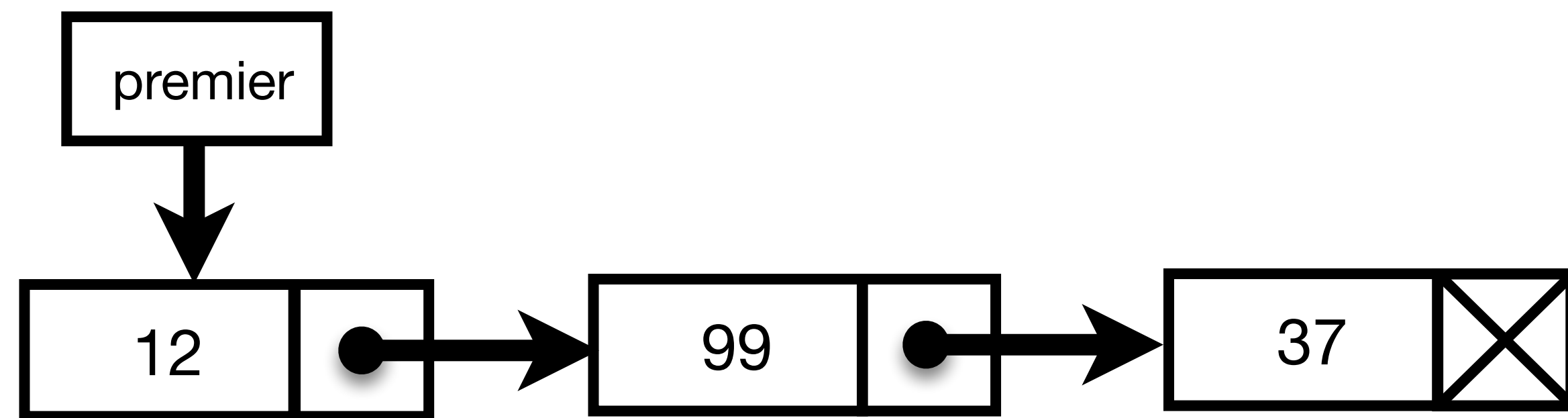
Construction



Par la fin

(nouveaux maillons ajoutés à la fin)

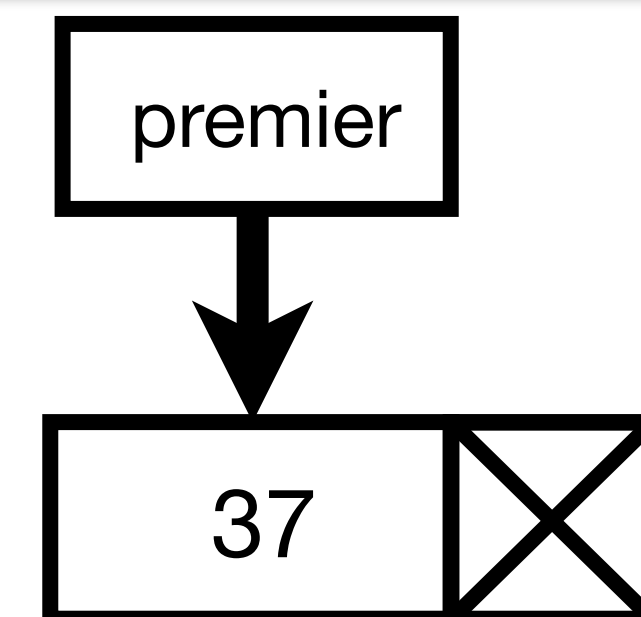
```
premier ← nouveau Maillon(12, ∅)
premier.suivant ← nouveau Maillon(99, ∅)
premier.suivant.suivant ← nouveau Maillon(37, ∅)
```



Par le début

(nouveaux maillons ajoutés au début)

```
premier ← nouveau Maillon(37, ∅)
```



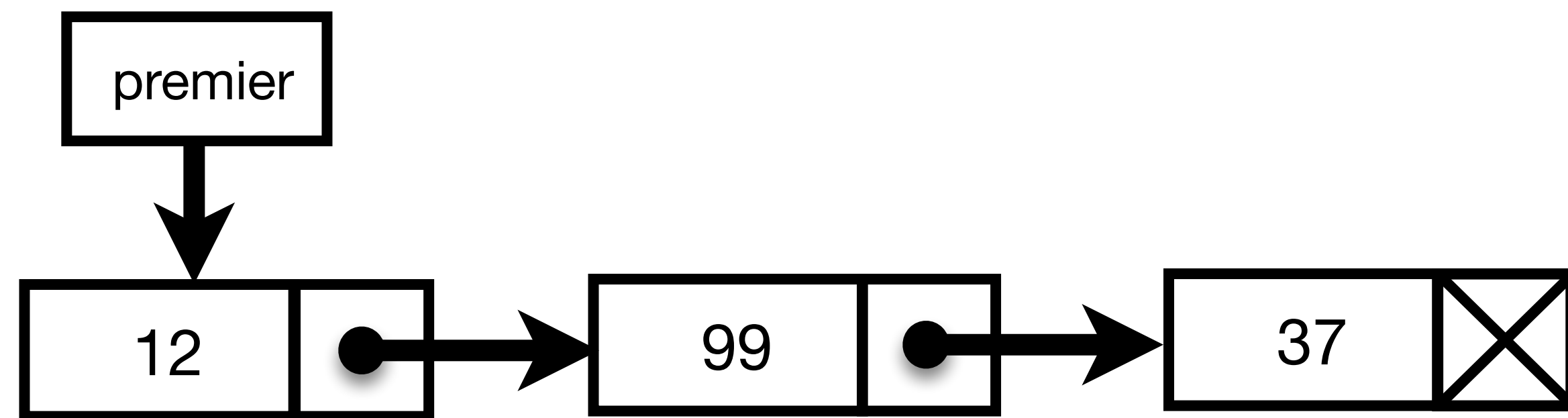
Construction



Par la fin

(nouveaux maillons ajoutés à la fin)

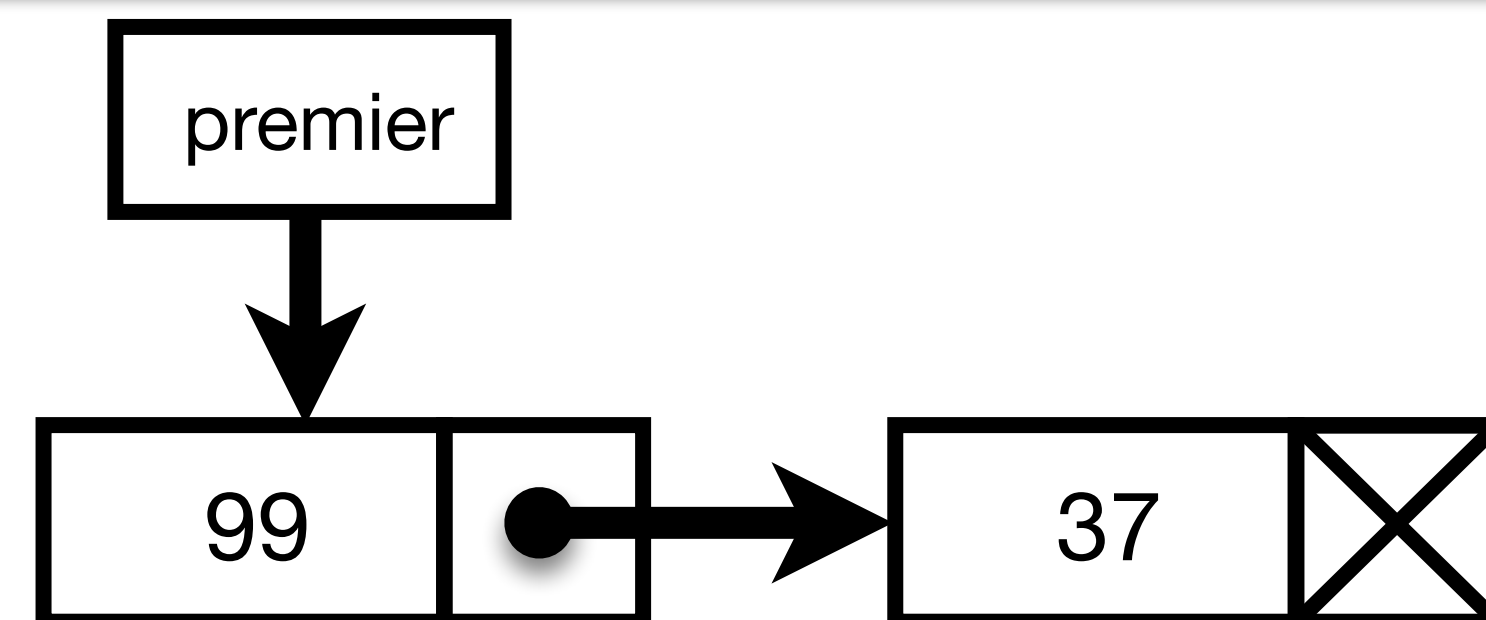
```
premier ← nouveau Maillon(12, ∅)
premier.suivant ← nouveau Maillon(99, ∅)
premier.suivant.suivant ← nouveau Maillon(37, ∅)
```



Par le début

(nouveaux maillons ajoutés au début)

```
premier ← nouveau Maillon(37, ∅)
premier ← nouveau Maillon(99, premier)
```



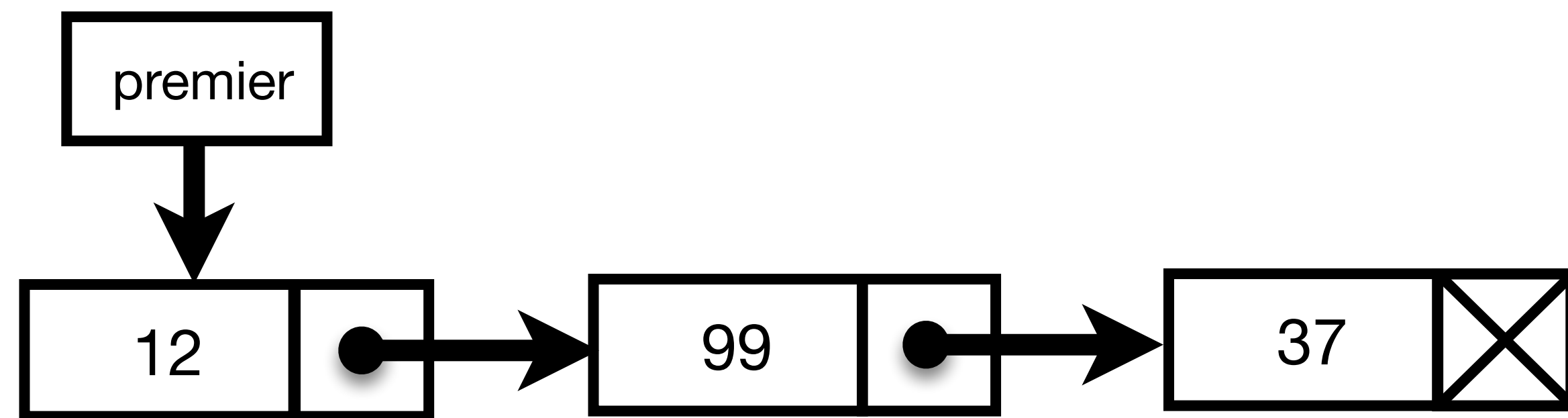
Construction



Par la fin

(nouveaux maillons ajoutés à la fin)

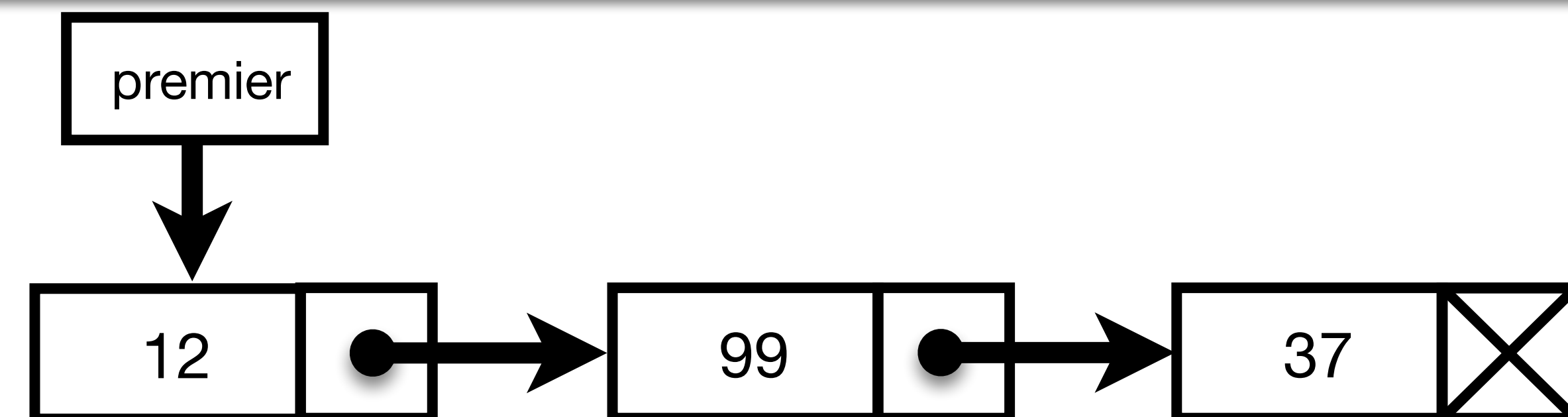
```
premier ← nouveau Maillon(12, ∅)
premier.suivant ← nouveau Maillon(99, ∅)
premier.suivant.suivant ← nouveau Maillon(37, ∅)
```



Par le début

(nouveaux maillons ajoutés au début)

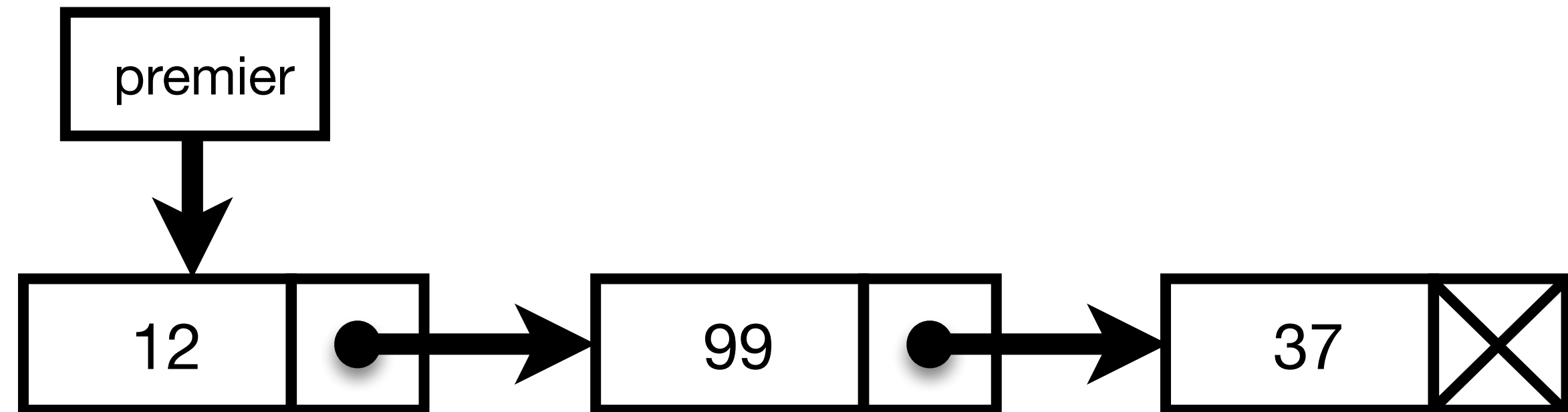
```
premier ← nouveau Maillon(37, ∅)
premier ← nouveau Maillon(99, premier)
premier ← nouveau Maillon(12, premier)
```





```
template < typename T >
struct Maillon {
    T valeur;
    Maillon* suivant;
};
```

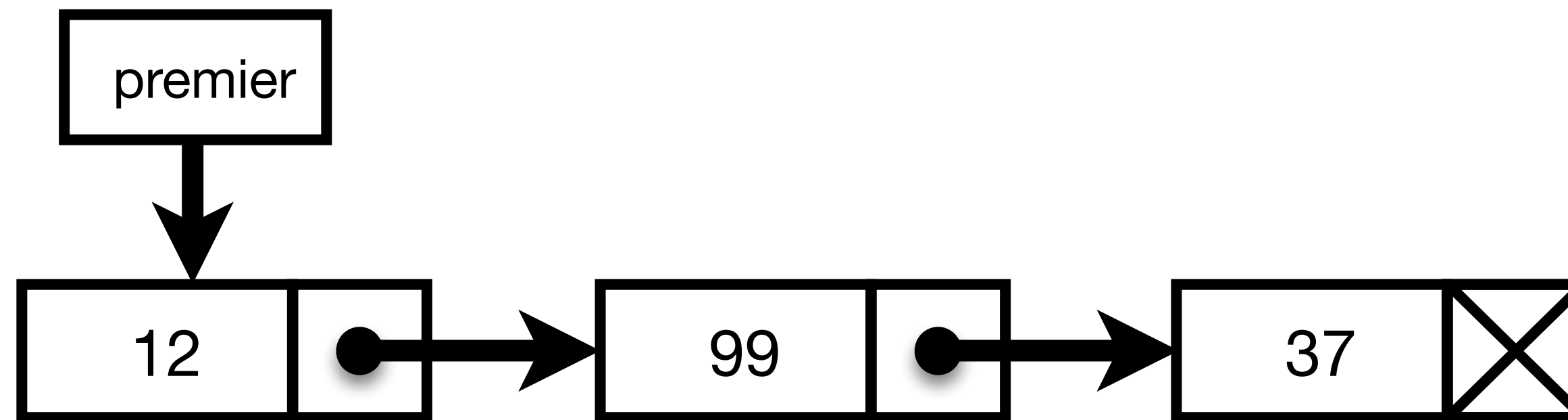
- Construction par la fin
- Construction par le début



```
Maillon<int>* premier = new Maillon<int> {12, nullptr};
premier->suivant      = new Maillon<int> {99, nullptr};
premier->suivant->suivant = new Maillon<int> {37, nullptr};
```

```
Maillon<int>* premier = new Maillon<int> {37, nullptr};
premier              = new Maillon<int> {99, premier};
premier              = new Maillon<int> {12, premier};
```

Parcourir



12→99→37→∅

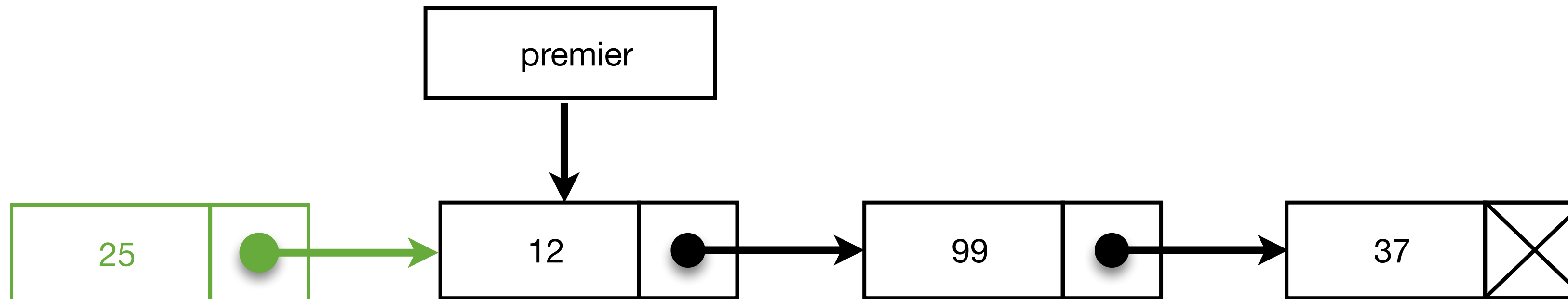
- Itérativement

```
fonction afficher(premier)
  courant ← premier
  tant que courant != ∅
    Afficher courant.valeur + "→"
    courant ← courant.suivant
  Afficher "∅"
```

- Récursivement

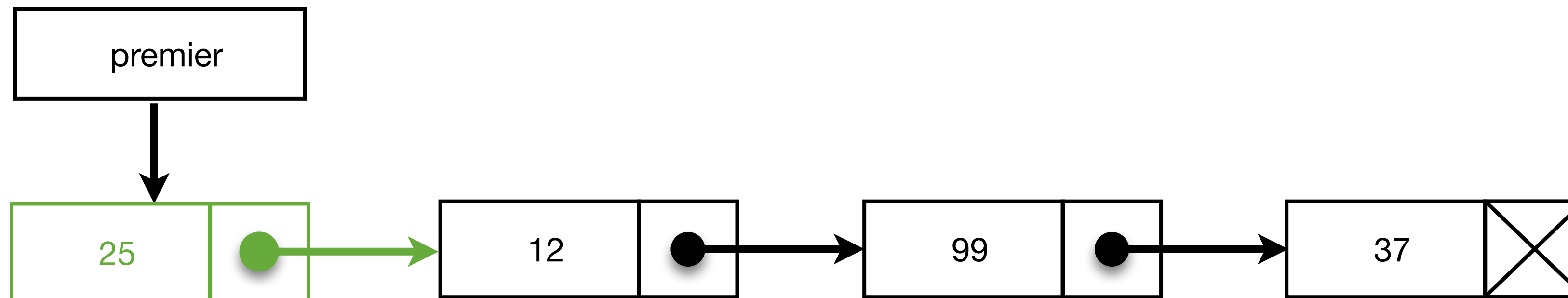
```
fonction afficher(premier):
  si premier != ∅
    Afficher premier.valeur + "→"
    afficher(premier.suivant)
  sinon
    Afficher "∅"
```

Insérer au début



```
fonction insérer_au_début(premier, valeur)
  tmp ← nouveau Maillon
  tmp.valeur ← valeur
  tmp.suivant ← premier
  retourner tmp
```

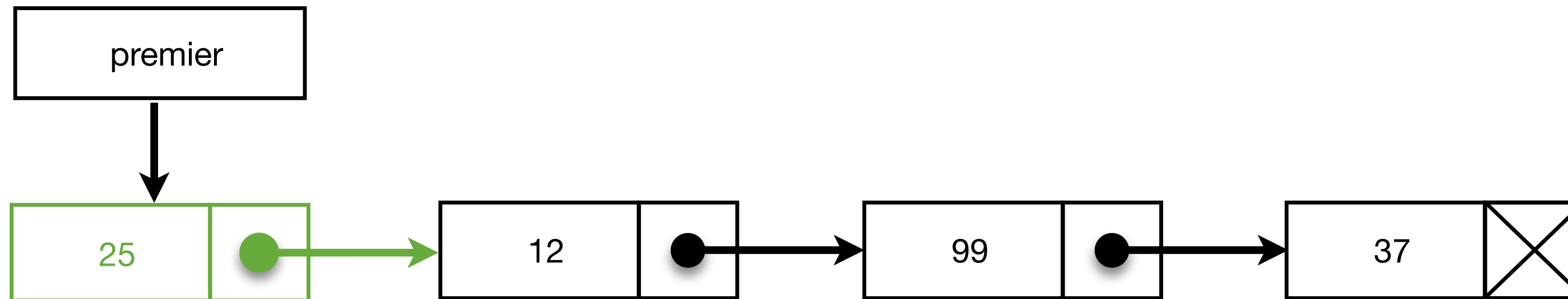
Insérer au début



```
fonction insérer_au_début(premier, valeur)
    tmp ← nouveau Maillon
    tmp.valeur ← valeur
    tmp.suivant ← premier
    retourner tmp
```

```
premier ← insérer_au_début(premier, valeur)
```

Insérer au début



ou

```

fonction insérer_au_début(premier, valeur)
  tmp ← nouveau Maillon
  tmp.valeur ← valeur
  tmp.suivant ← premier
  retourner tmp
  
```

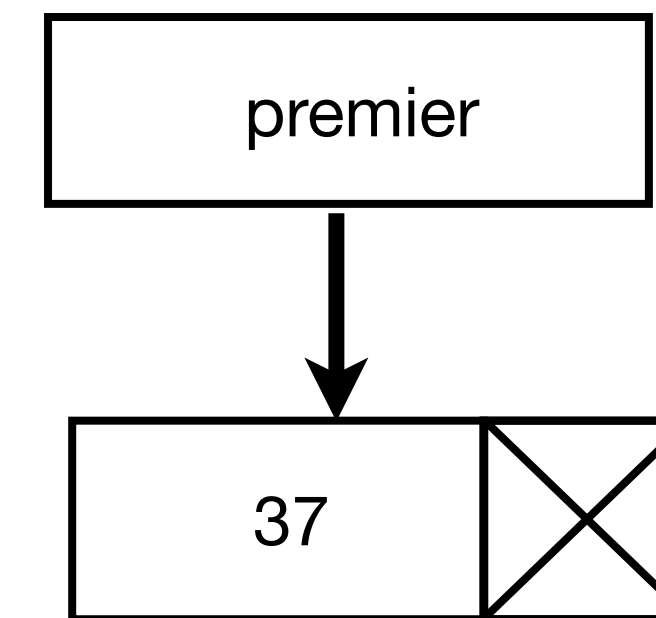
```

fonction insérer_au_début(premier, valeur)
  retourner nouveau Maillon(valeur, premier)
  
```

```

premier ← insérer_au_début(premier, valeur)
  
```

Insérer au début



ou

```

fonction insérer_au_début(premier, valeur)
  tmp ← nouveau Maillon
  tmp.valeur ← valeur
  tmp.suivant ← premier
  retourner tmp
  
```

```

fonction insérer_au_début(premier, valeur)
  retourner nouveau Maillon(valeur, premier)
  
```

```

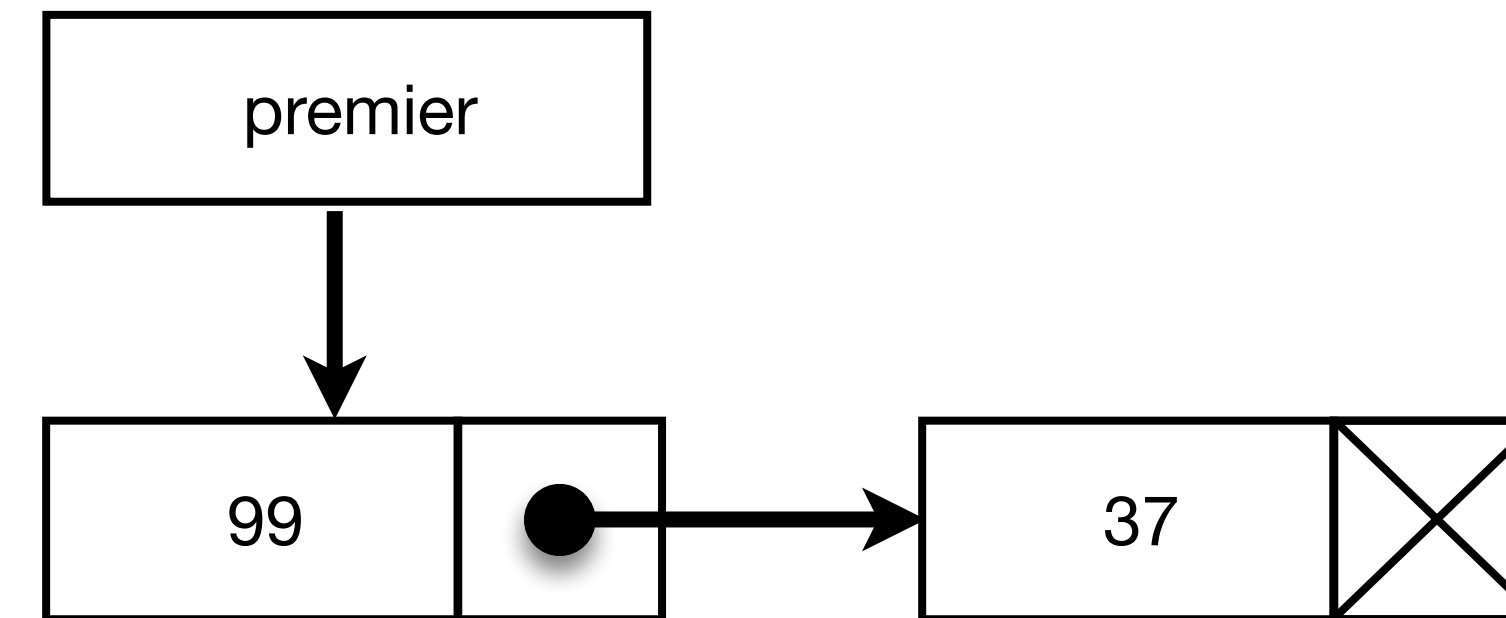
premier ← insérer_au_début(premier, valeur)
  
```

Construire par insertion au début

```

premier ← ∅
premier ← insérer_au_début(premier, 37)
  
```

Insérer au début



ou

```
fonction insérer_au_début(premier, valeur)
  tmp ← nouveau Maillon
  tmp.valeur ← valeur
  tmp.suivant ← premier
  retourner tmp
```

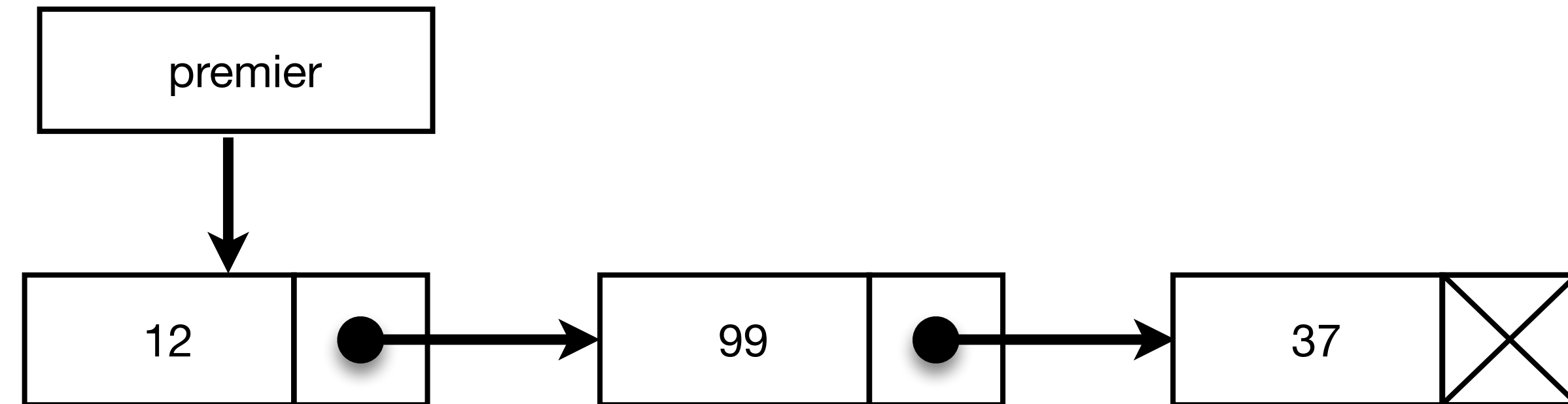
```
fonction insérer_au_début(premier, valeur)
  retourner nouveau Maillon(valeur, premier)
```

```
premier ← insérer_au_début(premier, valeur)
```

Construire par insertion au début

```
premier ← ∅
premier ← insérer_au_début(premier, 37)
premier ← insérer_au_début(premier, 99)
```

Insérer au début



ou

```
fonction insérer_au_début(premier, valeur)
    tmp ← nouveau Maillon
    tmp.valeur ← valeur
    tmp.suivant ← premier
    retourner tmp
```

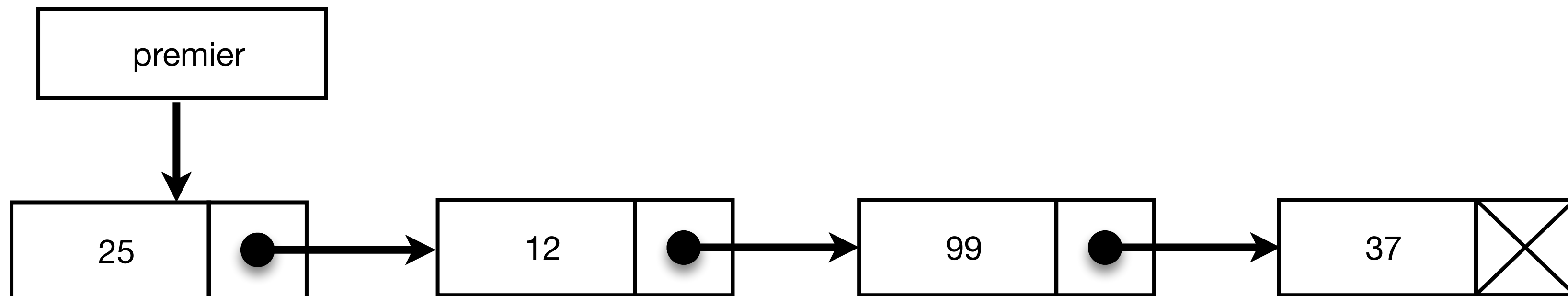
```
fonction insérer_au_début(premier, valeur)
    retourner nouveau Maillon(valeur, premier)
```

```
premier ← insérer_au_début(premier, valeur)
```

Construire par insertion au début

```
premier ← ∅
premier ← insérer_au_début(premier, 37)
premier ← insérer_au_début(premier, 99)
premier ← insérer_au_début(premier, 12)
```

Insérer au début



ou

```

fonction insérer_au_début(premier, valeur)
    tmp ← nouveau Maillon
    tmp.valeur ← valeur
    tmp.suivant ← premier
    retourner tmp
    
```

```

fonction insérer_au_début(premier, valeur)
    retourner nouveau Maillon(valeur, premier)
    
```

```

premier ← insérer_au_début(premier, valeur)
    
```

Construire par insertion au début

```

premier ← ∅
premier ← insérer_au_début(premier, 37)
premier ← insérer_au_début(premier, 99)
premier ← insérer_au_début(premier, 12)
premier ← insérer_au_début(premier, 25)
    
```

En C++, passer les pointeurs...



- ... par valeur

```
template <typename T>
Maillon<T>* inserer1 (Maillon<T>* p, const T& val) {
    return new Maillon<T> { val, p };
}
premier = inserer1(premier, 25);
```

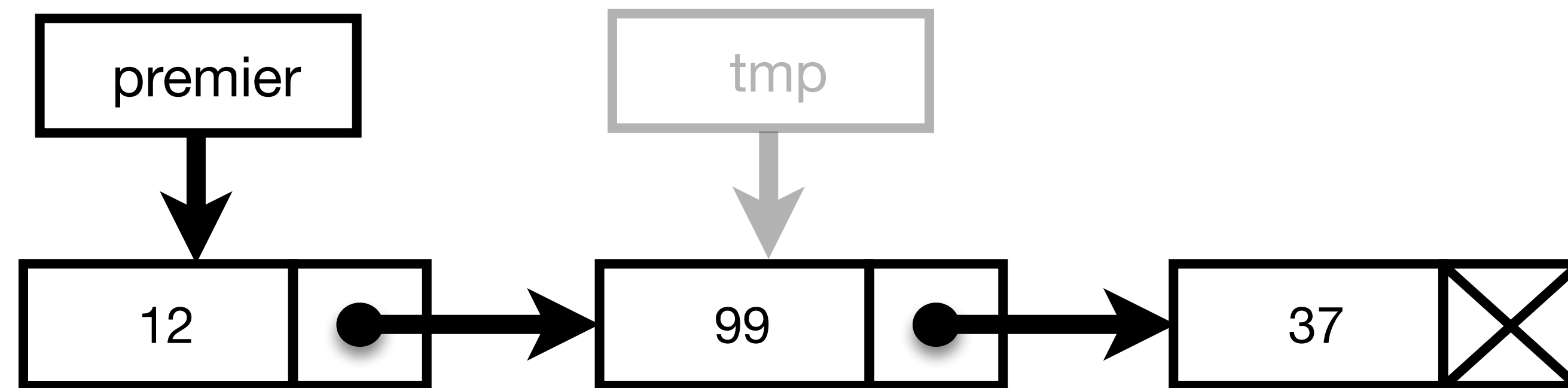
- ... par référence

```
template <typename T>
void inserer2 (Maillon<T>*& p, const T& val) {
    p = new Maillon<T> { val, p };
}
inserer2(premier, 25);
```

- ... par pointeur de pointeur

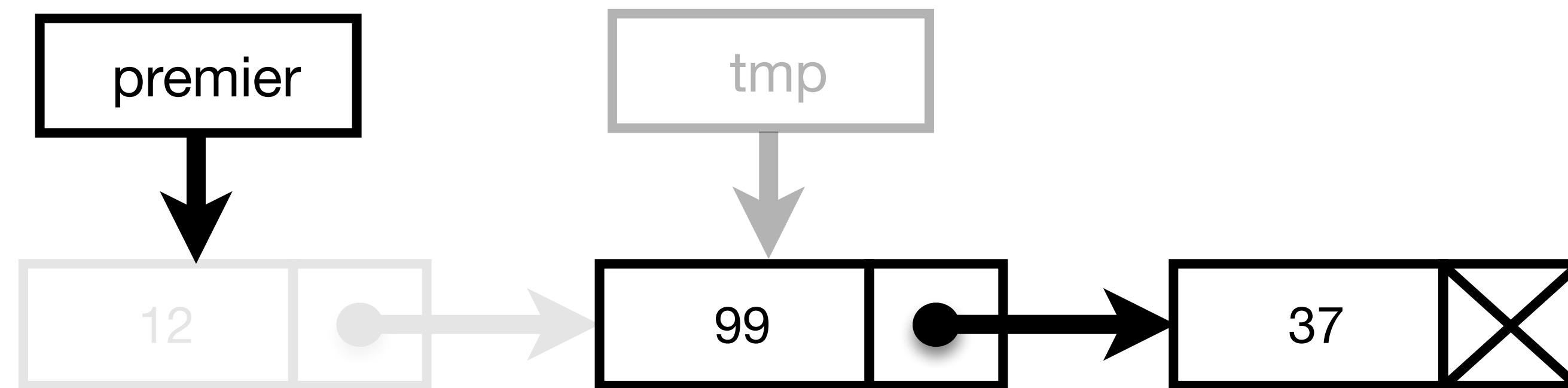
```
template <typename T>
void inserer3 (Maillon<T>** ptr_p, const T& val) {
    *ptr_p = new Maillon<T> { val, *ptr_p };
}
inserer3(&premier, 25);
```

Supprimer au début



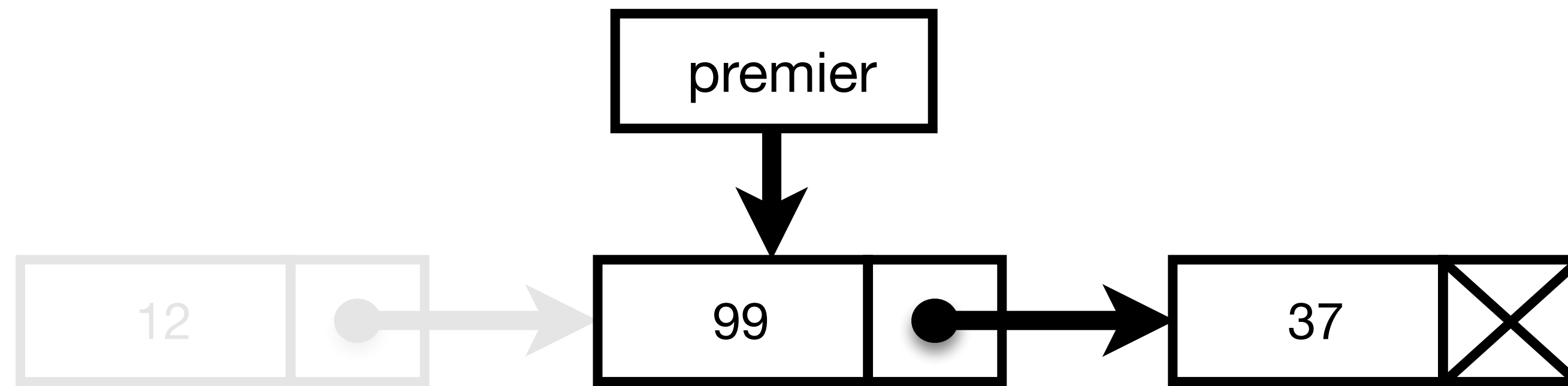
```
fonction supprimer_au_début(premier)
    alerter si premier == ∅
    tmp ← premier.suivant
```

Supprimer au début



```
fonction supprimer_au_début(premier)
  alerter si premier == ∅
  tmp ← premier.suivant
  effacer premier
  retourner tmp
```

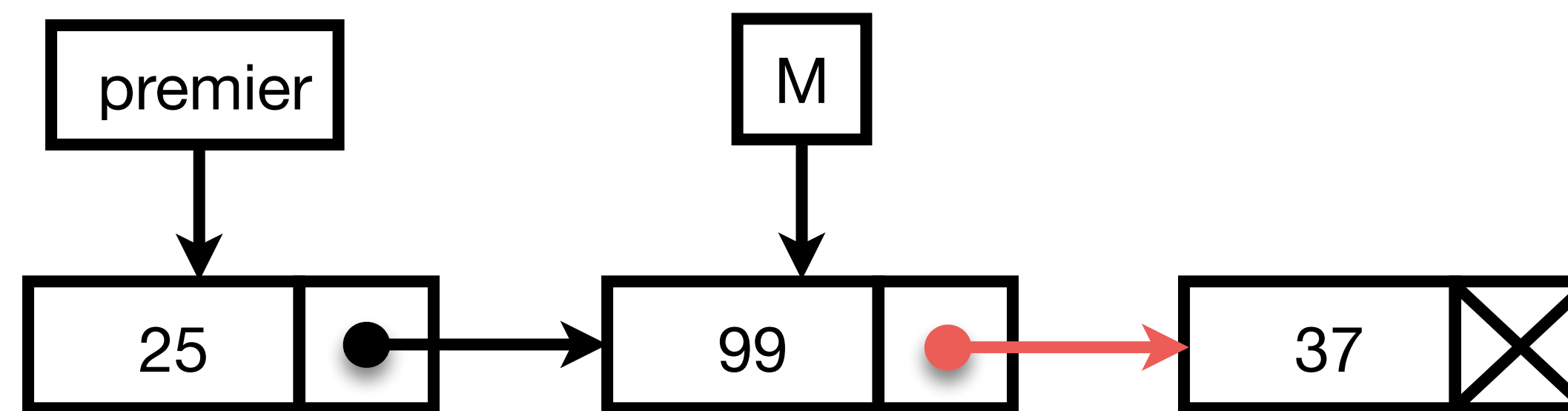
Supprimer au début



```
fonction supprimer_au_début(premier)  
  alerter si premier == ∅  
  tmp ← premier.suivant  
  effacer premier  
  retourner tmp
```

```
premier ← supprimer_au_début(premier)
```

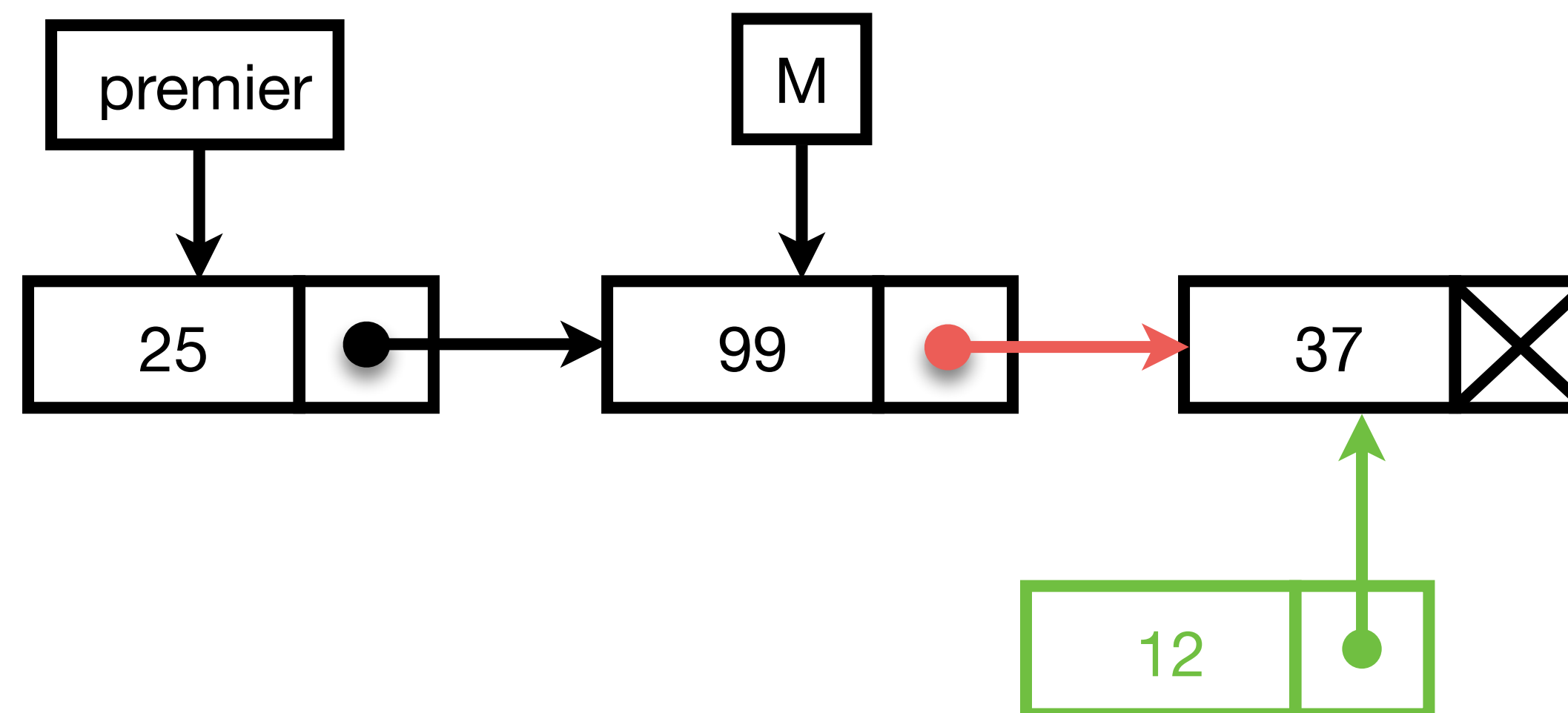
Insérer après le maillon M



```

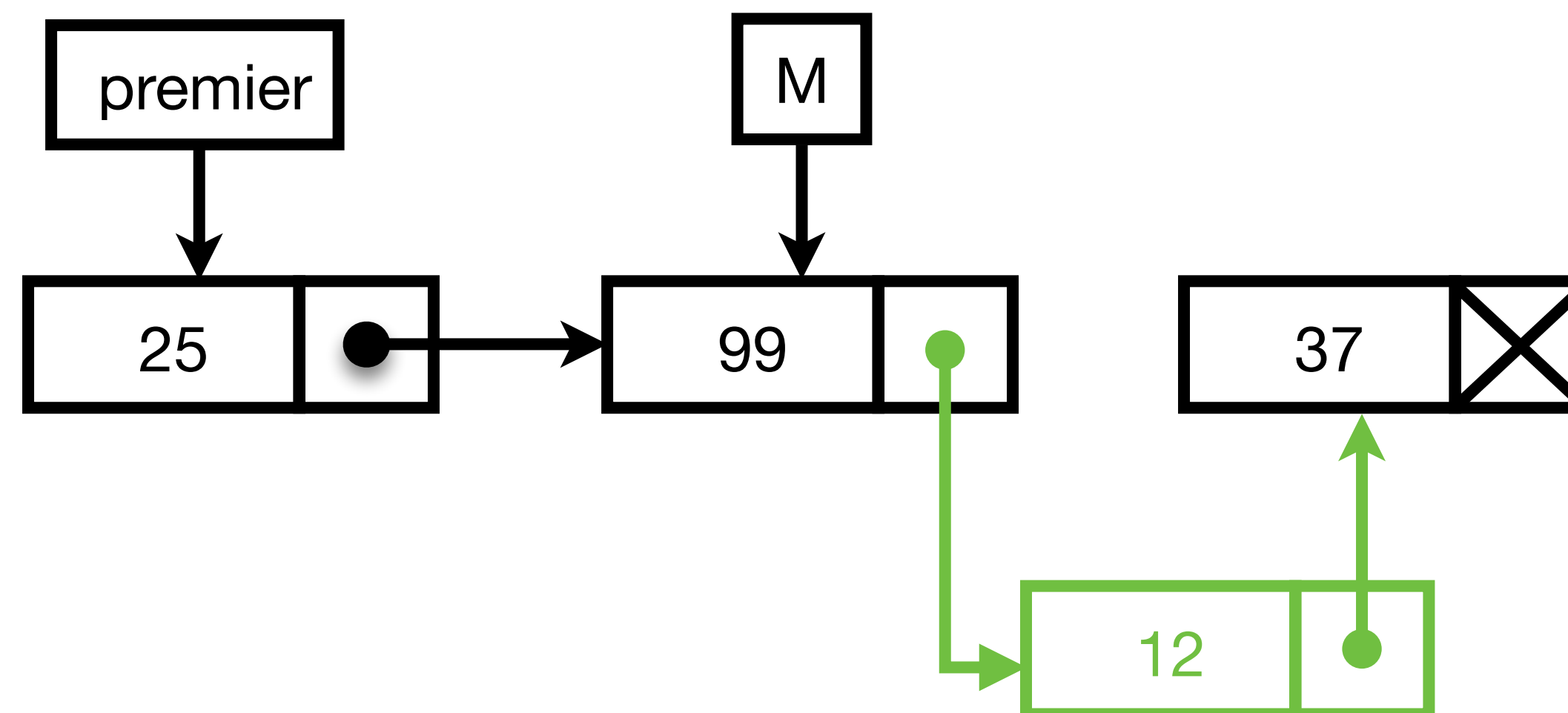
fonction insérer_après(M, val)
  alerter si M == ∅
  M.suivant ← nouveau Maillon(val, M.suivant)
  
```

Insérer après le maillon M



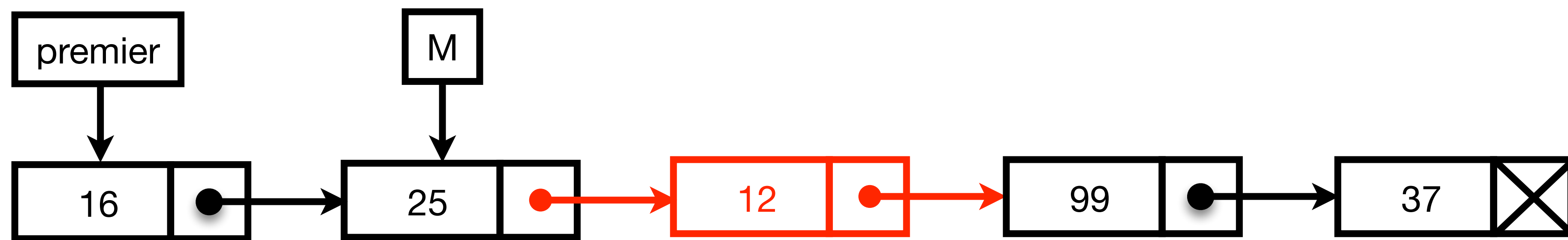
```
fonction insérer_après(M, val)  
    alerter si M == ∅  
    M.suivant ← nouveau Maillon(val, M.suivant)
```

Insérer après le maillon M



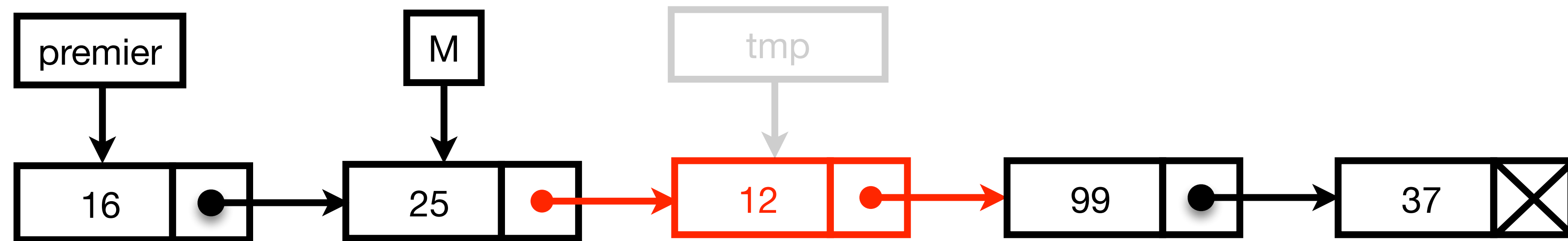
```
fonction insérer_après(M, val)
    alerter si M == ∅
    M.suivant ← nouveau Maillon(val, M.suivant)
```

Supprimer après le maillon M



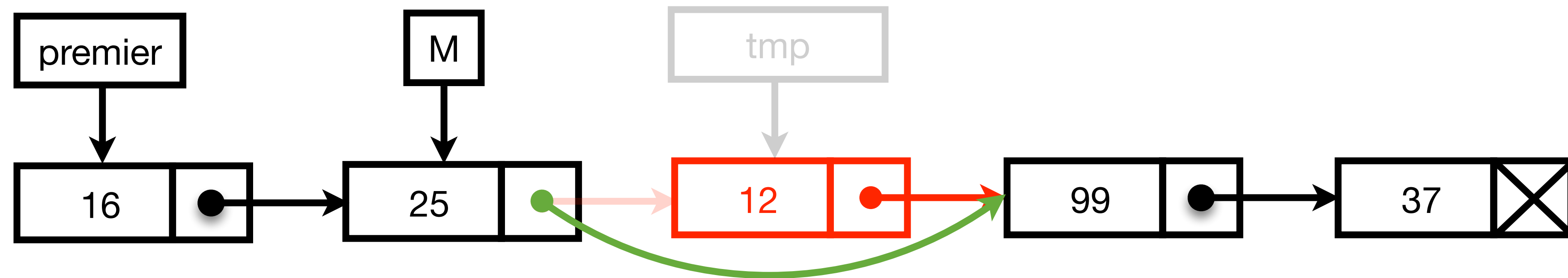
```
fonction supprimer_après(M)
  alerter si M == ∅
  alerter si M.suivant == ∅
```

Supprimer après le maillon M



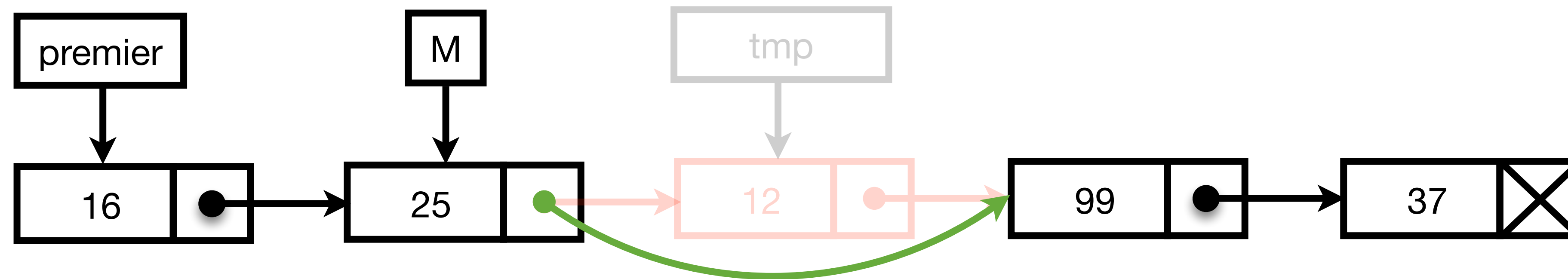
```
fonction supprimer_après(M)
  alerter si M == ∅
  alerter si M.suivant == ∅
  tmp ← M.suivant
```

Supprimer après le maillon M



```
fonction supprimer_après(M)
    alerter si M == ∅
    alerter si M.suivant == ∅
    tmp ← M.suivant
    M.suivant ← tmp.suivant
```

Supprimer après le maillon M



```
fonction supprimer_après (M)
  alerter si M == ∅
  alerter si M.suivant == ∅
  tmp ← M.suivant
  M.suivant ← tmp.suivant
  effacer tmp
```

Exemple: ôter les répétitions



In:

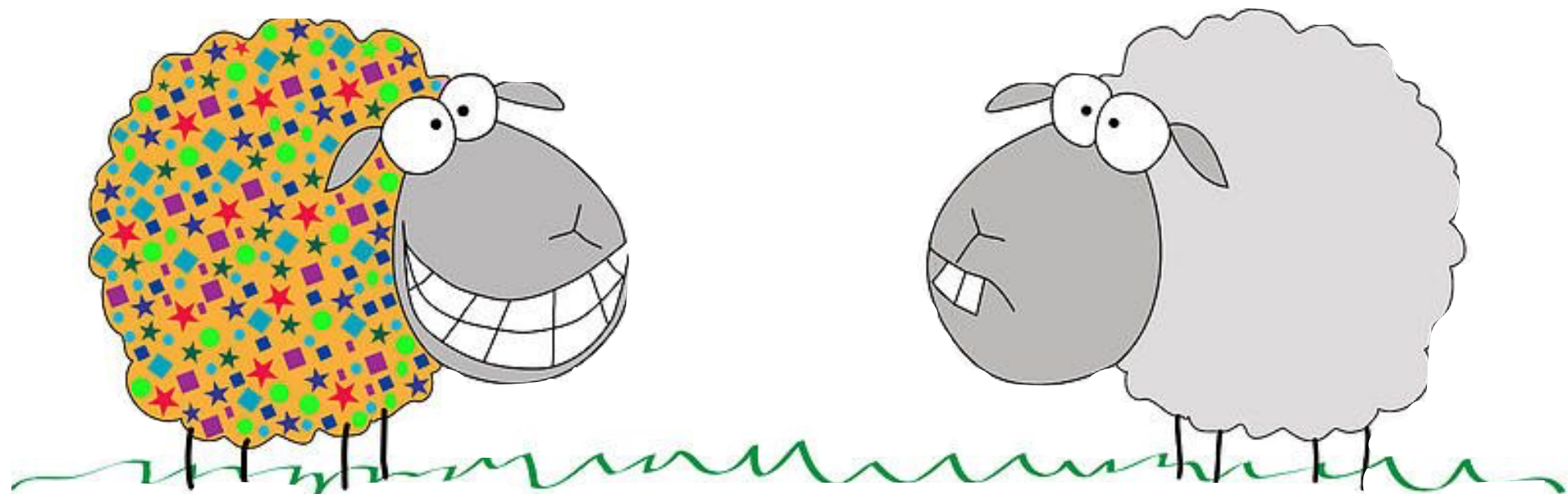
1 → 1 → 1 → 0 → 0 → 0 → 2 → 2 → 1 → 1 → 0 → ∅

Out:

1 → 0 → 2 → 1 → 0 → ∅

BE UNIQUE

```
fonction ôter_répétitions(premier)
    m ← premier
    tant que m != ∅ et m.suivant != ∅
        si m.valeur == m.suivant.valeur
            supprimer_après(m)
        sinon
            m ← m.suivant
```



Encapsulation



- Classe ListeSimple
- Stocke et cache le pointeur vers le premier maillon
- Met à jour ce pointeur dans `push_front()` et `pop_front()`

```
classe ListeSimple  
  
    attribut: premier ← ∅  
  
    méthode push_front(v)  
        premier ← insérer_au_début(premier, v)  
  
    méthode pop_front()  
        premier ← supprimer_au_début(premier)  
  
    méthode insert_after(m, v)  
        insérer_après(m, v)  
  
    méthode erase_after(m)  
        supprimer_après(m)  
  
    méthode empty()  
        retourne premier == ∅
```



7. Itération sur liste simple



Pour itérer, on traverse chaque maillon.

Cela soulève les questions suivantes :

- Dans la classe encapsulée, l'attribut `premier` doit-il être public ou privé ?
- La notion de `Maillon` doit-elle être publique ou privée ?

```
classe ListeSimple
    attribut: premier ← ∅

    méthode push_front(v)
        premier ← insérer_au_début(premier, v)

    méthode pop_front()
        premier ← supprimer_au_début(premier)

    méthode insert_after(m, v)
        insérer_après(m, v)

    méthode erase_after(m)
        supprimer_après(m)

    méthode empty()
        retourne premier == ∅
```

1ère étape



`Maillon`, `premier`, `suivant` et `valeur` sont des **détails de mise en oeuvre**, ils ne doivent pas être publiques.

Commençons par :

- Cacher le fonctionnement interne de la structure `Maillon`
- Éviter l'accès à `suivant` ou `valeur` par du code externe
- Éviter l'accès à `premier` par du code externe

Itération avec des fonctions



```
fonction suivant(m)  
    retourner m.suivant
```

```
fonction lire(m)  
    retourner m.valeur
```

```
fonction écrire(m, v)  
    m.valeur = v
```

```
fonction début(L)  
    retourner L.premier
```

```
fonction fin(L)  
    retourner ∅
```

```
classe ListeSimple  
  
    attributs: premier ← ∅  
  
    méthode push_front(v)  
        premier ← insérer_au_début(premier, v)  
  
    méthode pop_front()  
        premier ← supprimer_au_début(premier)  
  
    méthode insert_after(m, v)  
        insérer_après(m, v)  
  
    méthode erase_after(m)  
        supprimer_après(m)  
  
    méthode empty()  
        retourne premier == ∅
```

Exemple: ôter les répétitions



```
fonction ôter_répétitions(premier : Maillon*)  
  m ← premier  
  tant que m != ∅  
    si m.suivant != ∅ et m.valeur == m.suivant.valeur  
      supprimer_après(m)  
    sinon  
      m ← m.suivant
```

```
fonction ôter_répétitions(L : ListeSimple)  
  m ← début(L)  
  tant que m != fin(L)  
    n ← suivant(m)  
    si n != fin(L) et lire(m) == lire(n)  
      L.erase_after(m)  
    sinon  
      m ← n
```

Ex. 2: supprimer une valeur



Exemple: `supprimer_valeurs(L, 0)`

In: `2 → 1 → 0 → 3 → 2 → 1 → 0 → ∅`

Out: `2 → 1 → 3 → 2 → 1 → ∅`

```
fonction supprimer_valeurs(L, v)
  m ← début(L)
  tant que m != fin(L)
    n ← suivant(m)
    si n != fin(L) et lire(n) == v
      L.erase_after(m)
    sinon m ← n
```

Ex. 2: supprimer une valeur



Exemple: `supprimer_valeurs(L, 0)`

In: `2 → 1 → 0 → 3 → 2 → 1 → 0 → ∅`

Out: `2 → 1 → 3 → 2 → 1 → ∅`

Exemple: `supprimer_valeurs(L, 2)`

In: `2 → 1 → 3 → 2 → 1 → ∅`

Out: `2 → 1 → 3 → 1 → ∅`

```
fonction supprimer_valeurs(L, v)
  m ← début(L)
  tant que m != fin(L)
    n ← suivant(m)
    si n != fin(L) et lire(n) == v
      L.erase_after(m)
    sinon m ← n
```

`erase_after(m)` ne peut
pas supprimer le premier
maillon...

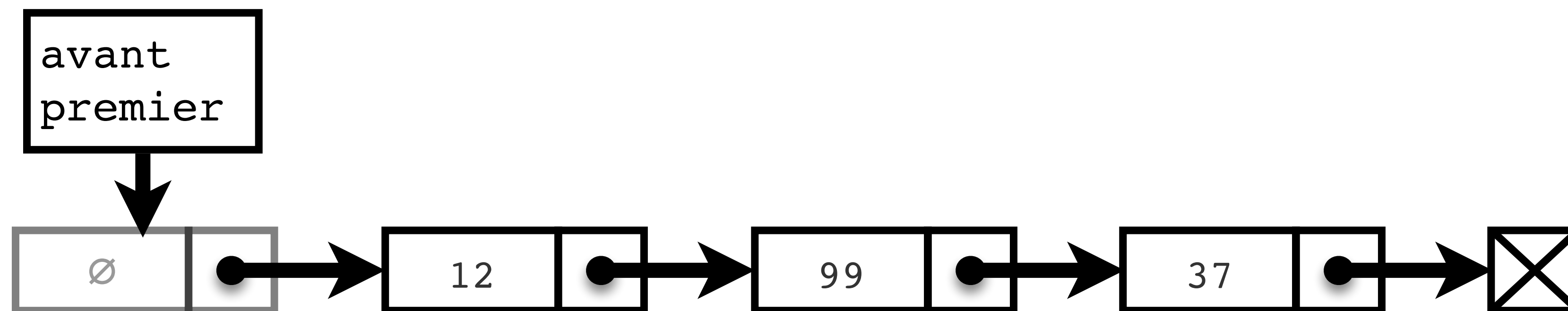
Solution ?

Ré-écrivons ListeSimple...



- Ajoutons un maillon vide devant la liste
- premier devient avant_premier.suivant

```
structure MaillonVide  
    Maillon* suivant
```



Ré-écrivons ListeSimple...



- L'attribut privé est `avant_premier` et plus `premier`
- `push_front` utilise `insert_after`
- `pop_front` utilise `erase_after`

```
classe ListeSimple

    attributs: avant_premier ← MaillonVide(∅)

    méthode insert_after(m, v)
        insérer_après(m, v)

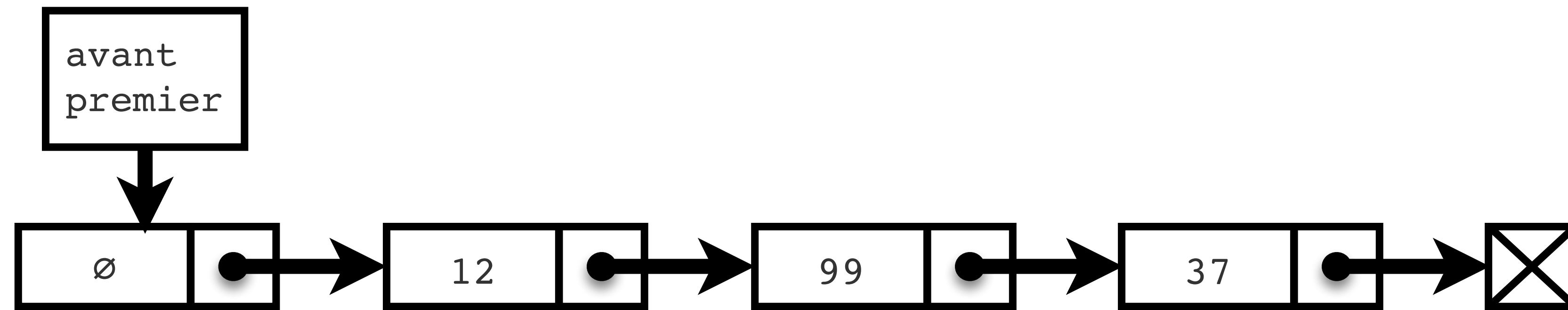
    méthode erase_after(m)
        supprimer_après(m)

    méthode push_front(v)
        insert_after(avant_premier, v)

    méthode pop_front()
        erase_after(avant_premier)

    méthode empty()
        retourner avant_premier.suivant == ∅
```

Itération sur ListeSimple



```
fonction avant_début(L)  
    retourner L.avant_premier
```

```
fonction début(L)  
    retourner suivant(L.avant_premier)
```

Supprimer une valeur en utilisant avant_début



```
fonction supprimer_valeurs(L, v)
    prec ← avant_début(L)
    m ← suivant(prec)
    tant que m != fin(L):
        m ← suivant(prec)
        si m != fin(L) et lire(m) == v
            L.erase_after(prec)
        sinon
            prec ← m
```

supprimer_valeurs(L, 2)

In: 2 → 1 → 3 → 2 → 1 → ∅

Out: 1 → 3 → 1 → ∅



```
struct MaillonVide {
    MaillonVide* suivant;
    MaillonVide(MaillonVide* n)
        : suivant(n) {}
    virtual ~MaillonVide() {}
};

struct Maillon : public MaillonVide {
    T valeur;
    Maillon(const T& v, MaillonVide* n)
        : valeur(v), MaillonVide{n} { }
};
```

```
MaillonVide avant_premier;
```

```
const T& lire(MaillonVide const * m) {
    return reinterpret_cast<Maillon const *>(m)->valeur;
}
```

- Distinguer les types `Maillon` et `MaillonVide` doit se faire par héritage
- Nous ne le ferons donc pas dans le cadre d'ASD

2ème étape



Maillon reste publique. Il faut :

- Cacher complètement la structure `Maillon`
- Retirer `Maillon*` des paramètres de toute méthode de `ListeSimple`

Classe itérateur



- Cacher le type `Maillon` en en faisant un attribut privé d'une classe
`Itérateur`

```
classe Itérateur  
    attribut privé:  
        Maillon* m
```

Classe itérateur



- Cacher le type `Maillon` en en faisant un attribut privé d'une classe `Itérateur`
- Mettre en oeuvre les opérateurs `++`, `*`, `==`, `!=`, ...

```
classe Itérateur
    attribut privé:
        Maillon* m

    méthode ++()
        m ← m.suivant

    méthode *()
        retourner m.valeur

    méthode ==(it)
        retourner m == it.m

    méthode !=(it)
        retourner m != it.m
```

Ré-écrivons ListSimple...



```
classe ListSimple

attributs: avant_premier ←
  MaillonVide(∅)

méthode insert_after(it, v)
  insérer_après(it.m, v)

méthode erase_after(it)
  supprimer_après(it.m)

méthode push_front(v)
  insert_after(before_begin(), v)

méthode pop_front()
  erase_after(before_begin())

méthode empty()
  retourner begin() = end()
```

```
méthode before_begin()
  retourner Itérateur(avant_début(L))

méthode begin()
  retourner Itérateur(début(L))

méthode end()
  retourner Itérateur(fin(L))
```

Ré-écrivons Tableau...



```
classe Tableau<T>

  attributs:
    data, taille, capacité

  méthode insert(it, v)
    insérer((it.a-data)/sizeof(T), v)

  méthode erase(it)
    supprimer((it.a-data)/sizeof(T))

  ...
```

```
méthode begin()
  retourner Itérateur(data)

méthode end()
  retourner Itérateur(data +
    taille*sizeof(T))
```

```
classe Itérateur
  attribut privé:
    Adresse a

  méthode ++()
    a ← a + sizeof(T)

  ...
```

Mise en oeuvre en C++

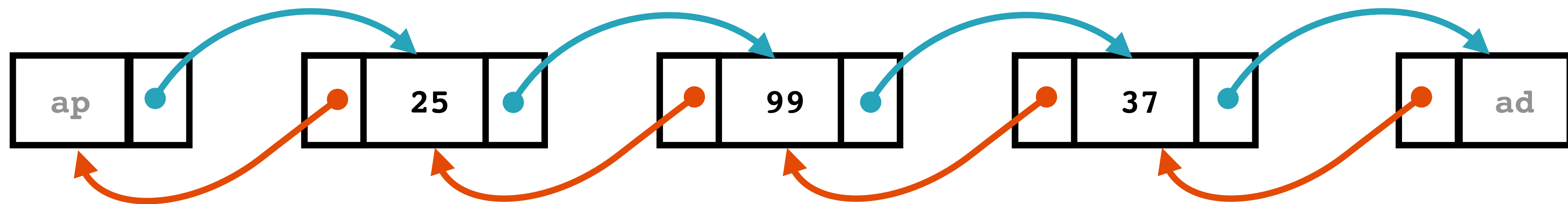


- Pour mettre en oeuvre un itérateur de liste simplement chaînée similaire à ceux de la STL, il faut ...
- Hériter de `std::iterator<std::forward_iterator_tag, T>`
- Mettre en oeuvre les deux `operator++`, `operator==`, `operator!=`, `operator*`, ...
- Mettre en oeuvre un `iterator` et un `const_iterator` et la conversion de `iterator` en `const_iterator`



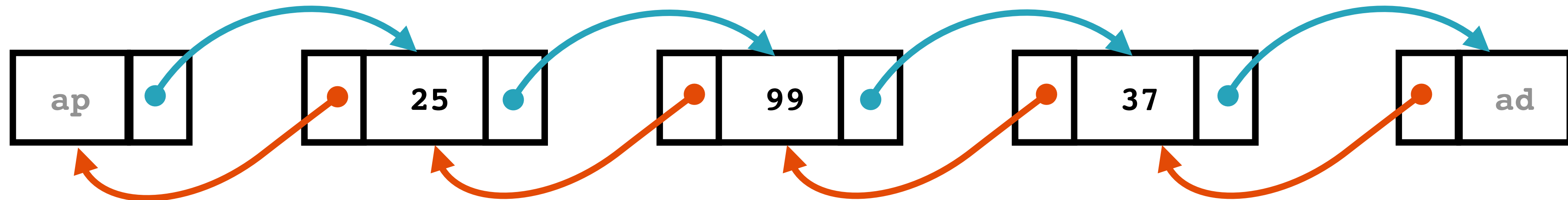
8. Listes doublement chaînées

Liste doublement chaînée



- Chaque maillon a un suivant *et* un précédent
- Il existe *deux* maillons “vides” : avant-premier (ap) et après-dernier (ad).

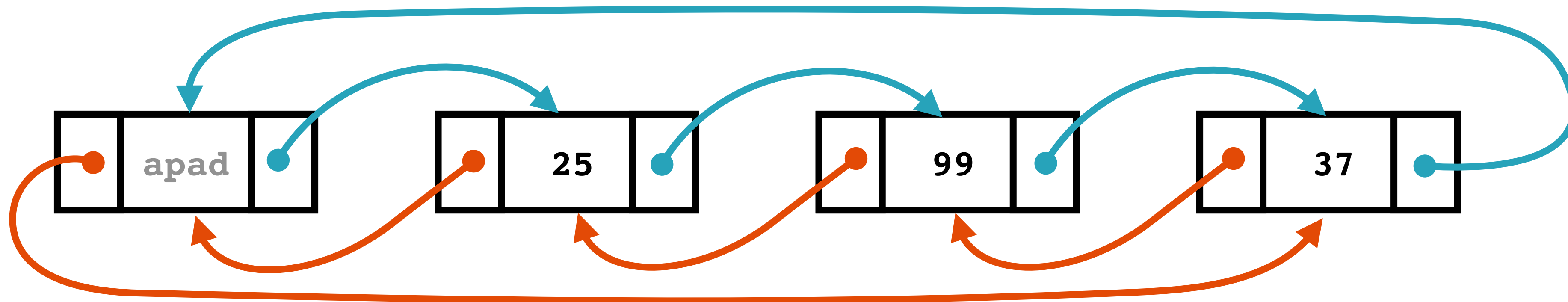
Liste doublement chaînée



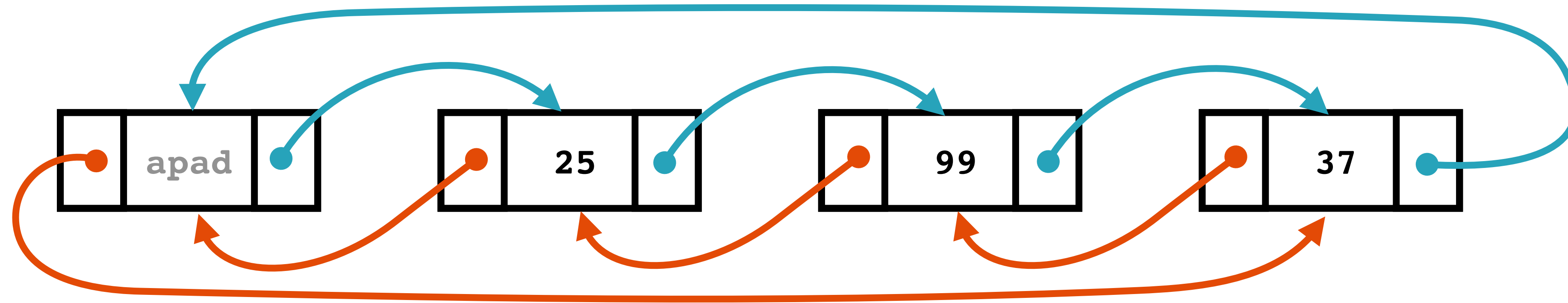
- Chaque maillon a un suivant *et* un précédent
- Il existe *deux* maillons “vides” : avant-premier (ap) et après-dernier (ad).

ou

- Il existe *un* maillon “vide” : avant-premier-après-dernier (apad)



Liste doublement chaînée

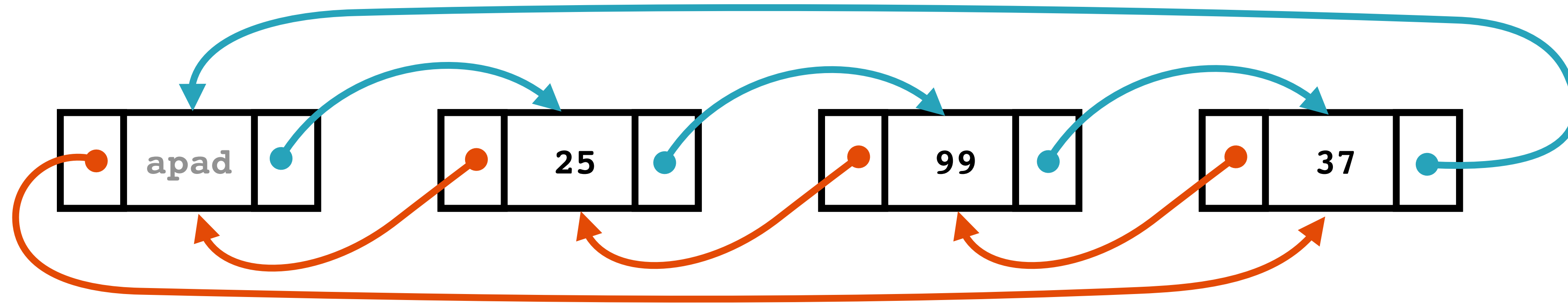


```
structure Maillon  
  Element valeur  
  Maillon* suivant  
  Maillon* précédent
```

```
structure MaillonVide  
  Maillon* suivant  
  Maillon* précédent
```

```
classe ListeDouble  
  attributs:  
    apad ← MaillonVide()  
    apad.précédent ← apad  
    apad.suivant ← apad
```

Liste doublement chaînée

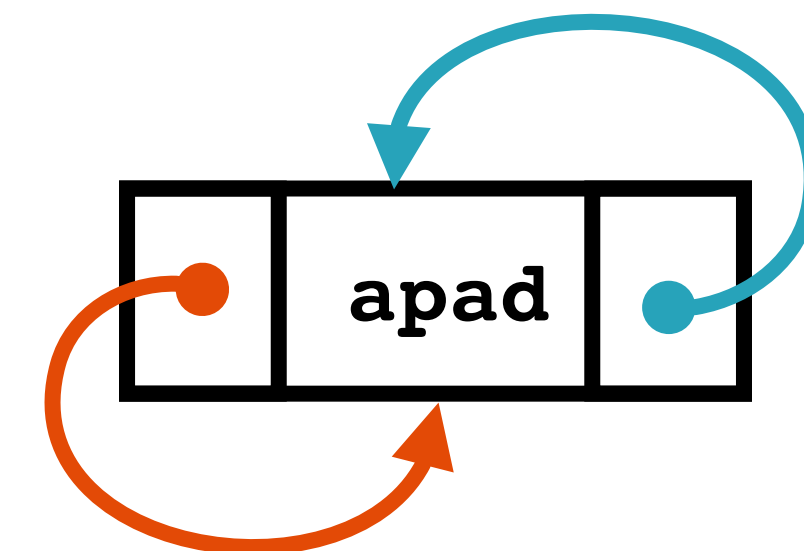


```
structure Maillon  
  Element valeur  
  Maillon* suivant  
  Maillon* précédent
```

```
structure MaillonVide  
  Maillon* suivant  
  Maillon* précédent
```

```
classe ListeDouble  
  attributs:  
    apad ← MaillonVide()  
    apad.précédent ← apad  
    apad.suivant ← apad
```

Donc, liste vide :



Notre plan cette fois...



- Commencer par définir l'itération
- Ecrire les 2 fonctions qui manipulent directement les Maillons
 - Insérer devant m
 - Supprimer m
- Ecrire le reste des fonctions en utilisant l'itération

Itération



```

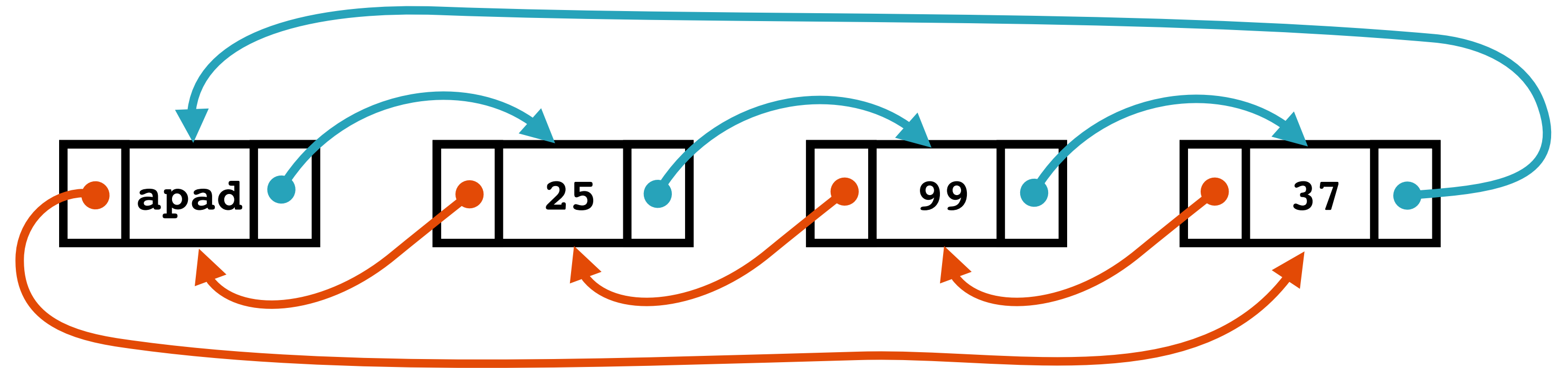
fonction suivant(m)
    retourner m.suivant

fonction précédent(m)
    retourner m.précédent

fonction *(m)
    retourner référence
        à m.valeur

fonction lire(m)
    retourner *m

fonction écrire(m, v)
    *m = v
    
```



```

classe ListeDouble
    ...
    fonction début()
        retourner apad.suivant

    fonction fin()
        retourner apad
    
```

Itération



```

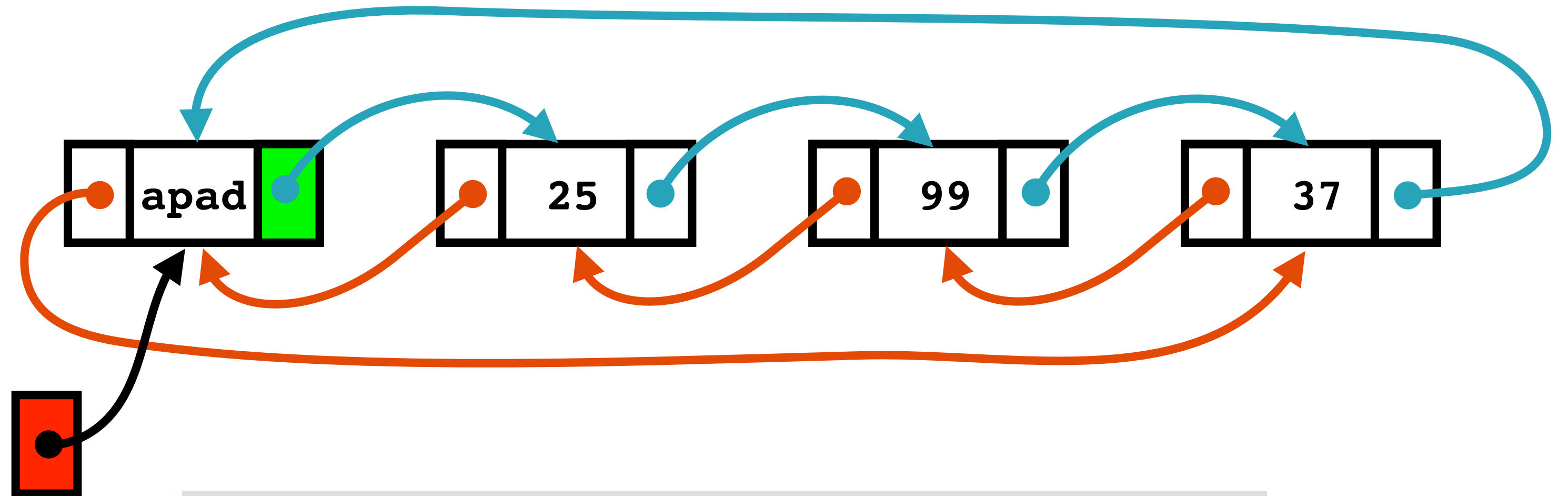
fonction suivant(m)
    retourner m.suivant

fonction précédent(m)
    retourner m.précédent

fonction *(m)
    retourner référence
        à m.valeur

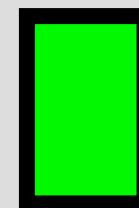
fonction lire(m)
    retourner *m

fonction écrire(m, v)
    *m = v
    
```

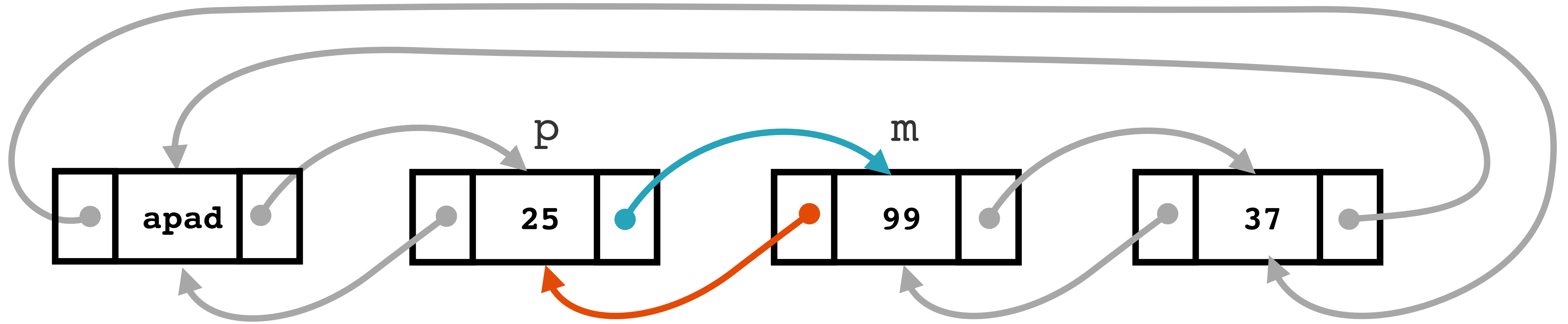


```

classe ListeDouble
    ...
    fonction début()
        retourner apad.suivant
    fonction fin()
        retourner apad
    
```



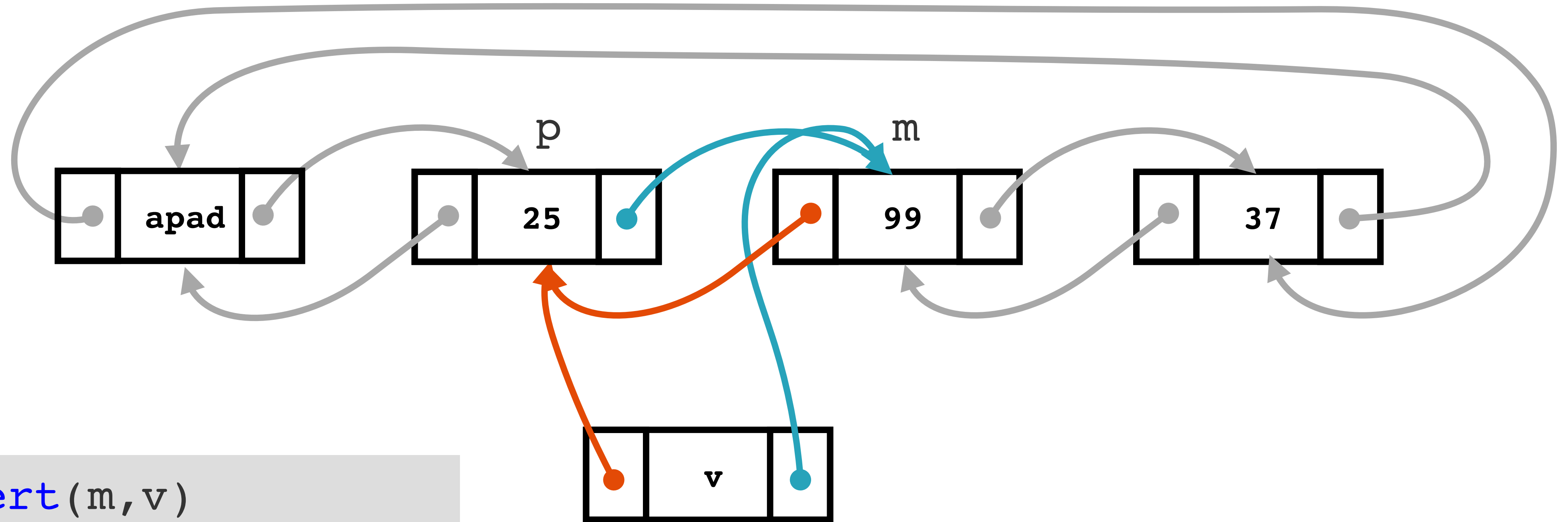
Insertion



```
méthode insert(m, v)
```

```
  p ← m.précédent
```

Insertion

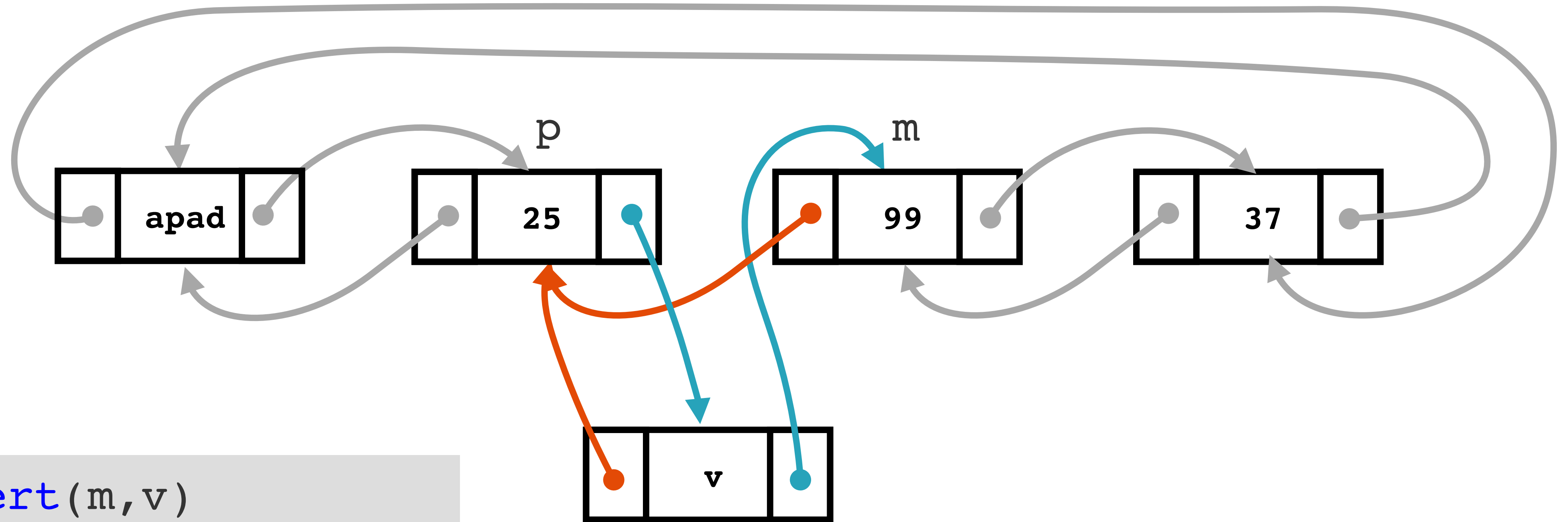


```

méthode insert(m, v)
  p ← m.précédent
  n ← nouveau Maillon(v, p, m)

```

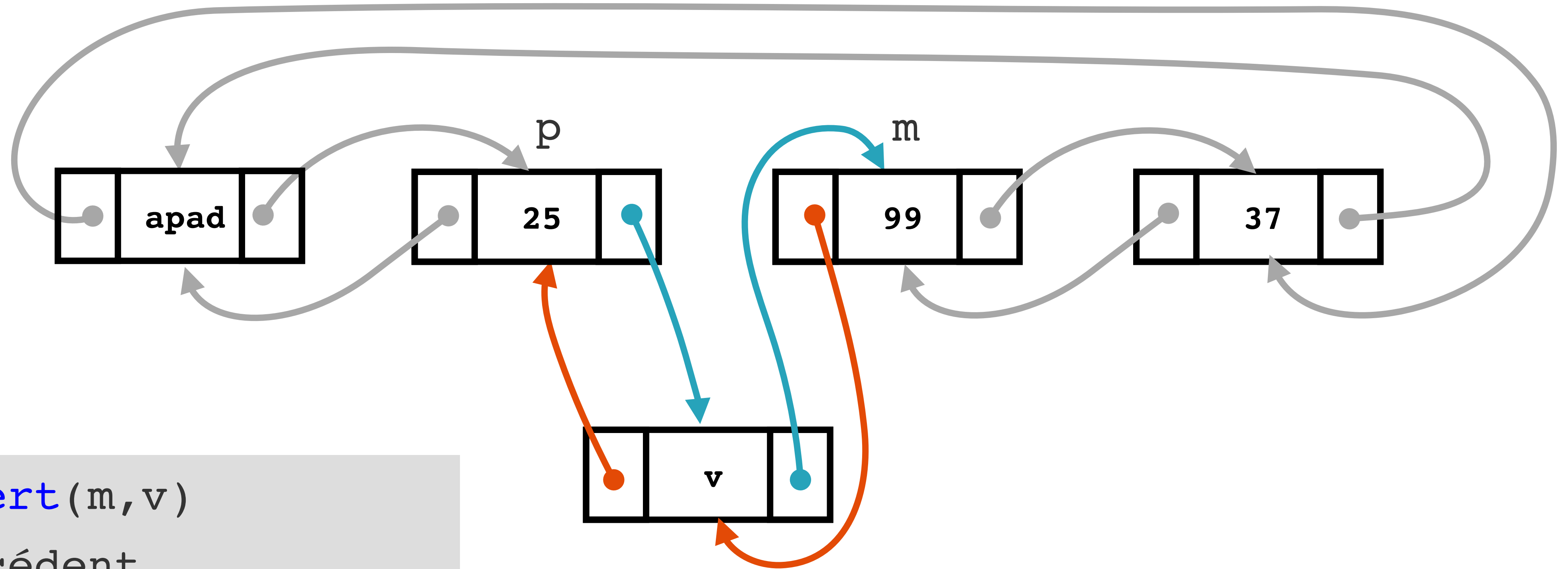
Insertion



```

méthode insert(m, v)
    p ← m.précédent
    n ← nouveau Maillon(v, p, m)
    p.suivant ← n
  
```

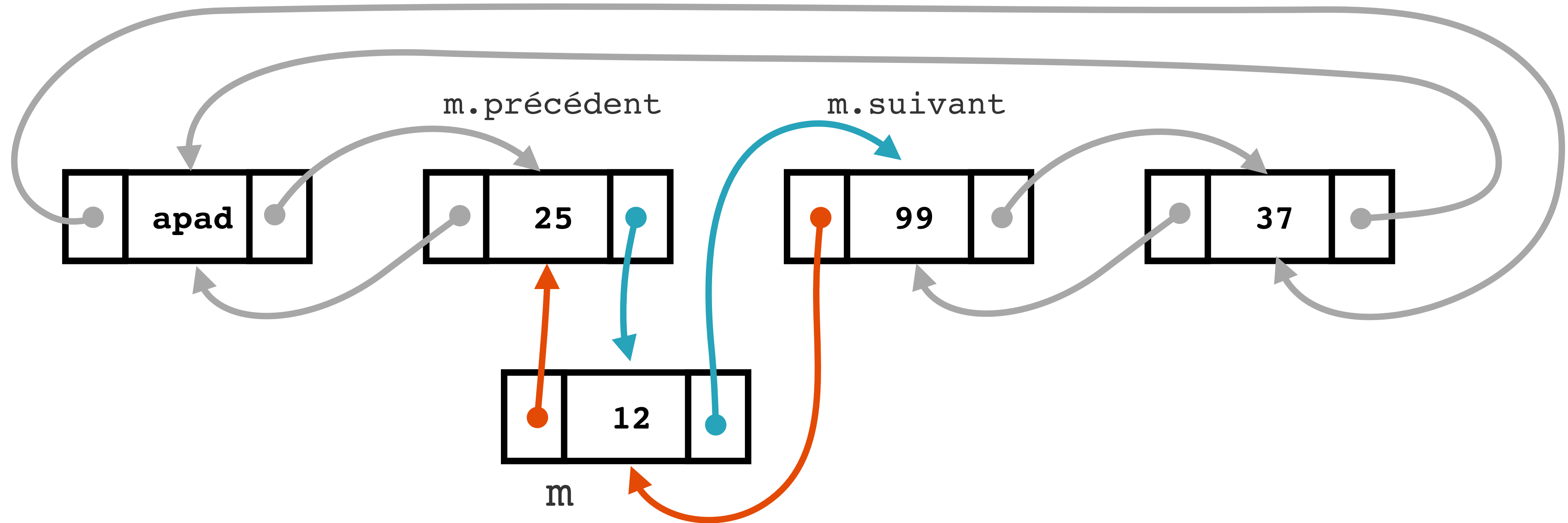
Insertion



```

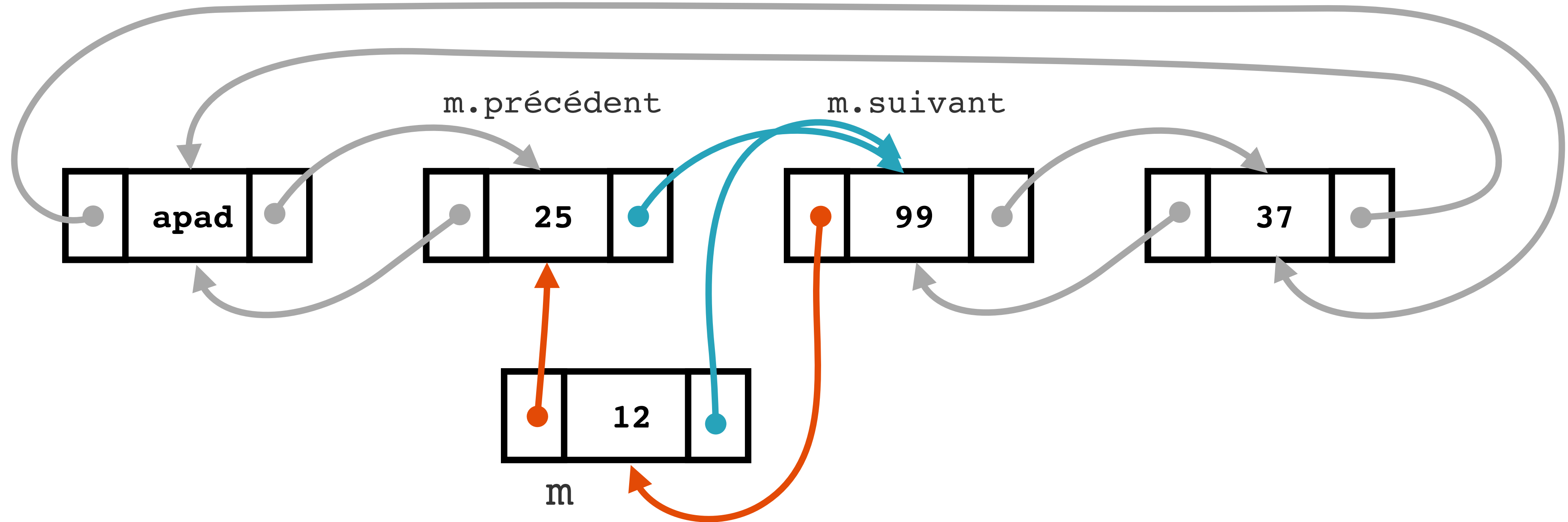
méthode insert(m, v)
  p ← m.précédent
  n ← nouveau Maillon(v, p, m)
  p.suivant ← n
  m.précédent ← n
  
```

Suppression



```
méthode erase(m)
```

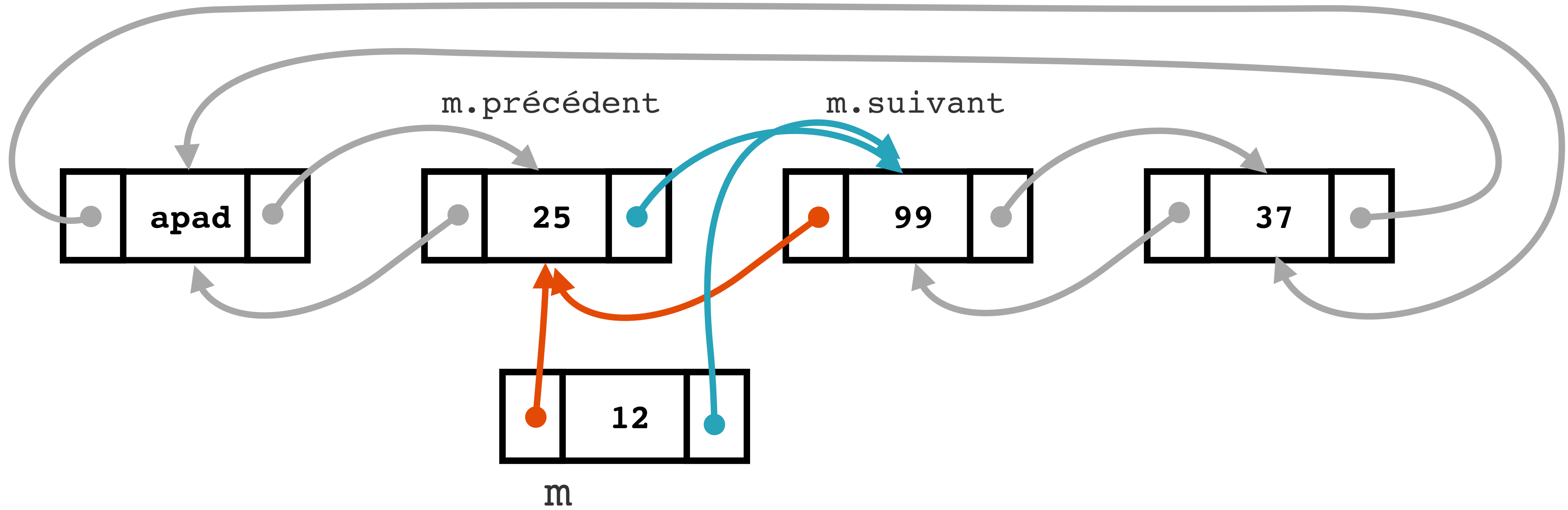
Suppression



```
méthode erase(m)
```

```
    m.précédent.suivant ← m.suivant
```

Suppression

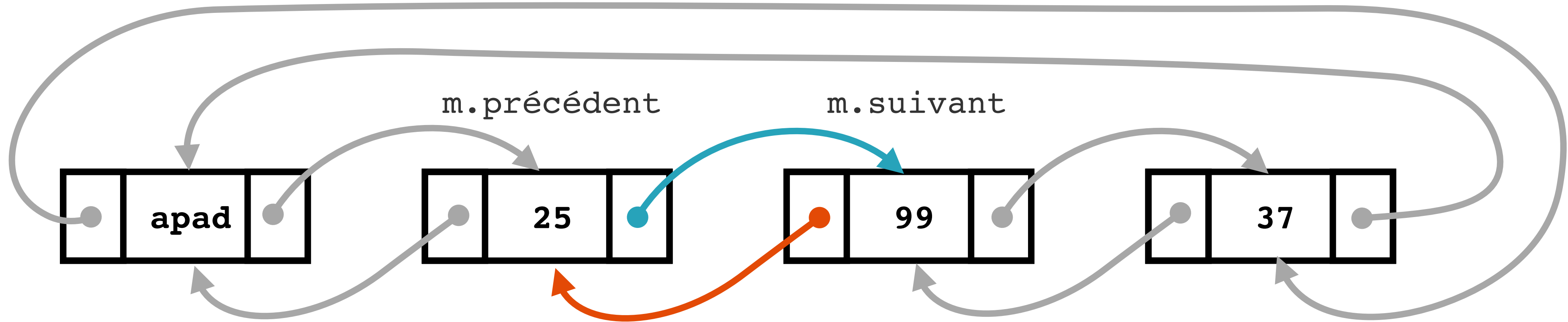


```
méthode erase(m)
```

```
  m.précédent.suivant ← m.suivant
```

```
  m.suivant.précédent ← m.précédent
```

Suppression



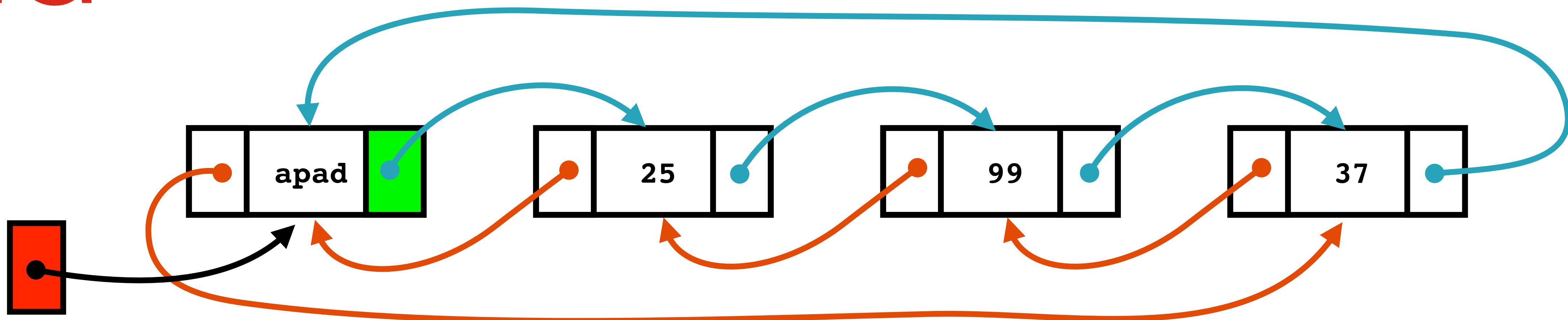
```
méthode erase(m)
```

```
    m.précédent.suivant ← m.suivant
```

```
    m.suivant.précédent ← m.précédent
```

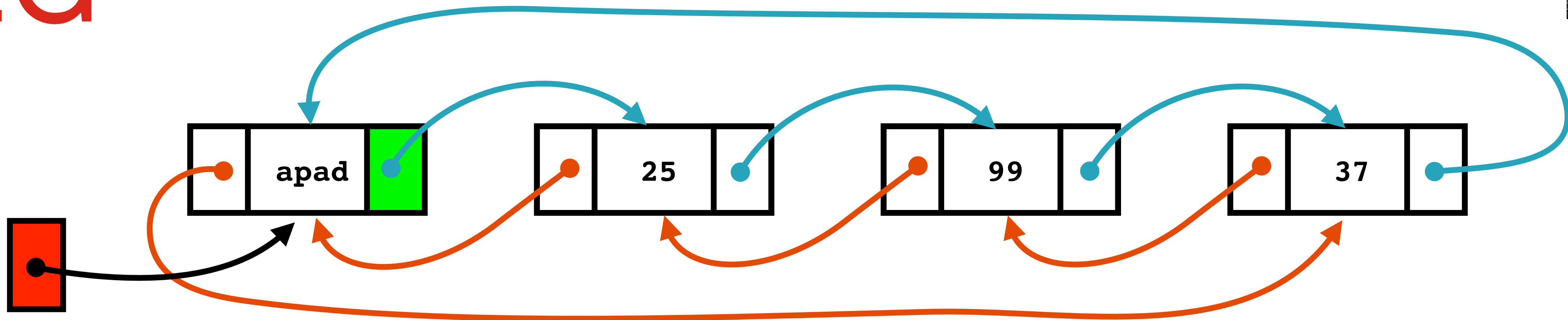
```
    effacer m
```

Opérations au début / en fin



- Le premier maillon de la liste L est début (L)
- Le dernier maillon de la liste L est précédent (fin (L))
- La fonction insert insère devant le maillon
- La fonction erase supprime le maillon

Opérations au début / en fin



```
méthode front()  
    retourner *(début())
```

```
méthode back()  
    retourner *(précédent(fin()))
```

```
méthode pop_front()  
    erase(début())
```

```
méthode pop_back()  
    erase(précédent(fin()))
```

```
méthode push_front(v)  
    insert(début(), v)
```

```
méthode push_back(v)  
    insert(fin(), v)
```

En conclusion



- Commencer par définir l'itération simplifie les choses
- Contrairement aux listes simples, on insère devant le maillon passé en paramètre
- Possible grâce au lien vers précédent
- Cohérent avec l'insertion dans les tableaux