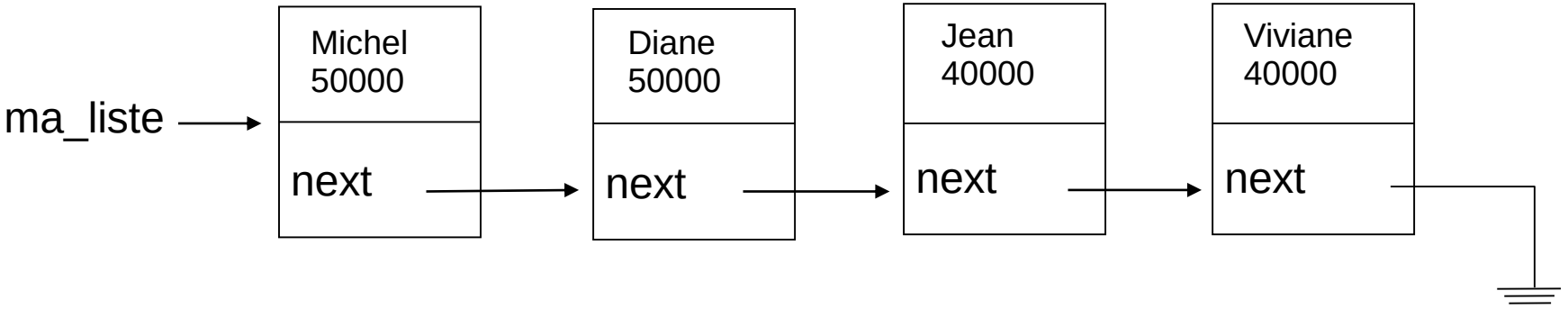
Programmation orientée objet

Listes liées

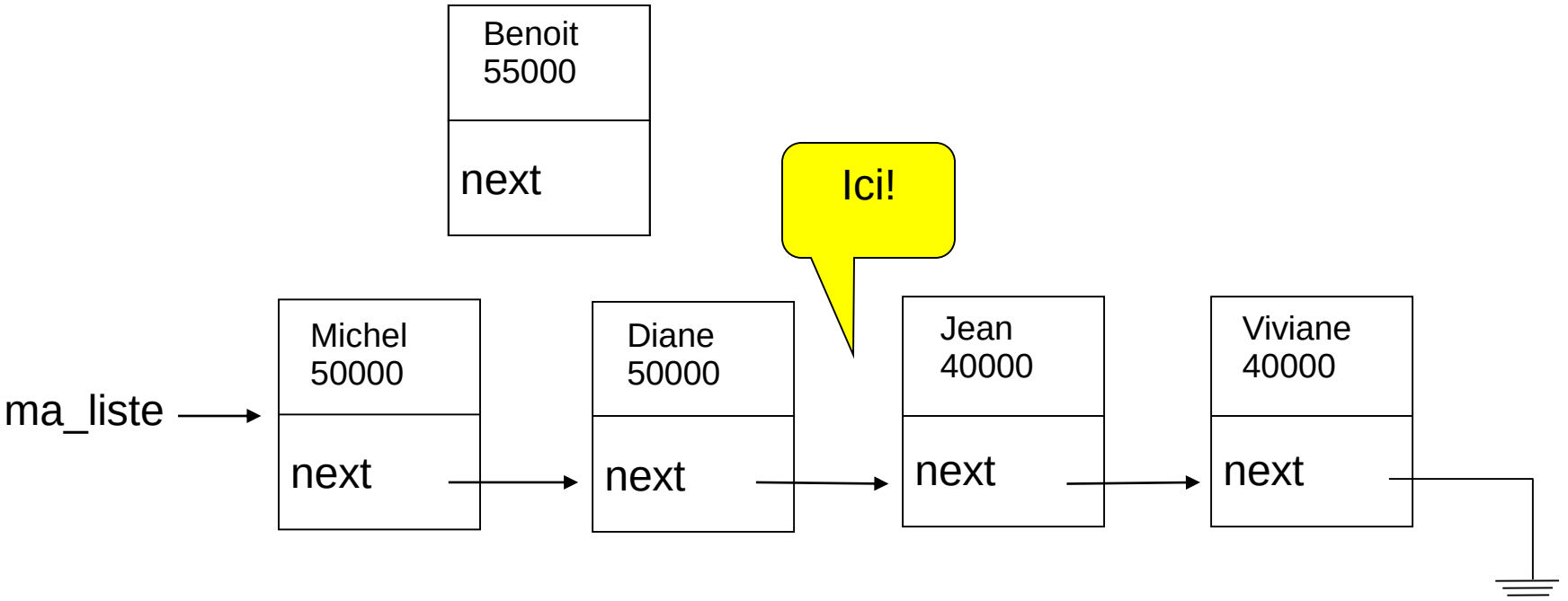
Motivation

- Imaginez un tableau contenant un certain nombre d'éléments
- On veut maintenant **ajouter un élément au milieu** (ou, pire encore, au début) du tableau
- Il faudra décaler tous les éléments à partir de cette position pour insérer le nouvel élément
- Il s'agit d'une procédure coûteuse
- Solution: **liste liée**

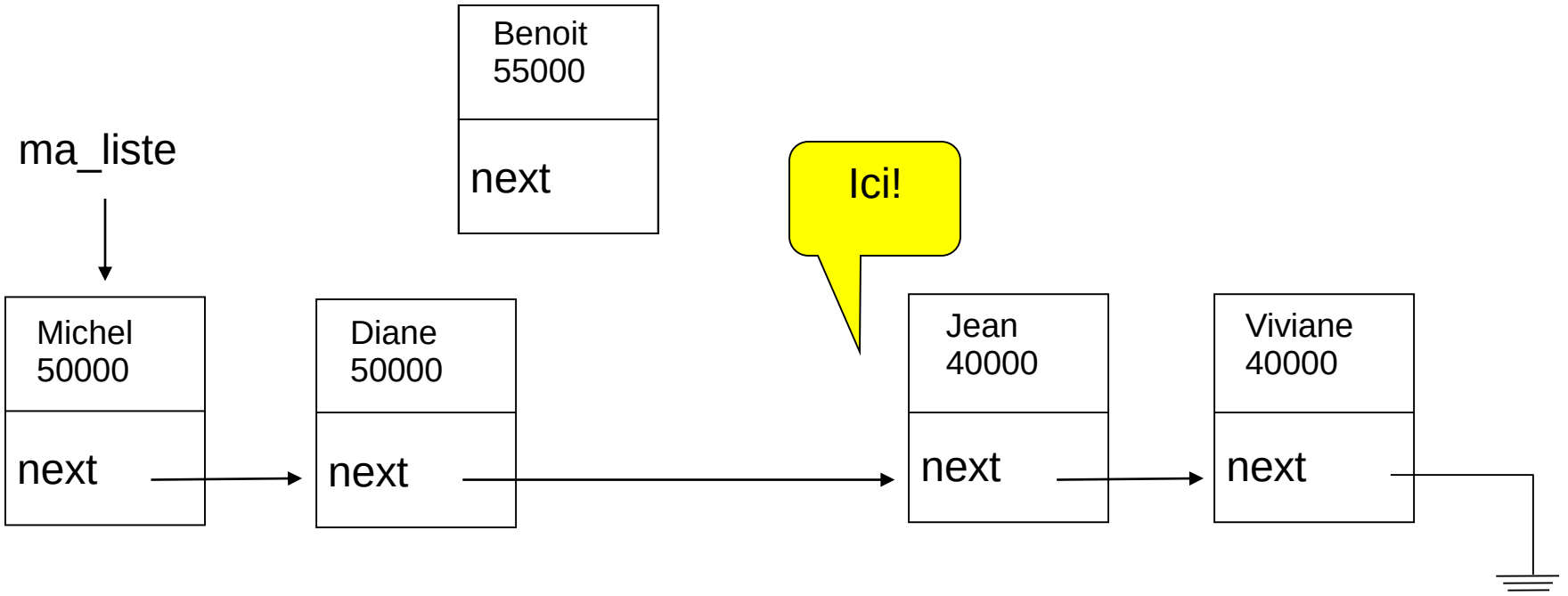
Liste liée simple



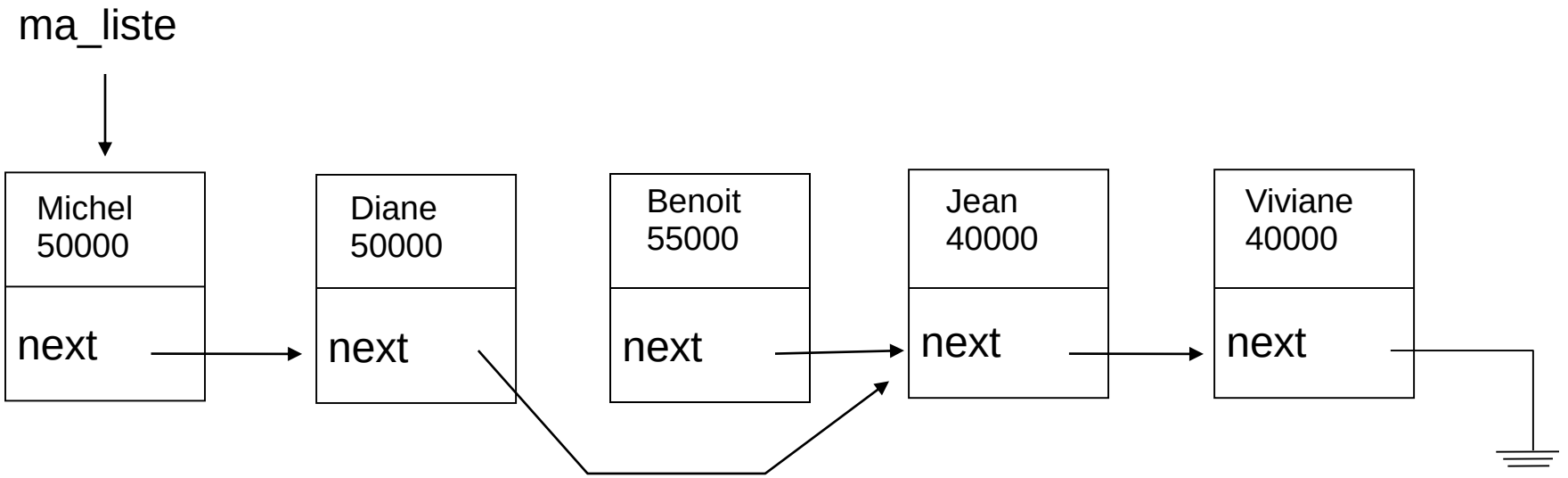
Ajout d'un élément



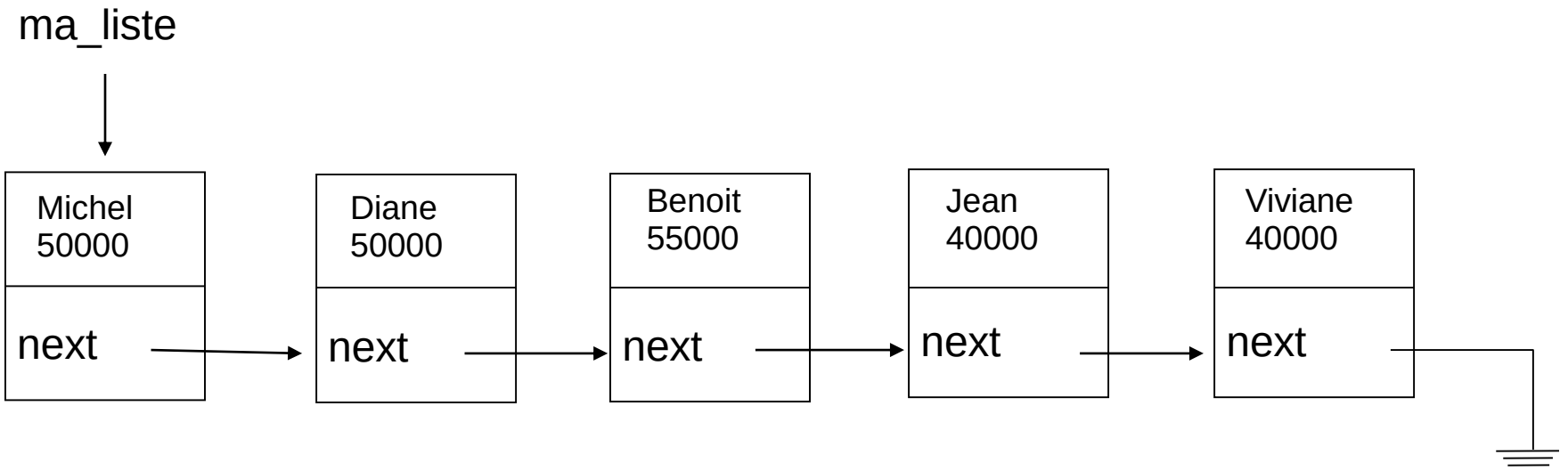
Ajout d'un élément (suite)



Ajout d'un élément (suite)



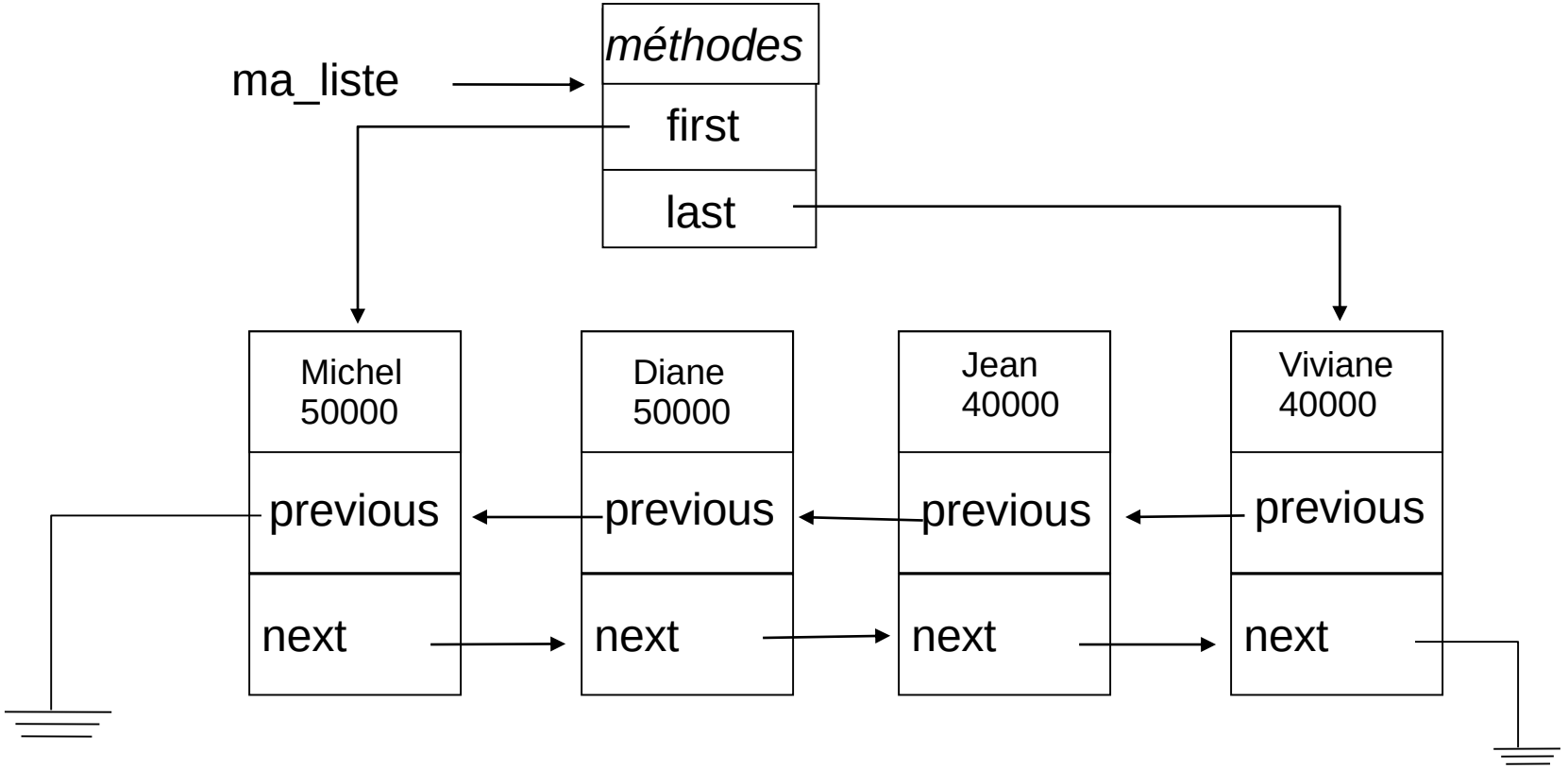
Ajout d'un élément (suite)



Ajout d'un élément

- On voit donc que **l'ajout** d'un élément est peu coûteux
- Il suffit de **deux affectations de pointeurs**
- Il y a un **prix** à payer par contre: pour trouver l'endroit d'insertion il faut **parcourir la liste à partir du début**

Liste à liens doubles



Implémentation de listes simples

- Pour implémenter les listes, il faut avoir trois classes:
  - La classe **Node**, pour représenter un élément de la liste à l'intérieur
  - La classe **List**, pour représenter la liste
  - La classe **Iterator**, pour implémenter le parcours de la liste

Classe Node - définition

```
class Node
{
public:
   Node(Employee* e);
private:
   Employee* employee_;
   Node* next_;
   Node* previous_;//liens doubles

   friend class List;
   friend class Iterator;
}
```

La déclaration *friend* permet à tout objet de classe List ou Iterator de manipuler directement les attributs privés d'un objet de la classe Node

Remarques sur *friend*

- Nécessaire ici parce que les classes List et Iterator sont intimement liées à la classe Node
- **Node n'est en fait qu'une implémentation des données qui restera cachée à l'utilisateur** de la liste (il n'a aucune méthode publique à part le constructeur)
- On aurait pu implémenter les listes sans utiliser *friend* en implémentant des méthodes (voir exemple sur le site du cours)
- Alors: *friend* ne brise pas l'encapsulation dans ce cas-ci

Itérateur

- Un itérateur est un «pointeur intelligent»
- Il «pointe» vers un élément de la liste
- On peut lui demander de nous retourner la valeur de l'élément pointé
- On peut lui demander de se déplacer à l'élément suivant ou précédent
- Finalement, on peut vérifier si deux itérateurs pointent vers le même élément

Classe Iterator - définition

```
class Iterator
{
public:
   Iterator();
   Employee* get() const;
   void next();
   void previous();//liens doubles
   bool equals(Iterator b) const;
private:
   Node* position_;

   friend class List;
}
```

Si on appelle next() alors qu'on est à la fin de la liste, **position** aura alors la valeur 0.

Classe Iterator - implémentation

```
void Iterator::next()
{
   assert(position_ != 0);
   position_ = position_->next;
}
```

Remarquez qu'on accède directement à l'attribut privé d'un objet de la classe Node

Classe Iterator – implémentation (suite)

```
Employee* Iterator::get() const
{
   assert(position_ != 0);
   return position_->employee_;
}

bool Iterator::equals(Iterator b) const
{
   return position_ == b.position_;
}
```

Pourquoi ce const?

Classe List - Définition

```
class List {
public:
   List();
   ~List(); // détruit contenu par valeur!
   void push_back(Employee* s) //pour insérer à la fin
   void insert(Iterator pos, Employee* s)
   Iterator erase(Iterator pos);
   Iterator begin(); // retourne un itér. qui pointe
                     // sur le premier item
   Iterator end();   // retourne un itér. qui pointe
                     // après le dernier item
private:
   Node* first_;
   Node* last_;
}
```

Parcours d'une liste

- On demande d'abord à la liste de nous retourner un itérateur qui pointe à son premier élément
- On demande aussi un itérateur qui pointe *après* le dernier élément
- Soient **pos** et **fin** ces deux itérateurs, respectivement

Parcours d'une liste (suite)

- Tant que **pos** est différent de **fin** (pour vérifier cela on appelle la méthode equals()):
  - Vérifier si l'item pointé par **pos** est celui qu'on recherche
  - Si c'est le cas, on arrête
  - Sinon on fait pointer **pos** à l'item suivant

Exemple de parcours de liste

```
int main()
{
   List staff;
   staff.push_back(new Employee("Michel”,50000));
   staff.push_back(new Employee("Roberta”,5000));
   staff.push_back(new Employee("Claudia”,45000));
   staff.push_back(new Employee("Mohamed”,45000));

   Iterator pos = staff.begin();
   Iterator fin = staff.end();
    while (!pos.equals(fin) &&
           pos.get()->getName() != "Claudia") {
      pos.next();
   };
   if (!pos.equals(fin)){
      staff.erase(pos);
      ...
```

Exemple de parcours de liste

```
int main()
{
    List staff;
    staff.push_back(new Employee("Michel”,50000));
    staff.push_back(new Employee("Roberta”,5000));
    staff.push_back(new Employee("Claudia”,45000));
    staff.push_back(new Employee("Mohamed”,45000));

    Iterator pos = staff.begin();
    Iterator fin = staff.end();
    while (!pos.equals(fin) &&
            pos.get()->getName() != "Claudia") {
       pos.next();
    };
    if (!pos.equals(fin)){
       staff.erase(pos);
       ...
```

On remplit la liste

Exemple de parcours de liste

```
int main()
{
    List staff;
    staff.push_back(new Employee("Mic
    staff.push_back(new Employee("Rob
    staff.push_back(new Employee("Cla
    staff.push_back(new Employee("Mo
    Iterator pos = staff.begin();
    Iterator fin = staff.end();
    while (!pos.equals(fin) &&
           pos.get()->getName() != "Claudia") {
      pos.next();
    };
    if (!pos.equals(fin)){
       staff.erase(pos);
       ...
```

On initialise un itérateur qui pointe sur le premier élément de la liste et un autre qui pointe après le dernier élément de la liste

Exemple de parcours de liste

```
int main()
{
    List staff;
    staff.push_back(new Employee("Michel”,50000));
    staff.push_back(new Employee("Roberta”,5000));
    staff.push_back(new Employee("Claudia”,45000));
    staff.push_back(new Employee("Mohamed”,45000));

    Iterator pos = staff.begin();
    Iterator fin = staff.end();
    while (!pos.equals(fin) &&
           pos.get()->getName() != "Claudia") {
      pos.next();
    };
    if (!pos.equals(fin)){
       staff.erase(pos);
       ...
```

On parcourt la liste jusqu’ à ce qu’ on trouve l’ élément recherché

Exemple de parcours de liste

```
int main()
{
   List staff;
   staff.push_back(new Employee("Michel”,50000));
   staff.push_back(new Employee("Roberta”,5000));
   staff.push_back(new Employee("Claudia”,45000));
   staff.push_back(new Employee("Mohamed”,45000));

   Iterator pos = staff.begin();
   Iterator fin = staff.end();
   while (!pos.equals(fin) &&
          pos.get()->getName() != "Claudia") {
     pos.next();
   };
   if (!pos.equals(fin)){
      staff.erase(pos);
      ...
```

L'itérateur pointe sur l'élément recherché, que l'on peut maintenant effacer

Classe List - implémentation

```
Iterator List::begin()
{
    Iterator iter;
    iter.position_ = first;
    return iter;
}

Iterator List::end()
{
    Iterator iter;
    iter.position_ = 0;
    return iter;
}
```

On crée un itérateur qui pointe au premier élément et on retourne cet itérateur

On crée un itérateur dont la position est 0. Dans notre implémentation, cela signifie qu'on est positionné *après* le dernier élément

Classe List – insertion d'un item à la fin

```
void List::push_back(Employee* s)
{
   Node* newnode = new Node(s);
   if (last_ == 0) /* la liste est vide  */
   {
      first_ = newnode;
      last_ = newnode;
   }
   else
   {
      last_->next_ = newnode;
     newnode->previous_ = last_;
      last_ = newnode;
   }
}
```

On crée un nouveau noeud

Si la liste est vide, les deux pointeurs pointeront sur le nouveau noeud

L'« ancien » dernier noeud doit maintenant pointer sur le nouveau noeud

La liste doit maintenant pointer sur le « nouveau » dernier noeud

Classe List – Insertion à la position indiquée par l'itérateur

```
void List::insert(Iterator iter, Employee* s)
{
   if (iter.position_ == 0)
   {
      push_back(s);
      return;
   }

   ...
```

Si la position demandée est après la fin de la liste, on ne se complique pas la vie et on appelle la méthode push_back()

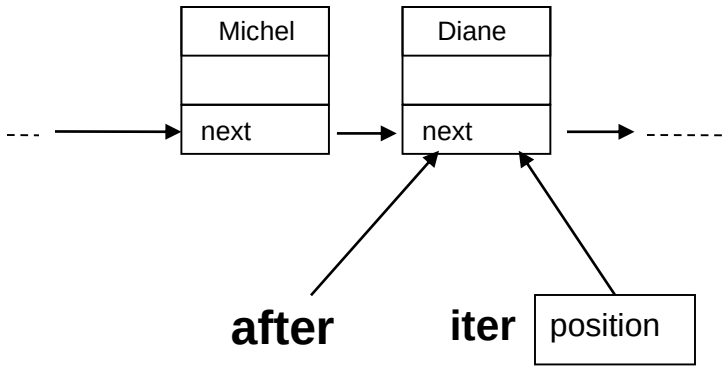
Insertion à la position indiquée par l'itérateur (suite)

```
  ...
  Node* after = iter.position_;

  Node* before = first_;
  while(before->next_ != after) before = before->next_;

  Node* newnode = new Node(s);
  newnode->next_ = after;
  before->next_ = newnode;
}
```

On crée un pointeur pour désigner le noeud qui se trouve à la position d'insertion



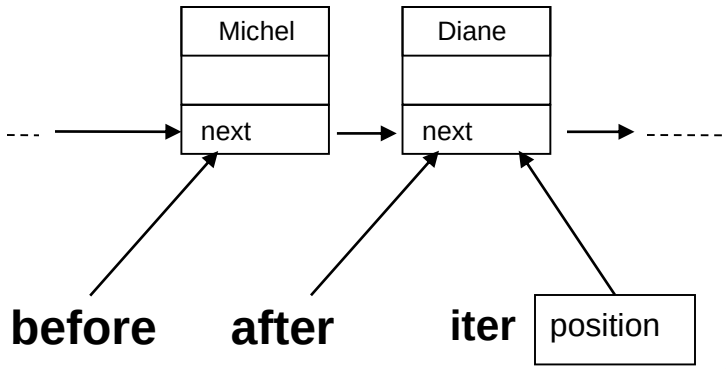
Insertion à la position indiquée par l'itérateur (suite)

```
  ...
  Node* after = iter.position;

  Node* before = first_;
  while(before->next_ != after) before = before->next_;

  Node* newnode = new Node(s);
  newnode->next_ = after;
  before->next_ = newnode;
}
```

On cherche le nœud devant le nouveau nœud

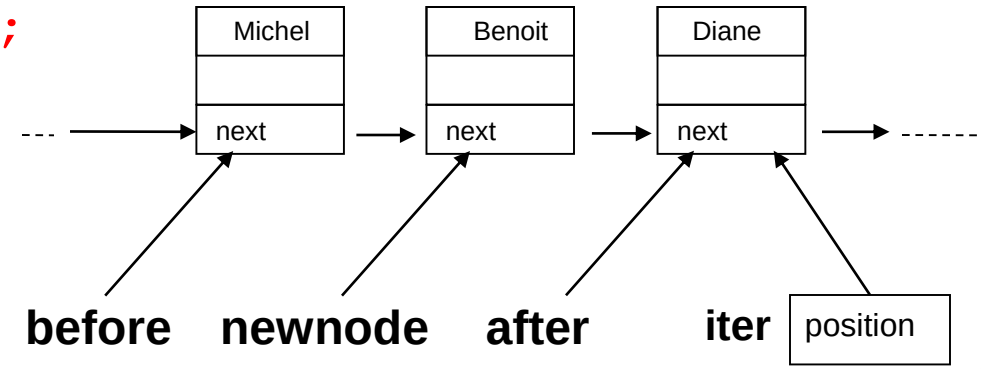


Insertion à la position indiquée par l'itérateur (suite)

```
  ...
  Node* after = iter.position;

  Node* before = first_;
  while(before->next_ != after) before = before->next_;

  Node* newnode = new Node(s);
  newnode->next_ = after;
  before->next_ = newnode;
}
```

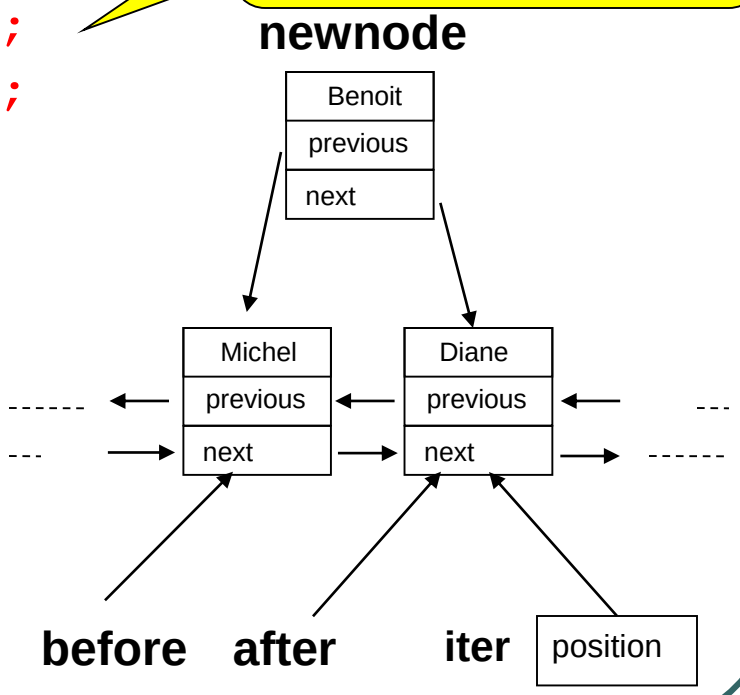


Insertion à la position indiquée par l'itérateur (suite)

```
  ...
  Node* after = iter.position;
  Node* before = after->previous_;
  Node* newnode = new Node(s);
  newnode->previous_ = before;
  newnode->next_ = after;
  after->previous_ = newnode;
  if (before == 0)
     first_ = newnode;
  else
     before->next_ = newnode;
}
```

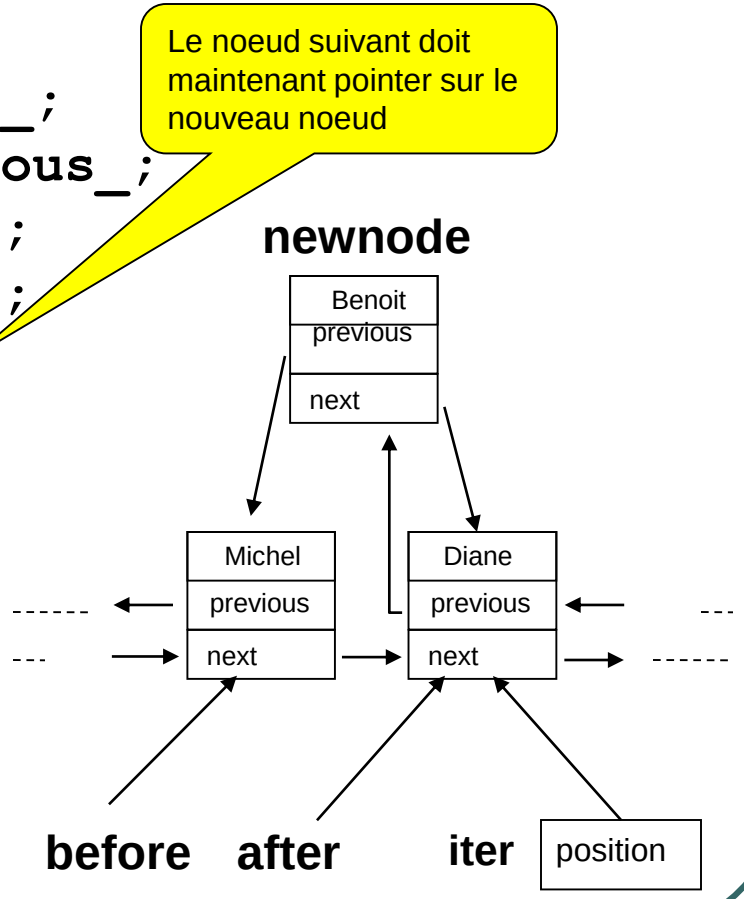
Aucun besoin de la boucle while
Utilisation du pointeur previous

On crée le nouveau noeud et on le fait pointer vers les noeuds qui seront son précédent et son suivant



Insertion à la position indiquée par l'itérateur (suite)

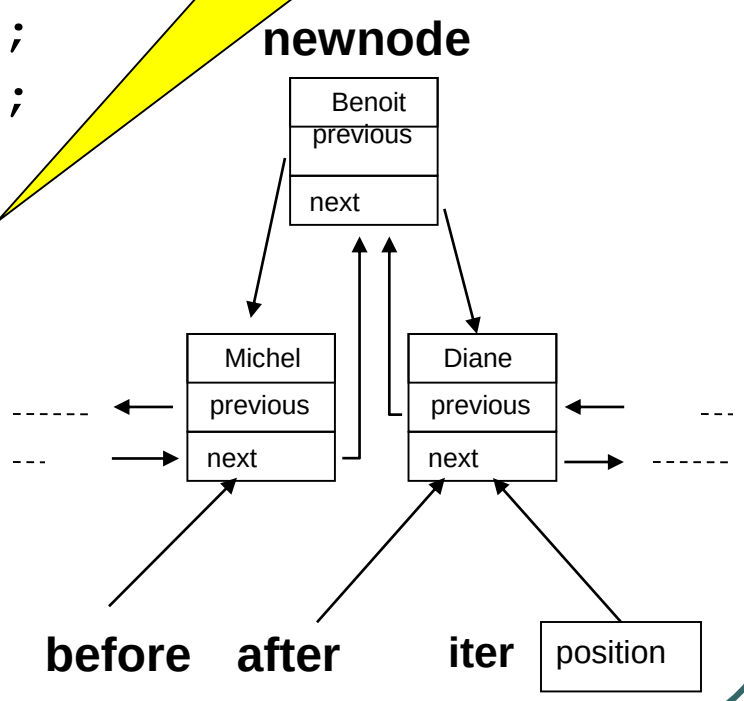
```
  ...
  Node* after = iter.position_;
  Node* before = after->previous_;
  Node* newnode = new Node(s);
  newnode->previous_ = before;
  newnode->next_ = after;
  after->previous_ = newnode;
  if (before == 0)
     first_ = newnode;
  else
     before->next_ = newnode;
}
```



Insertion à la position indiquée par l'itérateur (suite)

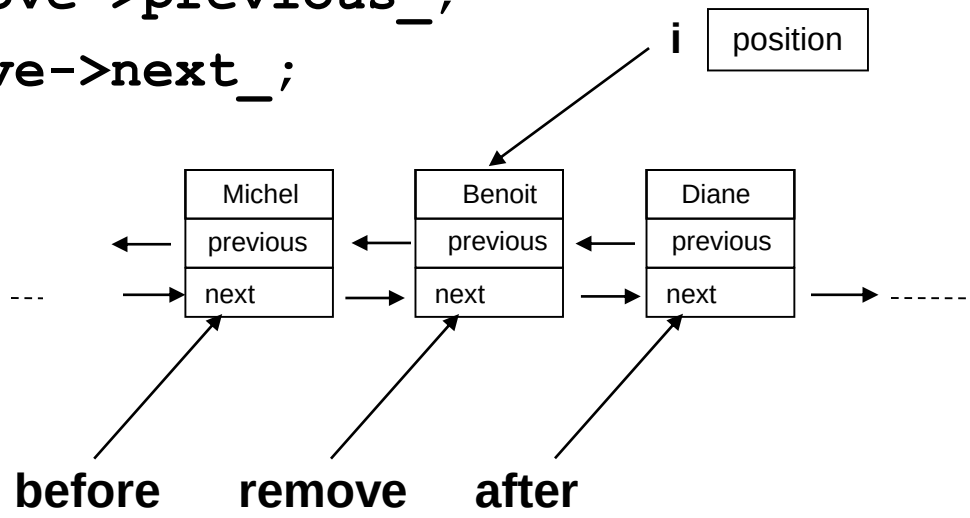
```
  ...
  Node* after = iter.position_;
  Node* before = after->previous_;
  Node* newnode = new Node(s);
  newnode->previous_ = before;
  newnode->next_ = after;
  after->previous_ = newnode;
  if (before == 0)
     first_ = newnode;
  else
     before->next_ = newnode;
}
```

Le noeud précédent, s'il existe, doit lui aussi pointer sur le nouveau noeud



Classe List – Retrait à la position indiquée par l'itérateur

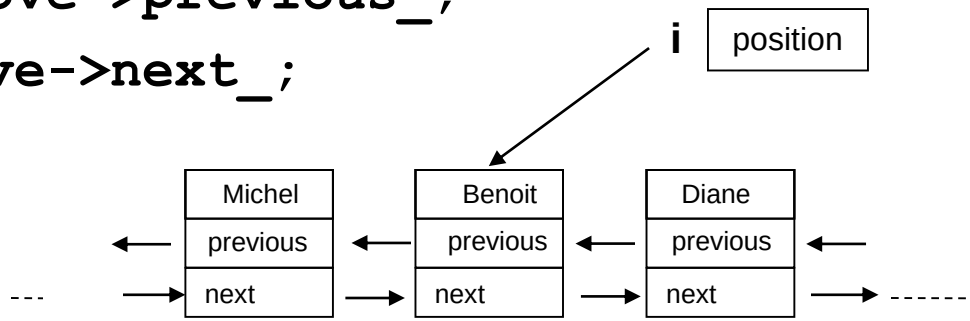
```
Iterator List::erase(Iterator i)
{
   assert(i.position_ != 0);
   Node* remove = i.position_;
   Node* before = remove->previous_;
   Node* after = remove->next_;
   ...
```



Classe List – Retrait à la position indiquée par l'itérateur

```
Iterator List::erase(Iterator i)
{
    assert(i.position_ != 0);
    Node* remove = i.position_;
    Node* before = remove->previous_;
    Node* after = remove->next_;
    ...
```

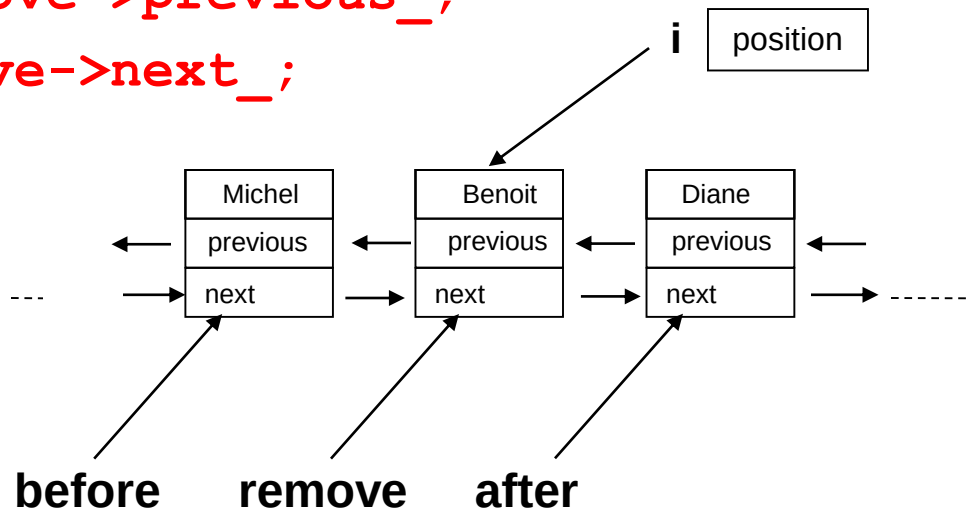
Si on esaie de faire un retrait après la fin de la liste, il s'agit évidemment d'une erreur.



Classe List – Retrait à la position indiquée par l'itérateur

```
Iterator List::erase(Iterator i)
{
    assert(i.position_ != 0);
    Node* remove = i.position_;
    Node* before = remove->previous_;
    Node* after = remove->next_;
    ...
```

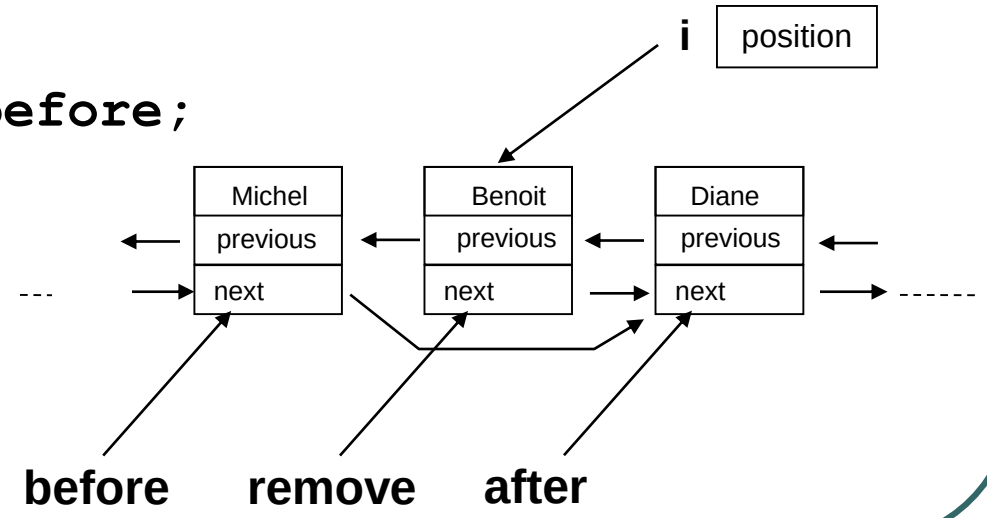
On pointe sur le noeud à effacer ainsi que son suivant et son précédent



Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
   first_ = after;
else
   before->next_ = after;
if (remove == last_)
   last_ = before;
else
   after->previous_ = before;
i.position_ = after;
delete remove;
remove = 0;
return i;
}
```

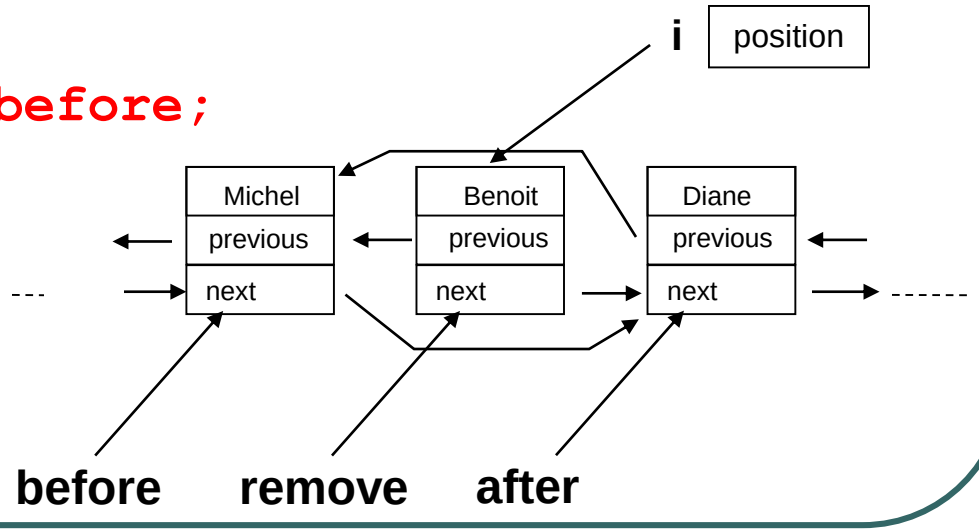
Si l'item retiré est le premier, on fait pointer le pointeur **first** de la liste sur l'item suivant. Sinon, on fait pointer le noeud précédent sur le noeud suivant.



Classe List – Retrait à la position indiquée par l'itérateur

```
    ...
    if (remove == first_)
       first_ = after;
    else
       before->next_ = after;
    if (remove == last_)
       last_ = before;
    else
       after->previous_ = before;
    i.position_ = after;
    delete remove;
    remove = 0;
    return i;
}
```

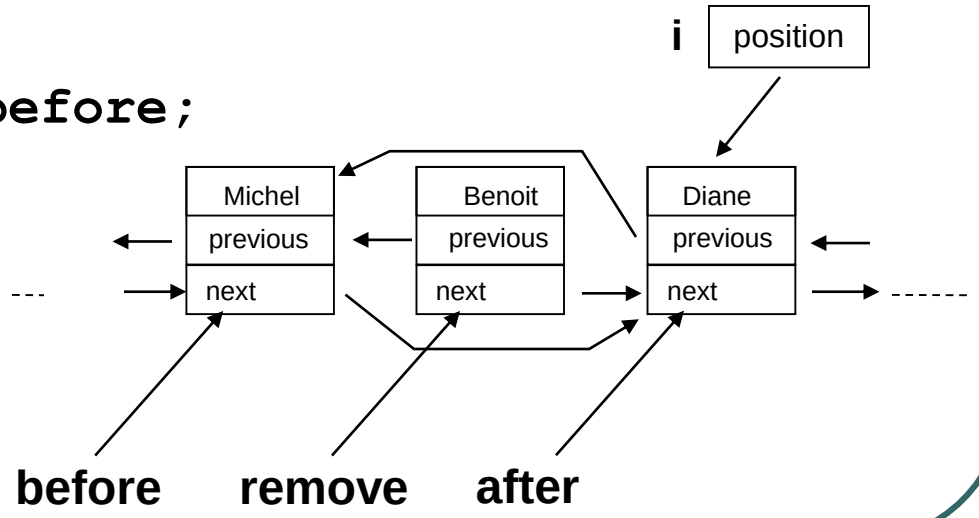
Si l'item retiré est le dernier, on fait pointer le pointeur **last** de la liste sur l'item précédent. Sinon, on fait pointer le noeud suivant sur le noeud précédent.



Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
   first_ = after;
else
   before->next_ = after;
if (remove == last_)
   last_ = before;
else
   after->previous_ = before;
i.position_ = after;
delete remove;
remove = 0;
return i;
}
```

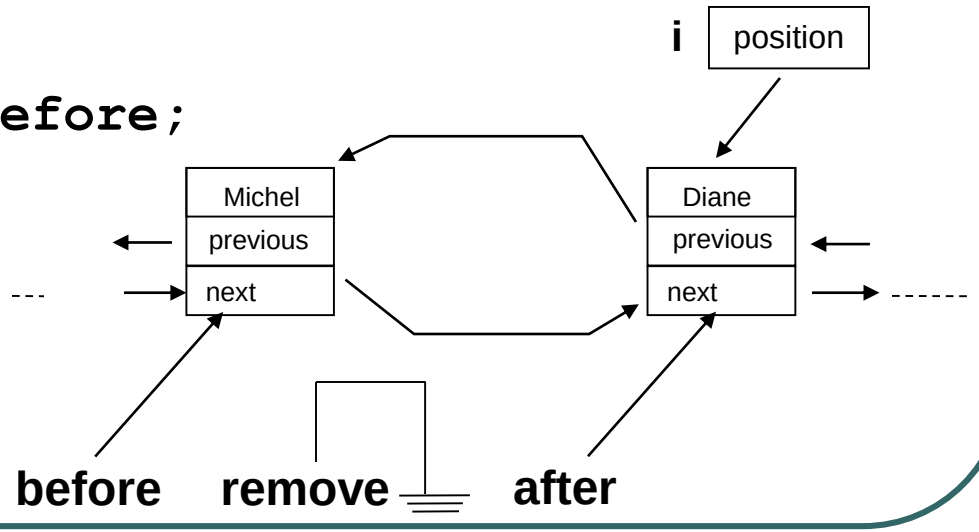
Le nouvel itérateur pointera sur le noeud qui suit le noeud retiré



Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
   first_ = after;
else
   before->next_ = after;
if (remove == last_)
   last_ = before;
else
   after->previous_ = before;
i.position_ = after;
delete remove;
remove = 0;
return i;
}
```

On peut maintenant éliminer le noeud retiré



Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
   first_ = after;
else
   before->next_ = after;
if (remove == last_)
   last_ = before;
else
   after->previous_ = before;
i.position_ = after;
delete remove;
remove = 0;
return i;
}
```

On retourne le nouvel itérateur, qui pointe sur l'item qui suit l'item retiré